Measure and Probability Theory

December 12, 2021

Contents

1 Probability measure **3**
- 1.1 Introduce binder for probability . . . 8
- 1.2 Distributions . . . 14

2 Distribution Functions **32**
- 2.1 Properties of cdf's . . . 32
- 2.2 Uniqueness . . . 35

3 Weak Convergence of Functions and Distributions **38**

4 Weak Convergence of Functions **38**

5 Weak Convergence of Distributions **38**

6 Skorohod's theorem **39**

7 The Giry monad **47**
- 7.1 Sub-probability spaces . . . 47
- 7.2 Properties of "return" . . . 59
- 7.3 Join . . . 64
- 7.4 Giry monad on probability spaces . . . 82

8 Projective Family **87**

9 Infinite Product Measure **103**
- 9.1 Sequence space . . . 109

10 Independent families of events, event sets, and random variables **113**

11 Convolution Measure **144**

12 Information theory **149**
12.1 Information theory . . . 150
12.2 Kullback−Leibler divergence . . . 151
12.3 Finite Entropy . . . 156
12.4 Mutual Information . . . 158
12.5 Entropy . . . 166
12.6 Conditional Mutual Information . . . 170
12.7 Conditional Entropy . . . 184
12.8 Equalities . . . 188

13 Properties of Various Distributions **194**
13.1 Erlang . . . 196
13.2 Exponential distribution . . . 202
13.3 Uniform distribution . . . 208
13.4 Normal distribution . . . 212

14 Characteristic Functions **226**
14.1 Application of the FTC: integrating e^ix . . . 227
14.2 The Characteristic Function of a Real Measure. . . . 227
14.3 Independence . . . 228
14.4 Approximations to e^{ix} . . . 229
14.5 Calculation of the Characteristic Function of the Standard Distribution . . . 237

15 Helly's selection theorem **239**

16 Integral of sinc **245**
16.1 Various preparatory integrals . . . 246

17 The sinc function, and the sine integral (Si) **248**
17.1 The final theorems: boundedness and scalability . . . 253

18 The Levy inversion theorem, and the Levy continuity theorem. **255**
18.1 The Levy inversion theorem . . . 255
18.2 The Levy continuity theorem . . . 261

19 The Central Limit Theorem **267**

20 Probability mass function **272**
20.1 PMF as measure . . . 273
20.2 Monad Interpretation . . . 279
20.3 PMFs as function . . . 288
20.4 Conditional Probabilities . . . 297
20.5 Relator . . . 299

20.6 Distributions . . . 310
20.6.1 Bernoulli Distribution . . . 310
20.6.2 Geometric Distribution . . . 311
20.6.3 Uniform Multiset Distribution . . . 313
20.6.4 Uniform Distribution . . . 314
20.6.5 Poisson Distribution . . . 317
20.6.6 Binomial Distribution . . . 317
20.7 Negative Binomial distribution . . . 320
20.8 PMFs from association lists . . . 325

21 Code generation for PMFs **329**
21.1 General code generation setup . . . 330
21.2 Code abbreviations for integrals and probabilities . . . 339

22 Finite Maps **342**
22.1 Domain and Application . . . 342
22.2 Constructor of Finite Maps . . . 342
22.3 Product set of Finite Maps . . . 343
22.3.1 Basic Properties of *Pi'* . . . 343
22.4 Topological Space of Finite Maps . . . 344
22.5 Metric Space of Finite Maps . . . 347
22.6 Complete Space of Finite Maps . . . 351
22.7 Second Countable Space of Finite Maps . . . 353
22.8 Polish Space of Finite Maps . . . 355
22.9 Product Measurable Space of Finite Maps . . . 355
22.10 Isomorphism between Functions and Finite Maps . . . 367

23 Projective Limit **371**
23.1 Sequences of Finite Maps in Compact Sets . . . 371
23.2 Daniell-Kolmogorov Theorem . . . 373

24 Random Permutations **382**

25 Discrete subprobability distribution **386**
25.1 Auxiliary material . . . 386
25.1.1 More about extended reals . . . 386
25.1.2 More about *'a option* . . . 387
25.1.3 A relator for sets that treats sets like predicates . . . 390
25.1.4 Monotonicity rules . . . 390
25.1.5 Bijections . . . 390
25.2 Subprobability mass function . . . 391
25.3 Support . . . 394
25.4 Functorial structure . . . 396
25.5 Monad operations . . . 398

25.5.1 Return . . . 398
25.5.2 Bind . . . 398
25.6 Relator . . . 401
25.7 From *'a pmf* to *'a spmf* . . . 404
25.8 Weight of a subprobability . . . 405
25.9 From density to spmfs . . . 408
25.10 Ordering on spmfs . . . 409
25.11 CCPO structure for the flat ccpo *ord-option* (=) . . . 415
25.11.1 Admissibility of *rel-spmf* . . . 426
25.12 Restrictions on spmfs . . . 429
25.13 Subprobability distributions of sets . . . 431
25.14 Losslessness . . . 435
25.15 Scaling . . . 437
25.16 Conditional spmfs . . . 442
25.17 Product spmf . . . 443
25.18 Assertions . . . 447
25.19 Try . . . 448
25.20 Miscellaneous . . . 450

26 Indexed products of PMFs **451**
26.1 Preliminaries . . . 451
26.2 Definition . . . 452
26.3 Dependent product sets with a default . . . 454
26.4 Common PMF operations on products . . . 457
26.5 Merging and splitting PMF products . . . 461
26.6 Additional properties . . . 465
26.7 Applications . . . 469

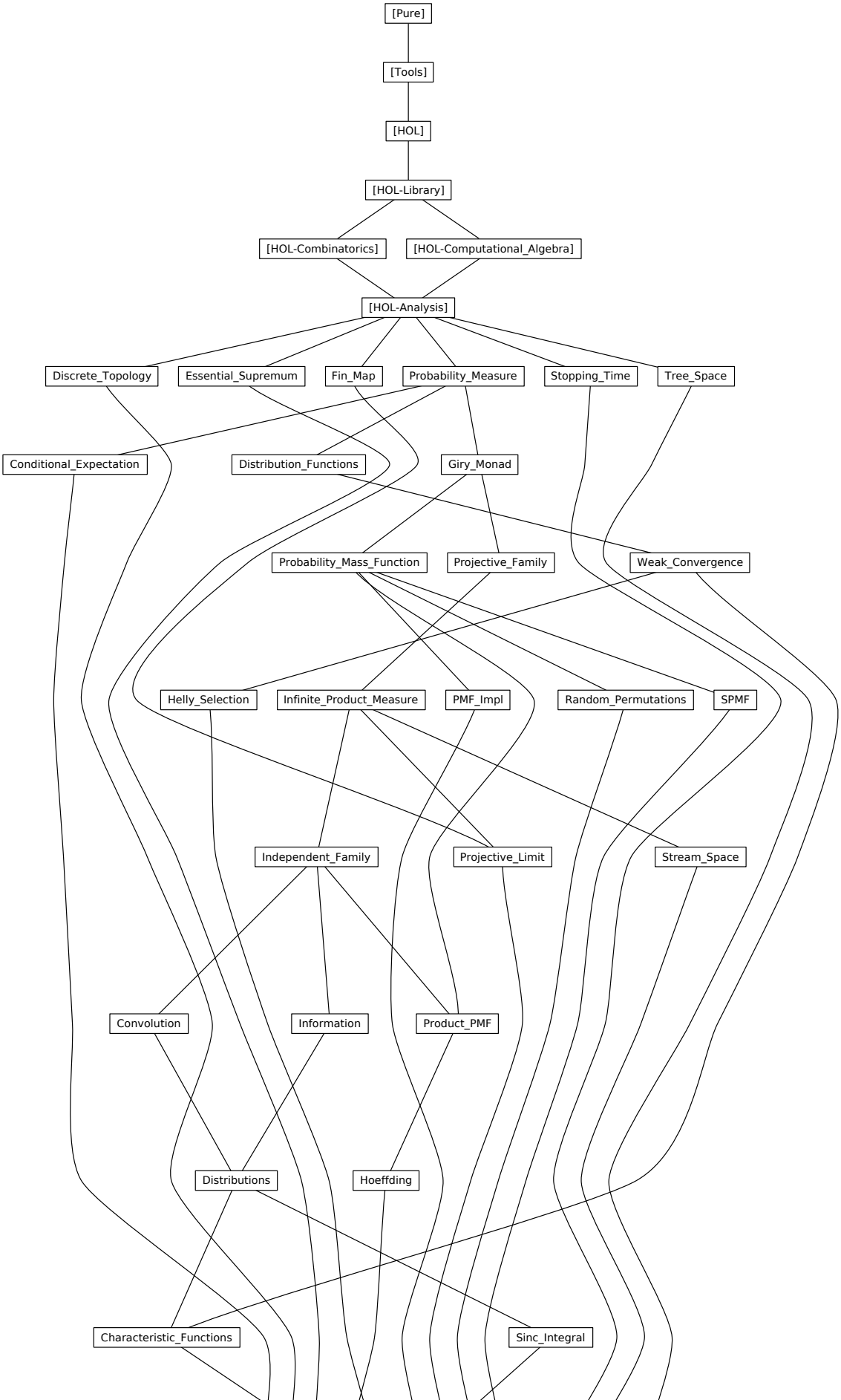
27 Hoeffding's Lemma and Hoeffding's Inequality **470**
27.1 Hoeffding's Lemma . . . 471
27.2 Hoeffding's Inequality . . . 475
27.3 Hoeffding's inequality for i.i.d. bounded random variables . . . 478
27.4 Hoeffding's Inequality for the Binomial distribution . . . 481
27.5 Tail bounds for the negative binomial distribution . . . 483

28 Conditional Expectation **514**
28.1 Restricting a measure to a sub-sigma-algebra . . . 514
28.2 Nonnegative conditional expectation . . . 518
28.3 Real conditional expectation . . . 524

29 The essential supremum **547**

30 Stopping times **550**
30.1 Stopping Time . . . 550

31 Filtration **551**
31.1 σ-algebra of a Stopping Time 551


[Pure]
[Tools]
[HOL]
[HOL-Library]
[HOL-Combinatorics]
[HOL-Computational_Algebra]
[HOL-Analysis]
Discrete_Topology
Essential_Supremum
Fin_Map
Probability_Measure
Stopping_Time
Tree_Space
Conditional_Expectation
Distribution_Functions
Giry_Monad
Probability_Mass_Function
Projective_Family
Weak_Convergence
Helly_Selection
Infinite_Product_Measure
PMF_Impl
Random_Permutations
SPMF
Independent_Family
Projective_Limit
Stream_Space
Convolution
Information
Product_PMF
Distributions
Hoeffding
Characteristic_Functions
Sinc_Integral

1 Probability measure

theory *Probability-Measure*
imports *HOL−Analysis.Analysis*
begin

locale *prob-space* = *finite-measure* +
assumes *emeasure-space-1*: *emeasure M* (*space M*) = *1*

lemma *prob-spaceI*[*Pure.intro*!]:
assumes ∗: *emeasure M* (*space M*) = *1*
shows *prob-space M*
proof −
interpret *finite-measure M*
proof
show *emeasure M* (*space M*) ≠ ∞ **using** ∗ **by** *simp*
qed
show *prob-space M* **by** *standard fact*
qed

lemma *prob-space-imp-sigma-finite*: *prob-space M* ⟹ *sigma-finite-measure M*
unfolding *prob-space-def finite-measure-def* **by** *simp*

abbreviation (**in** *prob-space*) *events* ≡ *sets M*
abbreviation (**in** *prob-space*) *prob* ≡ *measure M*
abbreviation (**in** *prob-space*) *random-variable M′ X* ≡ *X* ∈ *measurable M M′*
abbreviation (**in** *prob-space*) *expectation* ≡ *integral*^{L} *M*
abbreviation (**in** *prob-space*) *variance X* ≡ *integral*^{L} *M* (λ*x*. (*X x* − *expectation X*)^2)

lemma (**in** *prob-space*) *finite-measure* [*simp*]: *finite-measure M*
by *unfold-locales*

lemma (**in** *prob-space*) *prob-space-distr*:
assumes *f*: *f* ∈ *measurable M M′* **shows** *prob-space* (*distr M M′ f*)
proof (*rule prob-spaceI*)
have *f* −‘ *space M′* ∩ *space M* = *space M* **using** *f* **by** (*auto dest*: *measurable-space*)
with *f* **show** *emeasure* (*distr M M′ f*) (*space* (*distr M M′ f*)) = *1*
by (*auto simp*: *emeasure-distr emeasure-space-1*)
qed

lemma *prob-space-distrD*:
assumes *f*: *f* ∈ *measurable M N* **and** *M*: *prob-space* (*distr M N f*) **shows** *prob-space M*
proof
interpret *M*: *prob-space distr M N f* **by** *fact*
have *f* −‘ *space N* ∩ *space M* = *space M*
using *f*[*THEN measurable-space*] **by** *auto*

then show *emeasure M* (*space M*) = *1*
using *M.emeasure-space-1* **by** (*simp add*: *emeasure-distr*[*OF f*])
qed

lemma (**in** *prob-space*) *prob-space*: *prob* (*space M*) = *1*
using *emeasure-space-1* **unfolding** *measure-def* **by** (*simp add*: *one-ennreal.rep-eq*)

lemma (**in** *prob-space*) *prob-le-1*[*simp*, *intro*]: *prob A* ≤ *1*
using *bounded-measure*[*of A*] **by** (*simp add*: *prob-space*)

lemma (**in** *prob-space*) *not-empty*: *space M* ≠ {}
using *prob-space* **by** *auto*

lemma (**in** *prob-space*) *emeasure-eq-1-AE*:
S ∈ *sets M* ⟹ *AE x in M*. *x* ∈ *S* ⟹ *emeasure M S* = *1*
by (*subst emeasure-eq-AE*[**where** *B*=*space M*]) (*auto simp*: *emeasure-space-1*)

lemma (**in** *prob-space*) *emeasure-le-1*: *emeasure M S* ≤ *1*
unfolding *ennreal-1*[*symmetric*] *emeasure-eq-measure* **by** (*subst ennreal-le-iff*)
auto

lemma (**in** *prob-space*) *emeasure-ge-1-iff*: *emeasure M A* ≥ *1* ⟷ *emeasure M A*
= *1*
by (*rule iffI*, *intro antisym emeasure-le-1*) *simp-all*

lemma (**in** *prob-space*) *AE-iff-emeasure-eq-1*:
assumes [*measurable*]: *Measurable.pred M P*
shows (*AE x in M*. *P x*) ⟷ *emeasure M* {*x*∈*space M*. *P x*} = *1*
proof −
have ∗: {*x* ∈ *space M*. ¬ *P x*} = *space M* − {*x*∈*space M*. *P x*}
by *auto*
show *?thesis*
by (*auto simp add*: *ennreal-minus-eq-0* ∗ *emeasure-compl emeasure-space-1*
AE-iff-measurable[*OF - refl*]
intro: *antisym emeasure-le-1*)
qed

lemma (**in** *prob-space*) *measure-le-1*: *emeasure M X* ≤ *1*
using *emeasure-space*[*of M X*] **by** (*simp add*: *emeasure-space-1*)

lemma (**in** *prob-space*) *measure-ge-1-iff*: *measure M A* ≥ *1* ⟷ *measure M A* =
1
by (*auto intro*!: *antisym*)

lemma (**in** *prob-space*) *AE-I-eq-1*:
assumes *emeasure M* {*x*∈*space M*. *P x*} = *1* {*x*∈*space M*. *P x*} ∈ *sets M*
shows *AE x in M*. *P x*
proof (*rule AE-I*)
show *emeasure M* (*space M* − {*x* ∈ *space M*. *P x*}) = *0*

using *assms emeasure-space-1* **by** (*simp add*: *emeasure-compl*)
qed (*insert assms*, *auto*)

lemma *prob-space-restrict-space*:
S ∈ *sets M* ⟹ *emeasure M S* = *1* ⟹ *prob-space* (*restrict-space M S*)
by (*intro prob-spaceI*)
(*simp add*: *emeasure-restrict-space space-restrict-space*)

lemma (**in** *prob-space*) *prob-compl*:
assumes *A*: *A* ∈ *events*
shows *prob* (*space M* − *A*) = *1* − *prob A*
using *finite-measure-compl*[*OF A*] **by** (*simp add*: *prob-space*)

lemma (**in** *prob-space*) *AE-in-set-eq-1*:
assumes *A*[*measurable*]: *A* ∈ *events* **shows** (*AE x in M*. *x* ∈ *A*) ⟷ *prob A* = *1*
proof −
have ∗: {*x*∈*space M*. *x* ∈ *A*} = *A*
using *A*[*THEN sets.sets-into-space*] **by** *auto*
show *?thesis*
by (*subst AE-iff-emeasure-eq-1*) (*auto simp*: *emeasure-eq-measure* ∗)
qed

lemma (**in** *prob-space*) *AE-False*: (*AE x in M*. *False*) ⟷ *False*
proof
assume *AE x in M*. *False*
then have *AE x in M*. *x* ∈ {} **by** *simp*
then show *False*
by (*subst* (*asm*) *AE-in-set-eq-1*) *auto*
qed *simp*

lemma (**in** *prob-space*) *AE-prob-1*:
assumes *prob A* = *1* **shows** *AE x in M*. *x* ∈ *A*
proof −
from ‹*prob A* = *1*› **have** *A* ∈ *events*
by (*metis measure-notin-sets zero-neq-one*)
with *AE-in-set-eq-1 assms* **show** *?thesis* **by** *simp*
qed

lemma (**in** *prob-space*) *AE-const*[*simp*]: (*AE x in M*. *P*) ⟷ *P*
by (*cases P*) (*auto simp*: *AE-False*)

lemma (**in** *prob-space*) *ae-filter-bot*: *ae-filter M* ≠ *bot*
by (*simp add*: *trivial-limit-def*)

lemma (**in** *prob-space*) *AE-contr*:
assumes *ae*: *AE ω in M*. *P ω AE ω in M*. ¬ *P ω*
shows *False*
proof −

from *ae* **have** *AE* ω *in M. False* **by** *eventually-elim auto*
then show *False* **by** *auto*
qed

lemma (**in** *prob-space*) *integral-ge-const*:
fixes *c* :: *real*
shows *integrable M f* $\Longrightarrow$ (*AE x in M. c* $\leq$ *f x*) $\Longrightarrow$ *c* $\leq$ ($\int$ *x. f x* ∂*M*)
using *integral-mono-AE*[*of M* λ*x. c f*] *prob-space* **by** *simp*

lemma (**in** *prob-space*) *integral-le-const*:
fixes *c* :: *real*
shows *integrable M f* $\Longrightarrow$ (*AE x in M. f x* $\leq$ *c*) $\Longrightarrow$ ($\int$ *x. f x* ∂*M*) $\leq$ *c*
using *integral-mono-AE*[*of M f* λ*x. c*] *prob-space* **by** *simp*

lemma (**in** *prob-space*) *nn-integral-ge-const*:
(*AE x in M. c* $\leq$ *f x*) $\Longrightarrow$ *c* $\leq$ ($\int^+$*x. f x* ∂*M*)
using *nn-integral-mono-AE*[*of* λ*x. c f M*] *emeasure-space-1*
by (*simp split*: *if-split-asm*)

lemma (**in** *prob-space*) *expectation-less*:
fixes *X* :: - $\Rightarrow$ *real*
assumes [*simp*]: *integrable M X*
assumes *gt*: *AE x in M. X x* $<$ *b*
shows *expectation X* $<$ *b*
proof −
have *expectation X* $<$ *expectation* (λ*x. b*)
using *gt emeasure-space-1*
by (*intro integral-less-AE-space*) *auto*
then show *?thesis* **using** *prob-space* **by** *simp*
qed

lemma (**in** *prob-space*) *expectation-greater*:
fixes *X* :: - $\Rightarrow$ *real*
assumes [*simp*]: *integrable M X*
assumes *gt*: *AE x in M. a* $<$ *X x*
shows *a* $<$ *expectation X*
proof −
have *expectation* (λ*x. a*) $<$ *expectation X*
using *gt emeasure-space-1*
by (*intro integral-less-AE-space*) *auto*
then show *?thesis* **using** *prob-space* **by** *simp*
qed

lemma (**in** *prob-space*) *jensens-inequality*:
fixes *q* :: *real* $\Rightarrow$ *real*
assumes *X*: *integrable M X AE x in M. X x* $\in$ *I*
assumes *I*: *I* = {*a* <..< *b*} $\vee$ *I* = {*a* <..} $\vee$ *I* = {..< *b*} $\vee$ *I* = *UNIV*
assumes *q*: *integrable M* (λ*x. q* (*X x*)) *convex-on I q*
shows *q* (*expectation X*) $\leq$ *expectation* (λ*x. q* (*X x*))

proof −
let *?F* = λ*x*. *Inf* ((λ*t*. (*q x* − *q t*) / (*x* − *t*)) ‘ ({*x*<..} ∩ *I*))
from *X*(*2*) *AE-False* **have** *I* ≠ {} **by** *auto*

from *I* **have** *open I* **by** *auto*

note *I*
moreover
{ assume *I* ⊆ {*a* <..}
with *X* **have** *a* < *expectation X*
by (*intro expectation-greater*) *auto* **}**
moreover
{ assume *I* ⊆ {..< *b*}
with *X* **have** *expectation X* < *b*
by (*intro expectation-less*) *auto* **}**
ultimately have *expectation X* ∈ *I*
by (*elim disjE*) (*auto simp*: *subset-eq*)
moreover
{ fix *y* **assume** *y*: *y* ∈ *I*
with *q*(*2*) ‹*open I*› **have** *Sup* ((λ*x*. *q x* + *?F x* ∗ (*y* − *x*)) ‘ *I*) = *q y*
by (*auto intro*!: *cSup-eq-maximum convex-le-Inf-differential image-eqI* [*OF - y*] *simp*: *interior-open*) **}**
ultimately have *q* (*expectation X*) = *Sup* ((λ*x*. *q x* + *?F x* ∗ (*expectation X* − *x*)) ‘ *I*)
by *simp*
also have ... ≤ *expectation* (λ*w*. *q* (*X w*))
proof (*rule cSup-least*)
show (λ*x*. *q x* + *?F x* ∗ (*expectation X* − *x*)) ‘ *I* ≠ {}
using ‹*I* ≠ {}› **by** *auto*
next
fix *k* **assume** *k* ∈ (λ*x*. *q x* + *?F x* ∗ (*expectation X* − *x*)) ‘ *I*
then obtain *x*
where *x*: *k* = *q x* + (*INF t*∈{*x*<..} ∩ *I*. (*q x* − *q t*) / (*x* − *t*)) ∗ (*expectation X* − *x*) *x* ∈ *I* **..**
have *q x* + *?F x* ∗ (*expectation X* − *x*) = *expectation* (λ*w*. *q x* + *?F x* ∗ (*X w* − *x*))
using *prob-space* **by** (*simp add*: *X*)
also have ... ≤ *expectation* (λ*w*. *q* (*X w*))
using ‹*x* ∈ *I*› ‹*open I*› *X*(*2*)
apply (*intro integral-mono-AE Bochner-Integration.integrable-add Bochner-Integration.integrable-mult-righ Bochner-Integration.integrable-diff integrable-const X q*)
apply (*elim eventually-mono*)
apply (*intro convex-le-Inf-differential*)
apply (*auto simp*: *interior-open q*)
done
finally show *k* ≤ *expectation* (λ*w*. *q* (*X w*)) **using** *x* **by** *auto*
qed
finally show *q* (*expectation X*) ≤ *expectation* (λ*x*. *q* (*X x*)) **.**

qed

1.1 Introduce binder for probability

syntax
-prob :: *pttrn* ⇒ *logic* ⇒ *logic* ⇒ *logic* (‹('𝒫'((/- *in* -./ -)'))›)

translations
$\mathcal{P}(x \text{ in } M.\ P)$ => *CONST measure M* {$x \in$ *CONST space M*. *P*}

print-translation ‹
let
fun to-pattern (*Const* (***const-syntax*** ‹*Pair*›, -) $ *l* $ *r*) =
Syntax.const ***const-syntax*** ‹*Pair*› :: *to-pattern l* @ *to-pattern r*
| *to-pattern* (*t as* (*Const* (***syntax-const*** ‹*-bound*›, -)) $ -) = [*t*]

fun mk-pattern ((*t*, *n*) :: *xs*) = *mk-patterns n xs* |>> *curry list-comb t*
and mk-patterns 0 xs = ([], *xs*)
| *mk-patterns n xs* =
let
val (*t*, *xs'*) = *mk-pattern xs*
val (*ts*, *xs''*) = *mk-patterns* (*n* − *1*) *xs'*
in
(*t* :: *ts*, *xs''*)
end

fun unnest-tuples
(*Const* (***syntax-const*** ‹*-pattern*›, -) $
t1 $
(*t as* (*Const* (***syntax-const*** ‹*-pattern*›, -) $ - $ -)))
= *let*
val (- $ *t2* $ *t3*) = *unnest-tuples t*
in
Syntax.const ***syntax-const*** ‹*-pattern*› $
unnest-tuples t1 $
(*Syntax.const* ***syntax-const*** ‹*-patterns*› $ *t2* $ *t3*)
end
| *unnest-tuples pat* = *pat*

fun tr' [*sig-alg*, *Const* (***const-syntax*** ‹*Collect*›, -) $ *t*] =
let
val bound-dummyT = *Const* (***syntax-const*** ‹*-bound*›, *dummyT*)

fun go pattern elem
(*Const* (***const-syntax*** ‹*conj*›, -) $
(*Const* (***const-syntax*** ‹*Set.member*›, -) $ *elem'* $ (*Const* (***const-syntax*** ‹*space*›, -) $ *sig-alg'*)) $
u)
= *let*

```
            val - = if sig-alg aconv sig-alg' andalso to-pattern elem' = rev elem then
() else raise Match;
            val (pat, rest) = mk-pattern (rev pattern);
            val - = case rest of [] => () | - => raise Match
          in
            Syntax.const syntax-const ‹-prob› $ unnest-tuples pat $ sig-alg $ u
          end
      | go pattern elem (Abs abs) =
          let
            val (x as (- $ tx), t) = Syntax-Trans.atomic-abs-tr' abs
          in
            go ((x, 0) :: pattern) (bound-dummyT $ tx :: elem) t
          end
      | go pattern elem (Const (const-syntax ‹case-prod›, -) $ t) =
          go
            ((Syntax.const syntax-const ‹-pattern›, 2) :: pattern)
            (Syntax.const const-syntax ‹Pair› :: elem)
            t
    in
      go [] [] t
    end
  in
    [(const-syntax ‹Sigma-Algebra.measure›, K tr')]
  end
›

definition
  cond-prob M P Q = 𝒫(ω in M. P ω ∧ Q ω) / 𝒫(ω in M. Q ω)

syntax
  -conditional-prob :: pttrn ⇒ logic ⇒ logic ⇒ logic ⇒ logic (‹('𝒫'(- in -. - |/ -'))›)

translations
  𝒫(x in M. P | Q) => CONST cond-prob M (λx. P) (λx. Q)

lemma (in prob-space) AE-E-prob:
  assumes ae: AE x in M. P x
  obtains S where S ⊆ {x ∈ space M. P x} S ∈ events prob S = 1
proof −
  from ae[THEN AE-E] obtain N
    where {x ∈ space M. ¬ P x} ⊆ N emeasure M N = 0 N ∈ events by auto
  then show thesis
    by (intro that[of space M − N])
       (auto simp: prob-compl prob-space emeasure-eq-measure measure-nonneg)
qed

lemma (in prob-space) prob-neg: {x∈space M. P x} ∈ events ⟹ 𝒫(x in M. ¬ P
x) = 1 − 𝒫(x in M. P x)
  by (auto intro!: arg-cong[where f=prob] simp add: prob-compl[symmetric])
```

lemma (**in** *prob-space*) *prob-eq-AE*:
(*AE x in M. P x* $\longleftrightarrow$ *Q x*) $\Longrightarrow$ {*x*∈*space M. P x*} ∈ *events* $\Longrightarrow$ {*x*∈*space M. Q x*} ∈ *events* $\Longrightarrow$ $\mathcal{P}$(*x in M. P x*) = $\mathcal{P}$(*x in M. Q x*)
by (*rule finite-measure-eq-AE*) *auto*

lemma (**in** *prob-space*) *prob-eq-0-AE*:
assumes *not*: *AE x in M.* ¬ *P x* **shows** $\mathcal{P}$(*x in M. P x*) = *0*
proof *cases*
assume {*x*∈*space M. P x*} ∈ *events*
with *not* **have** $\mathcal{P}$(*x in M. P x*) = $\mathcal{P}$(*x in M. False*)
by (*intro prob-eq-AE*) *auto*
then show *?thesis* **by** *simp*
qed (*simp add*: *measure-notin-sets*)

lemma (**in** *prob-space*) *prob-Collect-eq-0*:
{*x* ∈ *space M. P x*} ∈ *sets M* $\Longrightarrow$ $\mathcal{P}$(*x in M. P x*) = *0* $\longleftrightarrow$ (*AE x in M.* ¬ *P x*)
using *AE-iff-measurable*[*OF - refl, of M* λ*x.* ¬ *P x*] **by** (*simp add*: *emeasure-eq-measure measure-nonneg*)

lemma (**in** *prob-space*) *prob-Collect-eq-1*:
{*x* ∈ *space M. P x*} ∈ *sets M* $\Longrightarrow$ $\mathcal{P}$(*x in M. P x*) = *1* $\longleftrightarrow$ (*AE x in M. P x*)
using *AE-in-set-eq-1*[*of* {*x*∈*space M. P x*}] **by** *simp*

lemma (**in** *prob-space*) *prob-eq-0*:
A ∈ *sets M* $\Longrightarrow$ *prob A* = *0* $\longleftrightarrow$ (*AE x in M. x* ∉ *A*)
using *AE-iff-measurable*[*OF - refl, of M* λ*x. x* ∉ *A*]
by (*auto simp add*: *emeasure-eq-measure Int-def*[*symmetric*] *measure-nonneg*)

lemma (**in** *prob-space*) *prob-eq-1*:
A ∈ *sets M* $\Longrightarrow$ *prob A* = *1* $\longleftrightarrow$ (*AE x in M. x* ∈ *A*)
using *AE-in-set-eq-1*[*of A*] **by** *simp*

lemma (**in** *prob-space*) *prob-sums*:
assumes *P*: $\bigwedge$*n.* {*x*∈*space M. P n x*} ∈ *events*
assumes *Q*: {*x*∈*space M. Q x*} ∈ *events*
assumes *ae*: *AE x in M.* (∀ *n. P n x* $\longrightarrow$ *Q x*) ∧ (*Q x* $\longrightarrow$ (∃!*n. P n x*))
shows (λ*n.* $\mathcal{P}$(*x in M. P n x*)) *sums* $\mathcal{P}$(*x in M. Q x*)
proof −
from *ae*[*THEN AE-E-prob*] **obtain** *S*
where *S*:
S ⊆ {*x* ∈ *space M.* (∀ *n. P n x* $\longrightarrow$ *Q x*) ∧ (*Q x* $\longrightarrow$ (∃!*n. P n x*))}
S ∈ *events*
prob S = *1*
by *auto*
then have *disj*: *disjoint-family* (λ*n.* {*x*∈*space M. P n x*} ∩ *S*)
by (*auto simp*: *disjoint-family-on-def*)
from *S* **have** *ae-S*:
AE x in M. x ∈ {*x*∈*space M. Q x*} $\longleftrightarrow$ *x* ∈ ($\bigcup$*n.* {*x*∈*space M. P n x*} ∩ *S*)

$\bigwedge n.\ AE\ x\ in\ M.\ x \in \{x \in space\ M.\ P\ n\ x\} \longleftrightarrow x \in \{x \in space\ M.\ P\ n\ x\} \cap S$
using *ae* **by** (*auto dest*!: *AE-prob-1*)
from *ae-S* **have** $*$:
$\mathcal{P}(x\ in\ M.\ Q\ x) = prob\ (\bigcup n.\ \{x \in space\ M.\ P\ n\ x\} \cap S)$
using *P Q S* **by** (*intro finite-measure-eq-AE*) *auto*
from *ae-S* **have** $**$:
$\bigwedge n.\ \mathcal{P}(x\ in\ M.\ P\ n\ x) = prob\ (\{x \in space\ M.\ P\ n\ x\} \cap S)$
using *P Q S* **by** (*intro finite-measure-eq-AE*) *auto*
show *?thesis*
unfolding $*$ $**$ **using** *S P disj*
by (*intro finite-measure-UNION*) *auto*
qed

lemma (**in** *prob-space*) *prob-sum*:
assumes [*simp*, *intro*]: *finite I*
assumes *P*: $\bigwedge n.\ n \in I \Longrightarrow \{x \in space\ M.\ P\ n\ x\} \in events$
assumes *Q*: $\{x \in space\ M.\ Q\ x\} \in events$
assumes *ae*: $AE\ x\ in\ M.\ (\forall n \in I.\ P\ n\ x \longrightarrow Q\ x) \wedge (Q\ x \longrightarrow (\exists! n \in I.\ P\ n\ x))$
shows $\mathcal{P}(x\ in\ M.\ Q\ x) = (\sum n \in I.\ \mathcal{P}(x\ in\ M.\ P\ n\ x))$
proof −
from *ae*[*THEN AE-E-prob*] **obtain** *S*
where *S*:
$S \subseteq \{x \in space\ M.\ (\forall n \in I.\ P\ n\ x \longrightarrow Q\ x) \wedge (Q\ x \longrightarrow (\exists! n.\ n \in I \wedge P\ n\ x))\}$
$S \in events$
$prob\ S = 1$
by *auto*
then have *disj*: *disjoint-family-on* $(\lambda n.\ \{x \in space\ M.\ P\ n\ x\} \cap S)\ I$
by (*auto simp*: *disjoint-family-on-def*)
from *S* **have** *ae-S*:
$AE\ x\ in\ M.\ x \in \{x \in space\ M.\ Q\ x\} \longleftrightarrow x \in (\bigcup n \in I.\ \{x \in space\ M.\ P\ n\ x\} \cap S)$
$\bigwedge n.\ n \in I \Longrightarrow AE\ x\ in\ M.\ x \in \{x \in space\ M.\ P\ n\ x\} \longleftrightarrow x \in \{x \in space\ M.\ P\ n\ x\} \cap S$
using *ae* **by** (*auto dest*!: *AE-prob-1*)
from *ae-S* **have** $*$:
$\mathcal{P}(x\ in\ M.\ Q\ x) = prob\ (\bigcup n \in I.\ \{x \in space\ M.\ P\ n\ x\} \cap S)$
using *P Q S* **by** (*intro finite-measure-eq-AE*) (*auto intro*!: *sets.Int*)
from *ae-S* **have** $**$:
$\bigwedge n.\ n \in I \Longrightarrow \mathcal{P}(x\ in\ M.\ P\ n\ x) = prob\ (\{x \in space\ M.\ P\ n\ x\} \cap S)$
using *P Q S* **by** (*intro finite-measure-eq-AE*) *auto*
show *?thesis*
using *S P disj*
by (*auto simp add*: $*$ $**$ *simp del*: *UN-simps intro*!: *finite-measure-finite-Union*)
qed

lemma (**in** *prob-space*) *prob-EX-countable*:
assumes *sets*: $\bigwedge i.\ i \in I \Longrightarrow \{x \in space\ M.\ P\ i\ x\} \in sets\ M$ **and** *I*: *countable I*
assumes *disj*: $AE\ x\ in\ M.\ \forall i \in I.\ \forall j \in I.\ P\ i\ x \longrightarrow P\ j\ x \longrightarrow i = j$
shows $\mathcal{P}(x\ in\ M.\ \exists i \in I.\ P\ i\ x) = (\int^+ i.\ \mathcal{P}(x\ in\ M.\ P\ i\ x)\ \partial count\text{-}space\ I)$

proof −
let *?N*= λ*x*. ∃!*i*∈*I*. *P i x*
have *ennreal* (P(*x in M*. ∃ *i*∈*I*. *P i x*)) = P(*x in M*. (∃ *i*∈*I*. *P i x* ∧ *?N x*))
unfolding *ennreal-inj*[*OF measure-nonneg measure-nonneg*]
proof (*rule prob-eq-AE*)
show *AE x in M*. (∃ *i*∈*I*. *P i x*) = (∃ *i*∈*I*. *P i x* ∧ *?N x*)
using *disj* **by** *eventually-elim blast*
qed (*auto intro*!: *sets.sets-Collect-countable-Ex′ sets.sets-Collect-conj sets.sets-Collect-countable-Ex1′ I sets*)+
also have P(*x in M*. (∃ *i*∈*I*. *P i x* ∧ *?N x*)) = *emeasure M* (⋃ *i*∈*I*. {*x*∈*space M*. *P i x* ∧ *?N x*})
unfolding *emeasure-eq-measure* **by** (*auto intro*!: *arg-cong*[**where** *f*=*prob*] *simp*: *measure-nonneg*)
also have ... = (∫ +*i*. *emeasure M* {*x*∈*space M*. *P i x* ∧ *?N x*} ∂*count-space I*)
by (*rule emeasure-UN-countable*)
(*auto intro*!: *sets.sets-Collect-countable-Ex′ sets.sets-Collect-conj sets.sets-Collect-countable-Ex1′ I sets*
simp: *disjoint-family-on-def*)
also have ... = (∫ +*i*. P(*x in M*. *P i x*) ∂*count-space I*)
unfolding *emeasure-eq-measure* **using** *disj*
by (*intro nn-integral-cong ennreal-inj*[*THEN iffD2*] *prob-eq-AE*)
(*auto intro*!: *sets.sets-Collect-countable-Ex′ sets.sets-Collect-conj sets.sets-Collect-countable-Ex1′ I sets measure-nonneg*)+
finally show *?thesis* **.**
qed

lemma (**in** *prob-space*) *cond-prob-eq-AE*:
assumes *P*: *AE x in M*. *Q x* ⟶ *P x* ⟷ *P′ x* {*x*∈*space M*. *P x*} ∈ *events* {*x*∈*space M*. *P′ x*} ∈ *events*
assumes *Q*: *AE x in M*. *Q x* ⟷ *Q′ x* {*x*∈*space M*. *Q x*} ∈ *events* {*x*∈*space M*. *Q′ x*} ∈ *events*
shows *cond-prob M P Q* = *cond-prob M P′ Q′*
using *P Q*
by (*auto simp*: *cond-prob-def intro*!: *arg-cong2*[**where** *f*=(/)] *prob-eq-AE sets.sets-Collect-conj*)

lemma (**in** *prob-space*) *joint-distribution-Times-le-fst*:
random-variable MX X ⟹ *random-variable MY Y* ⟹ *A* ∈ *sets MX* ⟹ *B* ∈ *sets MY*
⟹ *emeasure* (*distr M* (*MX* ⨂$_M$ *MY*) (λ*x*. (*X x*, *Y x*))) (*A* × *B*) ≤ *emeasure* (*distr M MX X*) *A*
by (*auto simp*: *emeasure-distr measurable-pair-iff comp-def intro*!: *emeasure-mono measurable-sets*)

lemma (**in** *prob-space*) *joint-distribution-Times-le-snd*:
random-variable MX X ⟹ *random-variable MY Y* ⟹ *A* ∈ *sets MX* ⟹ *B* ∈ *sets MY*
⟹ *emeasure* (*distr M* (*MX* ⨂$_M$ *MY*) (λ*x*. (*X x*, *Y x*))) (*A* × *B*) ≤ *emeasure* (*distr M MY Y*) *B*

by (*auto simp*: *emeasure-distr measurable-pair-iff comp-def intro*!: *emeasure-mono measurable-sets*)

lemma (**in** *prob-space*) *variance-eq*:
fixes $X :: 'a \Rightarrow real$
assumes [*simp*]: *integrable M X*
assumes [*simp*]: *integrable M* $(\lambda x.\ (X\ x)^2)$
shows *variance X* = *expectation* $(\lambda x.\ (X\ x)^2)$ − (*expectation X*)^2
by (*simp add*: *field-simps prob-space power2-diff power2-eq-square*[*symmetric*])

lemma (**in** *prob-space*) *variance-positive*: *0* ≤ *variance* ($X::'a \Rightarrow real$)
by (*intro integral-nonneg-AE*) (*auto intro*!: *integral-nonneg-AE*)

lemma (**in** *prob-space*) *variance-mean-zero*:
expectation X = *0* ⟹ *variance X* = *expectation* $(\lambda x.\ (X\ x)\hat{}2)$
by *simp*

theorem (**in** *prob-space*) *Chebyshev-inequality*:
assumes [*measurable*]: *random-variable borel f*
assumes *integrable M* $(\lambda x.\ f\ x\ \hat{}\ 2)$
defines $\mu \equiv$ *expectation f*
assumes *a* > *0*
shows *prob* $\{x \in$ *space M*. $|f\ x - \mu| \geq a\} \leq$ *variance f* / a^2
unfolding μ-*def*
proof (*rule second-moment-method*)
have *integrable*: *integrable M f*
using *assms* **by** (*blast dest*: *square-integrable-imp-integrable*)
show *integrable M* $(\lambda x.\ (f\ x - expectation\ f)^2)$
using *assms integrable* **unfolding** *power2-eq-square ring-distribs*
by (*intro Bochner-Integration.integrable-diff*) *auto*
qed (*use assms* **in** *auto*)

locale *pair-prob-space* = *pair-sigma-finite M1 M2* + *M1*: *prob-space M1* + *M2*: *prob-space M2* **for** *M1 M2*

sublocale *pair-prob-space* ⊆ *P*?: *prob-space M1* $\bigotimes_M$ *M2*
proof
show *emeasure* (*M1* $\bigotimes_M$ *M2*) (*space* (*M1* $\bigotimes_M$ *M2*)) = *1*
by (*simp add*: *M2.emeasure-pair-measure-Times M1.emeasure-space-1 M2.emeasure-space-1 space-pair-measure*)
qed

locale *product-prob-space* = *product-sigma-finite M* **for** *M* :: $'i \Rightarrow 'a$ *measure* +
fixes *I* :: *'i set*
assumes *prob-space*: $\bigwedge i.$ *prob-space* (*M i*)

sublocale *product-prob-space* ⊆ *M*?: *prob-space M i* **for** *i*
by (*rule prob-space*)

locale *finite-product-prob-space* = *finite-product-sigma-finite M I* + *product-prob-space M I* **for** *M I*

sublocale *finite-product-prob-space* ⊆ *prob-space* Π_M *i∈I. M i*
proof
show *emeasure* (Π_M *i∈I. M i*) (*space* (Π_M *i∈I. M i*)) = *1*
by (*simp add*: *measure-times M.emeasure-space-1 prod.neutral-const space-PiM*)
qed

lemma (**in** *finite-product-prob-space*) *prob-times*:
assumes *X*: $\bigwedge i.\ i \in I \Longrightarrow X\ i \in sets\ (M\ i)$
shows *prob* (Π_E *i∈I. X i*) = ($\prod$ *i∈I. M.prob i* (*X i*))
proof −
have *ennreal* (*measure* (Π_M *i∈I. M i*) (Π_E *i∈I. X i*)) = *emeasure* (Π_M *i∈I. M i*) (Π_E *i∈I. X i*)
using *X* **by** (*simp add*: *emeasure-eq-measure*)
also have ... = ($\prod$ *i∈I. emeasure* (*M i*) (*X i*))
using *measure-times X* **by** *simp*
also have ... = *ennreal* ($\prod$ *i∈I. measure* (*M i*) (*X i*))
using *X* **by** (*simp add*: *M.emeasure-eq-measure prod-ennreal measure-nonneg*)
finally show *?thesis* **by** (*simp add*: *measure-nonneg prod-nonneg*)
qed

lemma *product-prob-spaceI*:
assumes $\bigwedge i.$ *prob-space* (*M i*)
shows *product-prob-space M*
unfolding *product-prob-space-def product-prob-space-axioms-def product-sigma-finite-def*
proof *safe*
fix *i*
interpret *prob-space M i*
by (*rule assms*)
show *sigma-finite-measure* (*M i*) *prob-space* (*M i*)
by *unfold-locales*
qed

1.2 Distributions

definition *distributed* :: *'a measure* ⇒ *'b measure* ⇒ (*'a* ⇒ *'b*) ⇒ (*'b* ⇒ *ennreal*) ⇒ *bool*
where
distributed M N X f ⟷
distr M N X = *density N f* ∧ *f* ∈ *borel-measurable N* ∧ *X* ∈ *measurable M N*

lemma
assumes *distributed M N X f*
shows *distributed-distr-eq-density*: *distr M N X* = *density N f*
and *distributed-measurable*: *X* ∈ *measurable M N*
and *distributed-borel-measurable*: *f* ∈ *borel-measurable N*
using *assms* **by** (*simp-all add*: *distributed-def*)

lemma
assumes *D*: *distributed M N X f*
shows *distributed-measurable'*[*measurable-dest*]:
g ∈ *measurable L M* ⟹ (λ*x*. *X* (*g x*)) ∈ *measurable L N*
and *distributed-borel-measurable'*[*measurable-dest*]:
h ∈ *measurable L N* ⟹ (λ*x*. *f* (*h x*)) ∈ *borel-measurable L*
using *distributed-measurable*[*OF D*] *distributed-borel-measurable*[*OF D*]
by *simp-all*

lemma *distributed-real-measurable*:
(⋀*x*. *x* ∈ *space N* ⟹ *0* ≤ *f x*) ⟹ *distributed M N X* (λ*x*. *ennreal* (*f x*)) ⟹ *f* ∈ *borel-measurable N*
by (*simp-all add*: *distributed-def*)

lemma *distributed-real-measurable'*:
(⋀*x*. *x* ∈ *space N* ⟹ *0* ≤ *f x*) ⟹ *distributed M N X* (λ*x*. *ennreal* (*f x*)) ⟹
h ∈ *measurable L N* ⟹ (λ*x*. *f* (*h x*)) ∈ *borel-measurable L*
using *distributed-real-measurable*[*measurable*] **by** *simp*

lemma *joint-distributed-measurable1*:
distributed M (*S* ⨂$_M$ *T*) (λ*x*. (*X x*, *Y x*)) *f* ⟹ *h1* ∈ *measurable N M* ⟹ (λ*x*. *X* (*h1 x*)) ∈ *measurable N S*
by *simp*

lemma *joint-distributed-measurable2*:
distributed M (*S* ⨂$_M$ *T*) (λ*x*. (*X x*, *Y x*)) *f* ⟹ *h2* ∈ *measurable N M* ⟹ (λ*x*. *Y* (*h2 x*)) ∈ *measurable N T*
by *simp*

lemma *distributed-count-space*:
assumes *X*: *distributed M* (*count-space A*) *X P* **and** *a*: *a* ∈ *A* **and** *A*: *finite A*
shows *P a* = *emeasure M* (*X* −‘ {*a*} ∩ *space M*)
proof −
have *emeasure M* (*X* −‘ {*a*} ∩ *space M*) = *emeasure* (*distr M* (*count-space A*) *X*) {*a*}
using *X a A* **by** (*simp add*: *emeasure-distr*)
also have ... = *emeasure* (*density* (*count-space A*) *P*) {*a*}
using *X* **by** (*simp add*: *distributed-distr-eq-density*)
also have ... = (∫$^+$*x*. *P a* ∗ *indicator* {*a*} *x* ∂*count-space A*)
using *X a* **by** (*auto simp add*: *emeasure-density distributed-def indicator-def intro*!: *nn-integral-cong*)
also have ... = *P a*
using *X a* **by** (*subst nn-integral-cmult-indicator*) (*auto simp*: *distributed-def one-ennreal-def*[*symmetric*] *AE-count-space*)
finally show *?thesis* **..**
qed

lemma *distributed-cong-density*:

$(AE\ x\ in\ N.\ f\ x = g\ x) \Longrightarrow g \in borel\text{-}measurable\ N \Longrightarrow f \in borel\text{-}measurable\ N$
$\Longrightarrow$
$distributed\ M\ N\ X\ f \longleftrightarrow distributed\ M\ N\ X\ g$
by (*auto simp*: *distributed-def intro*!: *density-cong*)

lemma (**in** *prob-space*) *distributed-imp-emeasure-nonzero*:
assumes *X*: *distributed M MX X Px*
shows $emeasure\ MX\ \{x \in space\ MX.\ Px\ x \neq 0\} \neq 0$
proof
note $Px = distributed\text{-}borel\text{-}measurable[OF\ X]$
interpret *X*: *prob-space distr M MX X*
using $distributed\text{-}measurable[OF\ X]$ **by** (*rule prob-space-distr*)

assume $emeasure\ MX\ \{x \in space\ MX.\ Px\ x \neq 0\} = 0$
with *Px* **have** $AE\ x\ in\ MX.\ Px\ x = 0$
by (*intro AE-I*[*OF subset-refl*]) (*auto simp*: *borel-measurable-ennreal-iff*)
moreover
from *X.emeasure-space-1* **have** $(\int^+ x.\ Px\ x\ \partial MX) = 1$
unfolding $distributed\text{-}distr\text{-}eq\text{-}density[OF\ X]$ **using** *Px*
by (*subst* (*asm*) *emeasure-density*)
(*auto simp*: *borel-measurable-ennreal-iff intro*!: *integral-cong cong*: *nn-integral-cong*)
ultimately show *False*
by (*simp add*: *nn-integral-cong-AE*)
qed

lemma *subdensity*:
assumes *T*: $T \in measurable\ P\ Q$
assumes *f*: *distributed M P X f*
assumes *g*: *distributed M Q Y g*
assumes *Y*: $Y = T \circ X$
shows $AE\ x\ in\ P.\ g\ (T\ x) = 0 \longrightarrow f\ x = 0$
proof −
have $\{x \in space\ Q.\ g\ x = 0\} \in null\text{-}sets\ (distr\ M\ Q\ (T \circ X))$
using *g Y* **by** (*auto simp*: *null-sets-density-iff distributed-def*)
also have $distr\ M\ Q\ (T \circ X) = distr\ (distr\ M\ P\ X)\ Q\ T$
using $T\ f[THEN\ distributed\text{-}measurable]$ **by** (*rule distr-distr*[*symmetric*])
finally have $T -` \{x \in space\ Q.\ g\ x = 0\} \cap space\ P \in null\text{-}sets\ (distr\ M\ P\ X)$
using *T* **by** (*subst* (*asm*) *null-sets-distr-iff*) *auto*
also have $T -` \{x \in space\ Q.\ g\ x = 0\} \cap space\ P = \{x \in space\ P.\ g\ (T\ x) = 0\}$
using *T* **by** (*auto dest*: *measurable-space*)
finally show *?thesis*
using *f g* **by** (*auto simp add*: *null-sets-density-iff distributed-def*)
qed

lemma *subdensity-real*:
fixes $g :: 'a \Rightarrow real$ **and** $f :: 'b \Rightarrow real$
assumes *T*: $T \in measurable\ P\ Q$
assumes *f*: *distributed M P X f*
assumes *g*: *distributed M Q Y g*

assumes Y: $Y = T \circ X$
shows $(AE\ x\ in\ P.\ 0 \leq g\ (T\ x)) \Longrightarrow (AE\ x\ in\ P.\ 0 \leq f\ x) \Longrightarrow AE\ x\ in\ P.\ g\ (T\ x) = 0 \longrightarrow f\ x = 0$
using *subdensity*[*OF T*, *of M X* $\lambda x.\ ennreal\ (f\ x)$ *Y* $\lambda x.\ ennreal\ (g\ x)$] *assms*
by *auto*

lemma *distributed-emeasure*:
$distributed\ M\ N\ X\ f \Longrightarrow A \in sets\ N \Longrightarrow emeasure\ M\ (X\ -`\ A \cap space\ M) = (\int^+ x.\ f\ x * indicator\ A\ x\ \partial N)$
by (*auto simp*: *distributed-distr-eq-density*[*symmetric*] *emeasure-density*[*symmetric*] *emeasure-distr*)

lemma *distributed-nn-integral*:
$distributed\ M\ N\ X\ f \Longrightarrow g \in borel\text{-}measurable\ N \Longrightarrow (\int^+ x.\ f\ x * g\ x\ \partial N) = (\int^+ x.\ g\ (X\ x)\ \partial M)$
by (*auto simp*: *distributed-distr-eq-density*[*symmetric*] *nn-integral-density*[*symmetric*] *nn-integral-distr*)

lemma *distributed-integral*:
$distributed\ M\ N\ X\ f \Longrightarrow g \in borel\text{-}measurable\ N \Longrightarrow (\bigwedge x.\ x \in space\ N \Longrightarrow 0 \leq f\ x) \Longrightarrow$
$(\int x.\ f\ x * g\ x\ \partial N) = (\int x.\ g\ (X\ x)\ \partial M)$
supply *distributed-real-measurable*[*measurable*]
by (*auto simp*: *distributed-distr-eq-density*[*symmetric*] *integral-real-density*[*symmetric*] *integral-distr*)

lemma *distributed-transform-integral*:
assumes *Px*: *distributed M N X Px* $\bigwedge x.\ x \in space\ N \Longrightarrow 0 \leq Px\ x$
assumes *distributed M P Y Py* $\bigwedge x.\ x \in space\ P \Longrightarrow 0 \leq Py\ x$
assumes Y: $Y = T \circ X$ **and** T: $T \in measurable\ N\ P$ **and** f: $f \in borel\text{-}measurable\ P$
shows $(\int x.\ Py\ x * f\ x\ \partial P) = (\int x.\ Px\ x * f\ (T\ x)\ \partial N)$
proof −
have $(\int x.\ Py\ x * f\ x\ \partial P) = (\int x.\ f\ (Y\ x)\ \partial M)$
by (*rule distributed-integral*) *fact*+
also have $\ldots = (\int x.\ f\ (T\ (X\ x))\ \partial M)$
using Y **by** *simp*
also have $\ldots = (\int x.\ Px\ x * f\ (T\ x)\ \partial N)$
using *measurable-comp*[*OF T f*] *Px* **by** (*intro distributed-integral*[*symmetric*]) (*auto simp*: *comp-def*)
finally show *?thesis* **.**
qed

lemma (**in** *prob-space*) *distributed-unique*:
assumes *Px*: *distributed M S X Px*
assumes *Py*: *distributed M S X Py*
shows *AE x in S*. *Px x* = *Py x*
proof −
interpret X: *prob-space distr M S X*

```
    using Px by (intro prob-space-distr) simp
  have sigma-finite-measure (distr M S X) ..
  with sigma-finite-density-unique[of Px S Py ] Px Py
  show ?thesis
    by (auto simp: distributed-def)
qed

lemma (in prob-space) distributed-jointI:
  assumes sigma-finite-measure S sigma-finite-measure T
  assumes X[measurable]: X ∈ measurable M S and Y[measurable]: Y ∈ measur-
able M T
  assumes [measurable]: f ∈ borel-measurable (S ⨂M T) and f: AE x in S ⨂M
T. 0 ≤ f x
  assumes eq: ⋀A B. A ∈ sets S ⟹ B ∈ sets T ⟹
    emeasure M {x ∈ space M. X x ∈ A ∧ Y x ∈ B} = (∫+x. (∫+y. f (x, y) ∗
indicator B y ∂T) ∗ indicator A x ∂S)
  shows distributed M (S ⨂M T) (λx. (X x, Y x)) f
  unfolding distributed-def
proof safe
  interpret S: sigma-finite-measure S by fact
  interpret T: sigma-finite-measure T by fact
  interpret ST: pair-sigma-finite S T ..

  from ST.sigma-finite-up-in-pair-measure-generator
  obtain F :: nat ⇒ ('b × 'c) set
    where F: range F ⊆ {A × B |A B. A ∈ sets S ∧ B ∈ sets T} ∧ incseq F ∧
     ⋃ (range F) = space S × space T ∧ (∀i. emeasure (S ⨂M T) (F i) ≠ ∞) ..
  let ?E = {a × b |a b. a ∈ sets S ∧ b ∈ sets T}
  let ?P = S ⨂M T
  show distr M ?P (λx. (X x, Y x)) = density ?P f (is ?L = ?R)
  proof (rule measure-eqI-generator-eq[OF Int-stable-pair-measure-generator[of S
T]])
    show ?E ⊆ Pow (space ?P)
    using sets.space-closed[of S] sets.space-closed[of T] by (auto simp: space-pair-measure)
    show sets ?L = sigma-sets (space ?P) ?E
      by (simp add: sets-pair-measure space-pair-measure)
    then show sets ?R = sigma-sets (space ?P) ?E
      by simp
  next
    interpret L: prob-space ?L
      by (rule prob-space-distr) (auto intro!: measurable-Pair)
    show range F ⊆ ?E (⋃i. F i) = space ?P ⋀i. emeasure ?L (F i) ≠ ∞
      using F by (auto simp: space-pair-measure)
  next
    fix E assume E ∈ ?E
    then obtain A B where E[simp]: E = A × B
      and A[measurable]: A ∈ sets S and B[measurable]: B ∈ sets T by auto
    have emeasure ?L E = emeasure M {x ∈ space M. X x ∈ A ∧ Y x ∈ B}
      by (auto intro!: arg-cong[where f=emeasure M] simp add: emeasure-distr
```

measurable-Pair)
also have ... = ($\int^+ x.$ ($\int^+ y.$ (*f* (*x*, *y*) ∗ *indicator B y*) ∗ *indicator A x* ∂*T*) ∂*S*)
using *f* **by** (*auto simp add*: *eq nn-integral-multc intro*!: *nn-integral-cong*)
also have ... = *emeasure ?R E*
by (*auto simp add*: *emeasure-density T.nn-integral-fst*[*symmetric*] *intro*!: *nn-integral-cong split*: *split-indicator*)
finally show *emeasure ?L E* = *emeasure ?R E* **.**
qed
qed (*auto simp*: *f*)

lemma (**in** *prob-space*) *distributed-swap*:
assumes *sigma-finite-measure S sigma-finite-measure T*
assumes *Pxy*: *distributed M* ($S \bigotimes_M T$) (λ*x*. (*X x*, *Y x*)) *Pxy*
shows *distributed M* ($T \bigotimes_M S$) (λ*x*. (*Y x*, *X x*)) (λ(*x*, *y*). *Pxy* (*y*, *x*))
proof −
interpret *S*: *sigma-finite-measure S* **by** *fact*
interpret *T*: *sigma-finite-measure T* **by** *fact*
interpret *ST*: *pair-sigma-finite S T* **..**
interpret *TS*: *pair-sigma-finite T S* **..**

note *Pxy*[*measurable*]
show *?thesis*
apply (*subst TS.distr-pair-swap*)
unfolding *distributed-def*
proof *safe*
let *?D* = *distr* ($S \bigotimes_M T$) ($T \bigotimes_M S$) (λ(*x*, *y*). (*y*, *x*))
show *1*: (λ(*x*, *y*). *Pxy* (*y*, *x*)) ∈ *borel-measurable ?D*
by *auto*
show *2*: *random-variable* (*distr* ($S \bigotimes_M T$) ($T \bigotimes_M S$) (λ(*x*, *y*). (*y*, *x*))) (λ*x*. (*Y x*, *X x*))
using *Pxy* **by** *auto*
{ fix *A* **assume** *A*: *A* ∈ *sets* ($T \bigotimes_M S$)
let *?B* = (λ(*x*, *y*). (*y*, *x*)) −‘ *A* ∩ *space* ($S \bigotimes_M T$)
from *sets.sets-into-space*[*OF A*]
have *emeasure M* ((λ*x*. (*Y x*, *X x*)) −‘ *A* ∩ *space M*) =
emeasure M ((λ*x*. (*X x*, *Y x*)) −‘ *?B* ∩ *space M*)
by (*auto intro*!: *arg-cong2*[**where** *f*=*emeasure*] *simp*: *space-pair-measure*)
also have ... = ($\int^+ x.$ *Pxy x* ∗ *indicator ?B x* ∂($S \bigotimes_M T$))
using *Pxy A* **by** (*intro distributed-emeasure*) *auto*
finally have *emeasure M* ((λ*x*. (*Y x*, *X x*)) −‘ *A* ∩ *space M*) =
($\int^+ x.$ *Pxy x* ∗ *indicator A* (*snd x*, *fst x*) ∂($S \bigotimes_M T$))
by (*auto intro*!: *nn-integral-cong split*: *split-indicator*) **}**
note ∗ = *this*
show *distr M ?D* (λ*x*. (*Y x*, *X x*)) = *density ?D* (λ(*x*, *y*). *Pxy* (*y*, *x*))
apply (*intro measure-eqI*)
apply (*simp-all add*: *emeasure-distr*[*OF 2*] *emeasure-density*[*OF 1*])
apply (*subst nn-integral-distr*)
apply (*auto intro*!: ∗ *simp*: *comp-def split-beta*)

done
qed
qed

lemma (**in** *prob-space*) *distr-marginal1*:
assumes *sigma-finite-measure S sigma-finite-measure T*
assumes *Pxy*: *distributed M* $(S \bigotimes_M T)$ $(\lambda x.\ (X\ x,\ Y\ x))$ *Pxy*
defines $Px \equiv \lambda x.\ (\int^+ z.\ Pxy\ (x,\ z)\ \partial T)$
shows *distributed M S X Px*
unfolding *distributed-def*
proof *safe*
interpret *S*: *sigma-finite-measure S* **by** *fact*
interpret *T*: *sigma-finite-measure T* **by** *fact*
interpret *ST*: *pair-sigma-finite S T* **..**

note *Pxy*[*measurable*]
show *X*: $X \in$ *measurable M S* **by** *simp*

show *borel*: $Px \in$ *borel-measurable S*
by (*auto intro*!: *T.nn-integral-fst simp*: *Px-def*)

interpret *Pxy*: *prob-space distr M* $(S \bigotimes_M T)$ $(\lambda x.\ (X\ x,\ Y\ x))$
by (*intro prob-space-distr*) *simp*

show *distr M S X* = *density S Px*
proof (*rule measure-eqI*)
fix *A* **assume** *A*: $A \in$ *sets* (*distr M S X*)
with *X measurable-space*[*of Y M T*]
have *emeasure* (*distr M S X*) *A* = *emeasure* (*distr M* $(S \bigotimes_M T)$ $(\lambda x.\ (X\ x,$ *Y x*))) $(A \times$ *space T*)
by (*auto simp add*: *emeasure-distr intro*!: *arg-cong*[**where** *f*=*emeasure M*])
also have ... = *emeasure* (*density* $(S \bigotimes_M T)$ *Pxy*) $(A \times$ *space T*)
using *Pxy* **by** (*simp add*: *distributed-def*)
also have ... = $\int^+ x.\ \int^+ y.$ *Pxy* $(x,\ y) *$ *indicator* $(A \times$ *space T*$)$ $(x,\ y)$ ∂T ∂S
using *A borel Pxy*
by (*simp add*: *emeasure-density T.nn-integral-fst*[*symmetric*])
also have ... = $\int^+ x.$ *Px x* $*$ *indicator A x* ∂S
proof (*rule nn-integral-cong*)
fix *x* **assume** $x \in$ *space S*
moreover have *eq*: $\bigwedge y.\ y \in$ *space T* $\Longrightarrow$ *indicator* $(A \times$ *space T*$)$ $(x,\ y) =$ *indicator A x*
by (*auto simp*: *indicator-def*)
ultimately have $(\int^+ y.$ *Pxy* $(x,\ y) *$ *indicator* $(A \times$ *space T*$)$ $(x,\ y)$ $\partial T) = (\int^+ y.$ *Pxy* $(x,\ y)$ $\partial T) *$ *indicator A x*
by (*simp add*: *eq nn-integral-multc cong*: *nn-integral-cong*)
also have $(\int^+ y.$ *Pxy* $(x,\ y)$ $\partial T) =$ *Px x*
by (*simp add*: *Px-def*)
finally show $(\int^+ y.$ *Pxy* $(x,\ y) *$ *indicator* $(A \times$ *space T*$)$ $(x,\ y)$ $\partial T) =$ *Px*

$x * indicator\ A\ x$ **.**
qed
finally show *emeasure* (*distr M S X*) *A* = *emeasure* (*density S Px*) *A*
using *A borel Pxy* **by** (*simp add*: *emeasure-density*)
qed *simp*
qed

lemma (**in** *prob-space*) *distr-marginal2*:
assumes *S*: *sigma-finite-measure S* **and** *T*: *sigma-finite-measure T*
assumes *Pxy*: *distributed M* ($S \bigotimes_M T$) ($\lambda x.\ (X\ x,\ Y\ x)$) *Pxy*
shows *distributed M T Y* ($\lambda y.\ (\int^+ x.\ Pxy\ (x,\ y)\ \partial S)$)
using *distr-marginal1*[*OF T S distributed-swap*[*OF S T*]] *Pxy* **by** *simp*

lemma (**in** *prob-space*) *distributed-marginal-eq-joint1*:
assumes *T*: *sigma-finite-measure T*
assumes *S*: *sigma-finite-measure S*
assumes *Px*: *distributed M S X Px*
assumes *Pxy*: *distributed M* ($S \bigotimes_M T$) ($\lambda x.\ (X\ x,\ Y\ x)$) *Pxy*
shows *AE x in S*. *Px x* = ($\int^+ y.\ Pxy\ (x,\ y)\ \partial T$)
using *Px distr-marginal1*[*OF S T Pxy*] **by** (*rule distributed-unique*)

lemma (**in** *prob-space*) *distributed-marginal-eq-joint2*:
assumes *T*: *sigma-finite-measure T*
assumes *S*: *sigma-finite-measure S*
assumes *Py*: *distributed M T Y Py*
assumes *Pxy*: *distributed M* ($S \bigotimes_M T$) ($\lambda x.\ (X\ x,\ Y\ x)$) *Pxy*
shows *AE y in T*. *Py y* = ($\int^+ x.\ Pxy\ (x,\ y)\ \partial S$)
using *Py distr-marginal2*[*OF S T Pxy*] **by** (*rule distributed-unique*)

lemma (**in** *prob-space*) *distributed-joint-indep'*:
assumes *S*: *sigma-finite-measure S* **and** *T*: *sigma-finite-measure T*
assumes *X*[*measurable*]: *distributed M S X Px* **and** *Y*[*measurable*]: *distributed M T Y Py*
assumes *indep*: *distr M S X* $\bigotimes_M$ *distr M T Y* = *distr M* ($S \bigotimes_M T$) ($\lambda x.\ (X\ x,\ Y\ x)$)
shows *distributed M* ($S \bigotimes_M T$) ($\lambda x.\ (X\ x,\ Y\ x)$) ($\lambda(x,\ y).\ Px\ x * Py\ y$)
unfolding *distributed-def*
proof *safe*
interpret *S*: *sigma-finite-measure S* **by** *fact*
interpret *T*: *sigma-finite-measure T* **by** *fact*
interpret *ST*: *pair-sigma-finite S T* **..**

interpret *X*: *prob-space density S Px*
unfolding *distributed-distr-eq-density*[*OF X*, *symmetric*]
by (*rule prob-space-distr*) *simp*
have *sf-X*: *sigma-finite-measure* (*density S Px*) **..**

interpret *Y*: *prob-space density T Py*
unfolding *distributed-distr-eq-density*[*OF Y*, *symmetric*]

by (*rule prob-space-distr*) *simp*
have *sf-Y*: *sigma-finite-measure* (*density T Py*) **..**

show *distr M* (*S* $\bigotimes_M$ *T*) (λ*x*. (*X x*, *Y x*)) = *density* (*S* $\bigotimes_M$ *T*) (λ(*x*, *y*). *Px x* ∗ *Py y*)
unfolding *indep*[*symmetric*] *distributed-distr-eq-density*[*OF X*] *distributed-distr-eq-density*[*OF Y*]
using *distributed-borel-measurable*[*OF X*]
using *distributed-borel-measurable*[*OF Y*]
by (*rule pair-measure-density*[*OF - - T sf-Y*])

show *random-variable* (*S* $\bigotimes_M$ *T*) (λ*x*. (*X x*, *Y x*)) **by** *auto*

show *Pxy*: (λ(*x*, *y*). *Px x* ∗ *Py y*) ∈ *borel-measurable* (*S* $\bigotimes_M$ *T*) **by** *auto*
qed

lemma *distributed-integrable*:
distributed M N X f ⟹ *g* ∈ *borel-measurable N* ⟹ (⋀*x*. *x* ∈ *space N* ⟹ 0 ≤ *f x*) ⟹
integrable N (λ*x*. *f x* ∗ *g x*) ⟷ *integrable M* (λ*x*. *g* (*X x*))
supply *distributed-real-measurable*[*measurable*]
by (*auto simp*: *distributed-distr-eq-density*[*symmetric*] *integrable-real-density*[*symmetric*] *integrable-distr-eq*)

lemma *distributed-transform-integrable*:
assumes *Px*: *distributed M N X Px* ⋀*x*. *x* ∈ *space N* ⟹ 0 ≤ *Px x*
assumes *distributed M P Y Py* ⋀*x*. *x* ∈ *space P* ⟹ 0 ≤ *Py x*
assumes *Y*: *Y* = (λ*x*. *T* (*X x*)) **and** *T*: *T* ∈ *measurable N P* **and** *f*: *f* ∈ *borel-measurable P*
shows *integrable P* (λ*x*. *Py x* ∗ *f x*) ⟷ *integrable N* (λ*x*. *Px x* ∗ *f* (*T x*))
proof −
have *integrable P* (λ*x*. *Py x* ∗ *f x*) ⟷ *integrable M* (λ*x*. *f* (*Y x*))
by (*rule distributed-integrable*) *fact+*
also have ... ⟷ *integrable M* (λ*x*. *f* (*T* (*X x*)))
using *Y* **by** *simp*
also have ... ⟷ *integrable N* (λ*x*. *Px x* ∗ *f* (*T x*))
using *measurable-comp*[*OF T f*] *Px* **by** (*intro distributed-integrable*[*symmetric*]) (*auto simp*: *comp-def*)
finally show *?thesis* **.**
qed

lemma *distributed-integrable-var*:
fixes *X* :: *'a* ⇒ *real*
shows *distributed M lborel X* (λ*x*. *ennreal* (*f x*)) ⟹ (⋀*x*. 0 ≤ *f x*) ⟹
integrable lborel (λ*x*. *f x* ∗ *x*) ⟹ *integrable M X*
using *distributed-integrable*[*of M lborel X f* λ*x*. *x*] **by** *simp*

lemma (**in** *prob-space*) *distributed-variance*:
fixes *f*::*real* ⇒ *real*

assumes *D*: *distributed M lborel X f* **and** [*simp*]: $\bigwedge x.\ 0 \leq f\ x$
shows *variance X* = $(\int x.\ x^2 * f\ (x + expectation\ X)\ \partial lborel)$
proof (*subst distributed-integral*[*OF D*, *symmetric*])
show $(\int\ x.\ f\ x * (x - expectation\ X)^2\ \partial lborel) = (\int\ x.\ x^2 * f\ (x + expectation\ X)\ \partial lborel)$
by (*subst lborel-integral-real-affine*[**where** *c=1* **and** *t=expectation X*]) (*auto simp*: *ac-simps*)
qed *simp-all*

lemma (**in** *prob-space*) *variance-affine*:
fixes *f*::*real* ⇒ *real*
assumes [*arith*]: *b* ≠ *0*
assumes *D*[*intro*]: *distributed M lborel X f*
assumes [*simp*]: *prob-space* (*density lborel f*)
assumes *I*[*simp*]: *integrable M X*
assumes *I2*[*simp*]: *integrable M* $(\lambda x.\ (X\ x)^2)$
shows *variance* $(\lambda x.\ a + b * X\ x) = b^2 * variance\ X$
by (*subst variance-eq*)
(*auto simp*: *power2-sum power-mult-distrib prob-space variance-eq right-diff-distrib*)

definition
simple-distributed M X f ⟷
$(\forall x.\ 0 \leq f\ x) \wedge$
distributed M (*count-space* (*X*'*space M*)) *X* $(\lambda x.\ ennreal\ (f\ x)) \wedge$
finite (*X*'*space M*)

lemma *simple-distributed-nonneg*[*dest*]: *simple-distributed M X f* ⟹ $0 \leq f\ x$
by (*auto simp*: *simple-distributed-def*)

lemma *simple-distributed*:
simple-distributed M X Px ⟹ *distributed M* (*count-space* (*X*'*space M*)) *X Px*
unfolding *simple-distributed-def* **by** *auto*

lemma *simple-distributed-finite*[*dest*]: *simple-distributed M X P* ⟹ *finite* (*X*'*space M*)
by (*simp add*: *simple-distributed-def*)

lemma (**in** *prob-space*) *distributed-simple-function-superset*:
assumes *X*: *simple-function M X* $\bigwedge x.\ x \in X$ ' *space M* ⟹ *P x* = *measure M* $(X -$ ' $\{x\} \cap$ *space M*)
assumes *A*: *X*'*space M* ⊆ *A finite A*
defines *S* ≡ *count-space A* **and** *P'* ≡ $(\lambda x.$ *if x* ∈ *X*'*space M then P x else 0*)
shows *distributed M S X P'*
unfolding *distributed-def*
proof *safe*
show $(\lambda x.\ ennreal\ (P'\ x)) \in$ *borel-measurable S* **unfolding** *S-def* **by** *simp*
show *distr M S X* = *density S P'*
proof (*rule measure-eqI-finite*)
show *sets* (*distr M S X*) = *Pow A sets* (*density S P'*) = *Pow A*

```
    using A unfolding S-def by auto
  show finite A by fact
  fix a assume a: a ∈ A
  then have a ∉ X‘space M ⟹ X -‘ {a} ∩ space M = {} by auto
  with A a X have emeasure (distr M S X) {a} = P' a
    by (subst emeasure-distr)
       (auto simp add: S-def P'-def simple-functionD emeasure-eq-measure measurable-count-space-eq2
             intro!: arg-cong[where f=prob])
  also have ... = (∫⁺x. ennreal (P' a) * indicator {a} x ∂S)
    using A X a
    by (subst nn-integral-cmult-indicator)
       (auto simp: S-def P'-def simple-distributed-def simple-functionD measure-nonneg)
  also have ... = (∫⁺x. ennreal (P' x) * indicator {a} x ∂S)
    by (auto simp: indicator-def intro!: nn-integral-cong)
  also have ... = emeasure (density S P') {a}
    using a A by (intro emeasure-density[symmetric]) (auto simp: S-def)
  finally show emeasure (distr M S X) {a} = emeasure (density S P') {a} .
 qed
 show random-variable S X
  using X(1) A by (auto simp: measurable-def simple-functionD S-def)
qed

lemma (in prob-space) simple-distributedI:
 assumes X: simple-function M X
  ⋀x. 0 ≤ P x
  ⋀x. x ∈ X ‘ space M ⟹ P x = measure M (X -‘ {x} ∩ space M)
 shows simple-distributed M X P
 unfolding simple-distributed-def
proof (safe intro!: X)
 have distributed M (count-space (X ‘ space M)) X (λx. ennreal (if x ∈ X‘space M then P x else 0))
  (is ?A)
  using simple-functionD[OF X(1)] by (intro distributed-simple-function-superset[OF X(1,3)]) auto
 also have ?A ⟷ distributed M (count-space (X ‘ space M)) X (λx. ennreal (P x))
  by (rule distributed-cong-density) auto
 finally show ... .
qed (rule simple-functionD[OF X(1)])

lemma simple-distributed-joint-finite:
 assumes X: simple-distributed M (λx. (X x, Y x)) Px
 shows finite (X ‘ space M) finite (Y ‘ space M)
proof -
 have finite ((λx. (X x, Y x)) ‘ space M)
  using X by (auto simp: simple-distributed-def simple-functionD)
 then have finite (fst ‘ (λx. (X x, Y x)) ‘ space M) finite (snd ‘ (λx. (X x, Y x))
```

‘ *space M*)
by *auto*
then show *fin*: *finite* (*X* ‘ *space M*) *finite* (*Y* ‘ *space M*)
by (*auto simp*: *image-image*)
qed

lemma *simple-distributed-joint2-finite*:
assumes *X*: *simple-distributed M* (λ*x*. (*X x*, *Y x*, *Z x*)) *Px*
shows *finite* (*X* ‘ *space M*) *finite* (*Y* ‘ *space M*) *finite* (*Z* ‘ *space M*)
proof −
have *finite* ((λ*x*. (*X x*, *Y x*, *Z x*)) ‘ *space M*)
using *X* **by** (*auto simp*: *simple-distributed-def simple-functionD*)
then have *finite* (*fst* ‘ (λ*x*. (*X x*, *Y x*, *Z x*)) ‘ *space M*)
finite ((*fst* ∘ *snd*) ‘ (λ*x*. (*X x*, *Y x*, *Z x*)) ‘ *space M*)
finite ((*snd* ∘ *snd*) ‘ (λ*x*. (*X x*, *Y x*, *Z x*)) ‘ *space M*)
by *auto*
then show *fin*: *finite* (*X* ‘ *space M*) *finite* (*Y* ‘ *space M*) *finite* (*Z* ‘ *space M*)
by (*auto simp*: *image-image*)
qed

lemma *simple-distributed-simple-function*:
simple-distributed M X Px ⟹ *simple-function M X*
unfolding *simple-distributed-def distributed-def*
by (*auto simp*: *simple-function-def measurable-count-space-eq2*)

lemma *simple-distributed-measure*:
simple-distributed M X P ⟹ *a* ∈ *X*‘*space M* ⟹ *P a* = *measure M* (*X* −‘ {*a*} ∩ *space M*)
using *distributed-count-space*[*of M X*‘*space M X P a*, *symmetric*]
by (*auto simp*: *simple-distributed-def measure-def*)

lemma (**in** *prob-space*) *simple-distributed-joint*:
assumes *X*: *simple-distributed M* (λ*x*. (*X x*, *Y x*)) *Px*
defines *S* ≡ *count-space* (*X*‘*space M*) ⨂$_M$ *count-space* (*Y*‘*space M*)
defines *P* ≡ (λ*x*. *if x* ∈ (λ*x*. (*X x*, *Y x*))‘*space M then Px x else 0*)
shows *distributed M S* (λ*x*. (*X x*, *Y x*)) *P*
proof −
from *simple-distributed-joint-finite*[*OF X*, *simp*]
have *S-eq*: *S* = *count-space* (*X*‘*space M* × *Y*‘*space M*)
by (*simp add*: *S-def pair-measure-count-space*)
show *?thesis*
unfolding *S-eq P-def*
proof (*rule distributed-simple-function-superset*)
show *simple-function M* (λ*x*. (*X x*, *Y x*))
using *X* **by** (*rule simple-distributed-simple-function*)
fix *x* **assume** *x* ∈ (λ*x*. (*X x*, *Y x*)) ‘ *space M*
from *simple-distributed-measure*[*OF X this*]
show *Px x* = *prob* ((λ*x*. (*X x*, *Y x*)) −‘ {*x*} ∩ *space M*) .
qed *auto*

qed

lemma (**in** *prob-space*) *simple-distributed-joint2*:
assumes *X*: *simple-distributed M* (λ*x*. (*X x*, *Y x*, *Z x*)) *Px*
defines *S* ≡ *count-space* (*X*'*space M*) ⨂$_M$ *count-space* (*Y*'*space M*) ⨂$_M$ *count-space* (*Z*'*space M*)
defines *P* ≡ (λ*x*. *if x* ∈ (λ*x*. (*X x*, *Y x*, *Z x*))'*space M then Px x else 0*)
shows *distributed M S* (λ*x*. (*X x*, *Y x*, *Z x*)) *P*
proof −
from *simple-distributed-joint2-finite*[*OF X*, *simp*]
have *S-eq*: *S* = *count-space* (*X*'*space M* × *Y*'*space M* × *Z*'*space M*)
by (*simp add*: *S-def pair-measure-count-space*)
show *?thesis*
unfolding *S-eq P-def*
proof (*rule distributed-simple-function-superset*)
show *simple-function M* (λ*x*. (*X x*, *Y x*, *Z x*))
using *X* **by** (*rule simple-distributed-simple-function*)
fix *x* **assume** *x* ∈ (λ*x*. (*X x*, *Y x*, *Z x*)) ' *space M*
from *simple-distributed-measure*[*OF X this*]
show *Px x* = *prob* ((λ*x*. (*X x*, *Y x*, *Z x*)) −' {*x*} ∩ *space M*) **.**
qed *auto*
qed

lemma (**in** *prob-space*) *simple-distributed-sum-space*:
assumes *X*: *simple-distributed M X f*
shows *sum f* (*X*'*space M*) = *1*
proof −
from *X* **have** *sum f* (*X*'*space M*) = *prob* (⋃ *i*∈*X*'*space M*. *X* −' {*i*} ∩ *space M*)
by (*subst finite-measure-finite-Union*)
(*auto simp add*: *disjoint-family-on-def simple-distributed-measure simple-distributed-simple-function simple-functionD*
intro!: *sum.cong arg-cong*[**where** *f*=*prob*])
also have ... = *prob* (*space M*)
by (*auto intro*!: *arg-cong*[**where** *f*=*prob*])
finally show *?thesis*
using *emeasure-space-1* **by** (*simp add*: *emeasure-eq-measure*)
qed

lemma (**in** *prob-space*) *distributed-marginal-eq-joint-simple*:
assumes *Px*: *simple-function M X*
assumes *Py*: *simple-distributed M Y Py*
assumes *Pxy*: *simple-distributed M* (λ*x*. (*X x*, *Y x*)) *Pxy*
assumes *y*: *y* ∈ *Y*'*space M*
shows *Py y* = (∑ *x*∈*X*'*space M*. *if* (*x*, *y*) ∈ (λ*x*. (*X x*, *Y x*)) ' *space M then Pxy* (*x*, *y*) *else 0*)
proof −
note *Px* = *simple-distributedI*[*OF Px measure-nonneg refl*]
have *AE y in count-space* (*Y* ' *space M*). *ennreal* (*Py y*) =
∫$^+$ *x*. *ennreal* (*if* (*x*, *y*) ∈ (λ*x*. (*X x*, *Y x*)) ' *space M then Pxy* (*x*, *y*) *else*

0) *∂count-space* (*X ' space M*)
using *sigma-finite-measure-count-space-finite sigma-finite-measure-count-space-finite*
simple-distributed[*OF Py*] *simple-distributed-joint*[*OF Pxy*]
by (*rule distributed-marginal-eq-joint2*)
(*auto intro*: *Py Px simple-distributed-finite*)
then have *ennreal* (*Py y*) =
(∑ *x*∈*X'space M*. *ennreal* (*if* (*x*, *y*) ∈ (λ*x*. (*X x*, *Y x*)) ' *space M then Pxy* (*x*, *y*) *else 0*))
using *y Px*[*THEN simple-distributed-finite*]
by (*auto simp*: *AE-count-space nn-integral-count-space-finite*)
also have ... = (∑ *x*∈*X'space M*. *if* (*x*, *y*) ∈ (λ*x*. (*X x*, *Y x*)) ' *space M then Pxy* (*x*, *y*) *else 0*)
using *Pxy* **by** (*intro sum-ennreal*) *auto*
finally show *?thesis*
using *simple-distributed-nonneg*[*OF Py*] *simple-distributed-nonneg*[*OF Pxy*]
by (*subst* (*asm*) *ennreal-inj*) (*auto intro*!: *sum-nonneg*)
qed

lemma *distributedI-real*:
fixes *f* :: *'a* ⇒ *real*
assumes *gen*: *sets M1* = *sigma-sets* (*space M1*) *E* **and** *Int-stable E*
and *A*: *range A* ⊆ *E* (⋃ *i*::*nat*. *A i*) = *space M1* ⋀*i*. *emeasure* (*distr M M1 X*) (*A i*) ≠ ∞
and *X*: *X* ∈ *measurable M M1*
and *f*: *f* ∈ *borel-measurable M1 AE x in M1*. *0* ≤ *f x*
and *eq*: ⋀*A*. *A* ∈ *E* ⟹ *emeasure M* (*X* −' *A* ∩ *space M*) = (∫⁺ *x*. *f x* ∗ *indicator A x ∂M1*)
shows *distributed M M1 X f*
unfolding *distributed-def*
proof (*intro conjI*)
show *distr M M1 X* = *density M1 f*
proof (*rule measure-eqI-generator-eq*[**where** *A*=*A*])
{ **fix** *A* **assume** *A*: *A* ∈ *E*
then have *A* ∈ *sigma-sets* (*space M1*) *E* **by** *auto*
then have *A* ∈ *sets M1*
using *gen* **by** *simp*
with *f A eq*[*of A*] *X* **show** *emeasure* (*distr M M1 X*) *A* = *emeasure* (*density M1 f*) *A*
by (*auto simp add*: *emeasure-distr emeasure-density ennreal-indicator*
intro!: *nn-integral-cong split*: *split-indicator*) }
note *eq-E* = *this*
show *Int-stable E* **by** *fact*
{ **fix** *e* **assume** *e* ∈ *E*
then have *e* ∈ *sigma-sets* (*space M1*) *E* **by** *auto*
then have *e* ∈ *sets M1* **unfolding** *gen* .
then have *e* ⊆ *space M1* **by** (*rule sets.sets-into-space*) }
then show *E* ⊆ *Pow* (*space M1*) **by** *auto*
show *sets* (*distr M M1 X*) = *sigma-sets* (*space M1*) *E*
sets (*density M1* (λ*x*. *ennreal* (*f x*))) = *sigma-sets* (*space M1*) *E*

```
    unfolding gen[symmetric] by auto
  qed fact+
qed (insert X f, auto)

lemma distributedI-borel-atMost:
  fixes f :: real ⇒ real
  assumes [measurable]: X ∈ borel-measurable M
    and [measurable]: f ∈ borel-measurable borel and f[simp]: AE x in lborel. 0 ≤
f x
    and g-eq: ⋀a. (∫⁺x. f x * indicator {..a} x ∂lborel)  = ennreal (g a)
    and M-eq: ⋀a. emeasure M {x∈space M. X x ≤ a} = ennreal (g a)
  shows distributed M lborel X f
proof (rule distributedI-real)
  show sets (lborel::real measure) = sigma-sets (space lborel) (range atMost)
    by (simp add: borel-eq-atMost)
  show Int-stable (range atMost :: real set set)
    by (auto simp: Int-stable-def)
  have vimage-eq: ⋀a. (X -` {..a} ∩ space M) = {x∈space M. X x ≤ a} by auto
  define A where A i = {.. real i} for i :: nat
  then show range A ⊆ range atMost (⋃i. A i) = space lborel
    ⋀i. emeasure (distr M lborel X) (A i) ≠ ∞
    by (auto simp: real-arch-simple emeasure-distr vimage-eq M-eq)

  fix A :: real set assume A ∈ range atMost
  then obtain a where A: A = {..a} by auto
  show emeasure M (X -` A ∩ space M) = (∫⁺x. f x * indicator A x ∂lborel)
    unfolding vimage-eq A M-eq g-eq ..
qed auto

lemma (in prob-space) uniform-distributed-params:
  assumes X: distributed M MX X (λx. indicator A x / measure MX A)
  shows A ∈ sets MX measure MX A ≠ 0
proof -
  interpret X: prob-space distr M MX X
    using distributed-measurable[OF X] by (rule prob-space-distr)

  show measure MX A ≠ 0
  proof
    assume measure MX A = 0
    with X.emeasure-space-1 X.prob-space distributed-distr-eq-density[OF X]
    show False
      by (simp add: emeasure-density zero-ennreal-def[symmetric])
  qed
  with measure-notin-sets[of A MX] show A ∈ sets MX
    by blast
qed

lemma prob-space-uniform-measure:
  assumes A: emeasure M A ≠ 0 emeasure M A ≠ ∞
```

shows *prob-space* (*uniform-measure M A*)
proof
show *emeasure* (*uniform-measure M A*) (*space* (*uniform-measure M A*)) = *1*
using *emeasure-uniform-measure*[*OF emeasure-neq-0-sets*[*OF A*(*1*)], *of space M*]
using *sets.sets-into-space*[*OF emeasure-neq-0-sets*[*OF A*(*1*)]] *A*
by (*simp add*: *Int-absorb2 less-top*)
qed

lemma *prob-space-uniform-count-measure*: *finite A* ⟹ *A* ≠ {} ⟹ *prob-space* (*uniform-count-measure A*)
by *standard* (*auto simp*: *emeasure-uniform-count-measure space-uniform-count-measure one-ennreal-def*)

lemma (**in** *prob-space*) *measure-uniform-measure-eq-cond-prob*:
assumes [*measurable*]: *Measurable.pred M P Measurable.pred M Q*
shows *P*(*x in uniform-measure M* {*x*∈*space M*. *Q x*}. *P x*) = *P*(*x in M*. *P x* | *Q x*)
proof *cases*
assume *Q*: *measure M* {*x*∈*space M*. *Q x*} = *0*
then have ∗: *AE x in M*. ¬ *Q x*
by (*simp add*: *prob-eq-0*)
then have *density M* (λ*x*. *indicator* {*x* ∈ *space M*. *Q x*} *x* / *emeasure M* {*x* ∈ *space M*. *Q x*}) = *density M* (λ*x*. *0*)
by (*intro density-cong*) *auto*
with ∗ **show** *?thesis*
unfolding *uniform-measure-def*
by (*simp add*: *emeasure-density measure-def cond-prob-def emeasure-eq-0-AE*)
next
assume *Q*: *measure M* {*x*∈*space M*. *Q x*} ≠ *0*
then show *P*(*x in uniform-measure M* {*x* ∈ *space M*. *Q x*}. *P x*) = *cond-prob M P Q*
by (*subst measure-uniform-measure*)
(*auto simp*: *emeasure-eq-measure cond-prob-def measure-nonneg intro*!: *arg-cong*[**where** *f*=*prob*])
qed

lemma *prob-space-point-measure*:
finite S ⟹ (⋀*s*. *s* ∈ *S* ⟹ *0* ≤ *p s*) ⟹ (∑ *s*∈*S*. *p s*) = *1* ⟹ *prob-space* (*point-measure S p*)
by (*rule prob-spaceI*) (*simp add*: *space-point-measure emeasure-point-measure-finite*)

lemma (**in** *prob-space*) *distr-pair-fst*: *distr* (*N* ⨂$_M$ *M*) *N fst* = *N*
proof (*intro measure-eqI*)
fix *A* **assume** *A*: *A* ∈ *sets* (*distr* (*N* ⨂$_M$ *M*) *N fst*)
from *A* **have** *emeasure* (*distr* (*N* ⨂$_M$ *M*) *N fst*) *A* = *emeasure* (*N* ⨂$_M$ *M*) (*A* × *space M*)
by (*auto simp add*: *emeasure-distr space-pair-measure dest*: *sets.sets-into-space intro*!: *arg-cong2*[**where** *f*=*emeasure*])

with *A* **show** *emeasure* (*distr* (*N* $\bigotimes_M$ *M*) *N fst*) *A* = *emeasure N A*
by (*simp add*: *emeasure-pair-measure-Times emeasure-space-1*)
qed *simp*

lemma (**in** *product-prob-space*) *distr-reorder*:
assumes *inj-on t J t* ∈ *J* → *K finite K*
shows *distr* (*PiM K M*) (*Pi*$_M$ *J* (λ*x*. *M* (*t x*))) (λω. λ*n*∈*J*. ω (*t n*)) = *PiM J* (λ*x*. *M* (*t x*))
proof (*rule product-sigma-finite.PiM-eqI*)
show *product-sigma-finite* (λ*x*. *M* (*t x*)) ..
have *t*'*J* ⊆ *K* **using** *assms* **by** *auto*
then show [*simp*]: *finite J*
by (*rule finite-imageD*[*OF finite-subset*]) *fact*+
fix *A* **assume** *A*: ⋀*i*. *i* ∈ *J* ⟹ *A i* ∈ *sets* (*M* (*t i*))
moreover have ((λω. λ*n*∈*J*. ω (*t n*)) −' *Pi*$_E$ *J A* ∩ *space* (*Pi*$_M$ *K M*)) =
(Π$_E$ *i*∈*K*. *if i* ∈ *t*'*J then A* (*the-inv-into J t i*) *else space* (*M i*))
using *A A*[*THEN sets.sets-into-space*] ‹*t* ∈ *J* → *K*› ‹*inj-on t J*›
by (*subst prod-emb-Pi*[*symmetric*]) (*auto simp*: *space-PiM PiE-iff the-inv-into-f-f prod-emb-def*)
ultimately show *distr* (*Pi*$_M$ *K M*) (*Pi*$_M$ *J* (λ*x*. *M* (*t x*))) (λω. λ*n*∈*J*. ω (*t n*)) (*Pi*$_E$ *J A*) = (∏ *i*∈*J*. *M* (*t i*) (*A i*))
using *assms*
apply (*subst emeasure-distr*)
apply (*auto intro*!: *sets-PiM-I-finite simp*: *Pi-iff*)
apply (*subst emeasure-PiM*)
apply (*auto simp*: *the-inv-into-f-f* ‹*inj-on t J*› *prod.reindex*[*OF* ‹*inj-on t J*›]
if-distrib[**where** *f*=*emeasure* (*M* -)] *prod.If-cases emeasure-space-1 Int-absorb1* ‹*t*'*J* ⊆ *K*›)
done
qed *simp*

lemma (**in** *product-prob-space*) *distr-restrict*:
J ⊆ *K* ⟹ *finite K* ⟹ (Π$_M$ *i*∈*J*. *M i*) = *distr* (Π$_M$ *i*∈*K*. *M i*) (Π$_M$ *i*∈*J*. *M i*) (λ*f*. *restrict f J*)
using *distr-reorder*[*of* λ*x*. *x J K*] **by** (*simp add*: *Pi-iff subset-eq*)

lemma (**in** *product-prob-space*) *emeasure-prod-emb*[*simp*]:
assumes *L*: *J* ⊆ *L finite L* **and** *X*: *X* ∈ *sets* (*Pi*$_M$ *J M*)
shows *emeasure* (*Pi*$_M$ *L M*) (*prod-emb L M J X*) = *emeasure* (*Pi*$_M$ *J M*) *X*
by (*subst distr-restrict*[*OF L*])
(*simp add*: *prod-emb-def space-PiM emeasure-distr measurable-restrict-subset L X*)

lemma *emeasure-distr-restrict*:
assumes *I* ⊆ *K* **and** *Q*[*measurable-cong*]: *sets Q* = *sets* (*PiM K M*) **and** *A*[*measurable*]: *A* ∈ *sets* (*PiM I M*)
shows *emeasure* (*distr Q* (*PiM I M*) (λω. *restrict* ω *I*)) *A* = *emeasure Q* (*prod-emb K M I A*)
using ‹*I*⊆*K*› *sets-eq-imp-space-eq*[*OF Q*]

by (*subst emeasure-distr*)
(*auto simp*: *measurable-cong-sets*[*OF Q*] *prod-emb-def space-PiM*[*symmetric*] *intro*!: *measurable-restrict*)

lemma (**in** *prob-space*) *prob-space-completion*: *prob-space* (*completion M*)
by (*rule prob-spaceI*) (*simp add*: *emeasure-space-1*)

lemma *distr-PiM-finite-prob-space*:
assumes *fin*: *finite I*
assumes *product-prob-space M*
assumes *product-prob-space M'*
assumes [*measurable*]: $\bigwedge i.\ i \in I \Longrightarrow f \in$ *measurable* (*M i*) (*M' i*)
shows *distr* (*PiM I M*) (*PiM I M'*) (*compose I f*) = *PiM I* ($\lambda i.$ *distr* (*M i*) (*M' i*) *f*)
proof −
interpret *M*: *product-prob-space M* **by** *fact*
interpret *M'*: *product-prob-space M'* **by** *fact*
define *N* **where** *N* = ($\lambda i.$ *if i* ∈ *I then distr* (*M i*) (*M' i*) *f else M' i*)
have [*intro*]: *prob-space* (*N i*) **for** *i*
by (*auto simp*: *N-def intro*!: *M.M.prob-space-distr M'.prob-space*)

interpret *N*: *product-prob-space N*
by (*intro product-prob-spaceI*) (*auto simp*: *N-def M'.prob-space intro*: *M.M.prob-space-distr*)

have *distr* (*PiM I M*) (*PiM I M'*) (*compose I f*) = *PiM I N*
proof (*rule N.PiM-eqI*)
have *N-events-eq*: *sets* (*Pi*$_M$ *I N*) = *sets* (*Pi*$_M$ *I M'*)
unfolding *N-def* **by** (*intro sets-PiM-cong*) *auto*
also have ... = *sets* (*distr* (*Pi*$_M$ *I M*) (*Pi*$_M$ *I M'*) (*compose I f*))
by *simp*
finally show *sets* (*distr* (*Pi*$_M$ *I M*) (*Pi*$_M$ *I M'*) (*compose I f*)) = *sets* (*Pi*$_M$ *I N*) **..**

fix *A* **assume** *A*: $\bigwedge i.\ i \in I \Longrightarrow A\ i \in N.M.events\ i$
have *emeasure* (*distr* (*Pi*$_M$ *I M*) (*Pi*$_M$ *I M'*) (*compose I f*)) (*Pi*$_E$ *I A*) =
emeasure (*Pi*$_M$ *I M*) (*compose I f* −‘ *Pi*$_E$ *I A* ∩ *space* (*Pi*$_M$ *I M*))
proof (*intro emeasure-distr*)
show *compose I f* ∈ *Pi*$_M$ *I M* →$_M$ *Pi*$_M$ *I M'*
unfolding *compose-def* **by** *measurable*
show *Pi*$_E$ *I A* ∈ *sets* (*Pi*$_M$ *I M'*)
unfolding *N-events-eq* [*symmetric*] **by** (*intro sets-PiM-I-finite fin A*)
qed
also have *compose I f* −‘ *Pi*$_E$ *I A* ∩ *space* (*Pi*$_M$ *I M*) = *Pi*$_E$ *I* ($\lambda i.$ *f* −‘ *A i* ∩ *space* (*M i*))
using *A* **by** (*auto simp*: *space-PiM PiE-def Pi-def extensional-def N-def compose-def*)
also have *emeasure* (*Pi*$_M$ *I M*) (*Pi*$_E$ *I* ($\lambda i.$ *f* −‘ *A i* ∩ *space* (*M i*))) =
($\prod i \in I.$ *emeasure* (*M i*) (*f* −‘ *A i* ∩ *space* (*M i*)))
using *A* **by** (*intro M.emeasure-PiM fin*) (*auto simp*: *N-def*)

also have ... = ($\prod i{\in}I$. *emeasure* (*distr* (*M i*) (*M′ i*) *f*) (*A i*))
using *A* **by** (*intro prod.cong emeasure-distr* [*symmetric*]) (*auto simp*: *N-def*)
also have ... = ($\prod i{\in}I$. *emeasure* (*N i*) (*A i*))
unfolding *N-def* **by** (*intro prod.cong*) (*auto simp*: *N-def*)
finally show *emeasure* (*distr* (Pi_M *I M*) (Pi_M *I M′*) (*compose I f*)) (Pi_E *I A*) = ... **.**
qed *fact+*
also have *PiM I N* = *PiM I* (λ*i*. *distr* (*M i*) (*M′ i*) *f*)
by (*intro PiM-cong*) (*auto simp*: *N-def*)
finally show *?thesis* **.**
qed

end

2 Distribution Functions

Shows that the cumulative distribution function (cdf) of a distribution (a measure on the reals) is nondecreasing and right continuous, which tends to 0 and 1 in either direction.

Conversely, every such function is the cdf of a unique distribution. This direction defines the measure in the obvious way on half-open intervals, and then applies the Caratheodory extension theorem.

theory *Distribution-Functions*
imports *Probability-Measure*
begin

lemma *UN-Ioc-eq-UNIV*: ($\bigcup n$. { −*real n* <.. *real n*}) = *UNIV*
by *auto*
(*metis le-less-trans minus-minus neg-less-iff-less not-le real-arch-simple of-nat-0-le-iff reals-Archimedean2*)

2.1 Properties of cdf's

definition
cdf :: *real measure* ⇒ *real* ⇒ *real*
where
cdf M ≡ λ*x*. *measure M* {..*x*}

lemma *cdf-def2*: *cdf M x* = *measure M* {..*x*}
by (*simp add*: *cdf-def*)

locale *finite-borel-measure* = *finite-measure M* **for** *M* :: *real measure* +
assumes *M-is-borel*: *sets M* = *sets borel*
begin

lemma *sets-M*[*intro*]: *a* ∈ *sets borel* ⟹ *a* ∈ *sets M*
using *M-is-borel* **by** *auto*

lemma *cdf-diff-eq*:
assumes $x < y$
shows *cdf M y − cdf M x = measure M {x<..y}*
proof −
from *assms* **have** ∗: $\{..x\} \cup \{x<..y\} = \{..y\}$ **by** *auto*
have *measure M {..y} = measure M {..x} + measure M {x<..y}*
by (*subst finite-measure-Union* [*symmetric*], *auto simp add*: ∗)
thus *?thesis*
unfolding *cdf-def* **by** *auto*
qed

lemma *cdf-nondecreasing*: $x \leq y \Longrightarrow$ *cdf M x ≤ cdf M y*
unfolding *cdf-def* **by** (*auto intro*!: *finite-measure-mono*)

lemma *borel-UNIV*: *space M = UNIV*
by (*metis in-mono sets.sets-into-space space-in-borel top-le M-is-borel*)

lemma *cdf-nonneg*: *cdf M x ≥ 0*
unfolding *cdf-def* **by** (*rule measure-nonneg*)

lemma *cdf-bounded*: *cdf M x ≤ measure M* (*space M*)
unfolding *cdf-def* **by** (*intro bounded-measure*)

lemma *cdf-lim-infty*:
((λ*i*. *cdf M* (*real i*)) ⟶ *measure M* (*space M*))
proof −
have (λ*i*. *cdf M* (*real i*)) ⟶ *measure M* (⋃ *i*::*nat*. {..*real i*})
unfolding *cdf-def* **by** (*rule finite-Lim-measure-incseq*) (*auto simp*: *incseq-def*)
also have (⋃ *i*::*nat*. {..*real i*}) = *space M*
by (*auto simp*: *borel-UNIV intro*: *real-arch-simple*)
finally show *?thesis* **.**
qed

lemma *cdf-lim-at-top*: (*cdf M* ⟶ *measure M* (*space M*)) *at-top*
by (*rule tendsto-at-topI-sequentially-real*)
(*simp-all add*: *mono-def cdf-nondecreasing cdf-lim-infty*)

lemma *cdf-lim-neg-infty*: ((λ*i*. *cdf M* (− *real i*)) ⟶ *0*)
proof −
have (λ*i*. *cdf M* (− *real i*)) ⟶ *measure M* (⋂ *i*::*nat*. {.. − *real i* })
unfolding *cdf-def* **by** (*rule finite-Lim-measure-decseq*) (*auto simp*: *decseq-def*)
also have (⋂ *i*::*nat*. {..− *real i*}) = {}
by *auto* (*metis leD le-minus-iff reals-Archimedean2*)
finally show *?thesis*
by *simp*
qed

lemma *cdf-lim-at-bot*: (*cdf M* ⟶ *0*) *at-bot*

proof −
have ∗: ((λx :: *real*. − *cdf M* (− *x*)) ⟶ *0*) *at-top*
by (*intro tendsto-at-topI-sequentially-real monoI*)
(*auto simp*: *cdf-nondecreasing cdf-lim-neg-infty tendsto-minus-cancel-left*[*symmetric*])
from *filterlim-compose* [*OF* ∗, *OF filterlim-uminus-at-top-at-bot*]
show *?thesis*
unfolding *tendsto-minus-cancel-left*[*symmetric*] **by** *simp*
qed

lemma *cdf-is-right-cont*: *continuous* (*at-right a*) (*cdf M*)
unfolding *continuous-within*
proof (*rule tendsto-at-right-sequentially*[**where** *b*=*a* + *1*])
fix *f* :: *nat* ⇒ *real* **and** *x* **assume** *f*: *decseq f f* ⟶ *a*
then have (λ*n*. *cdf M* (*f n*)) ⟶ *measure M* (⋂ *i*. {.. *f i*})
using ‹*decseq f*› **unfolding** *cdf-def*
by (*intro finite-Lim-measure-decseq*) (*auto simp*: *decseq-def*)
also have (⋂ *i*. {.. *f i*}) = {.. *a*}
using *decseq-ge*[*OF f*] **by** (*auto intro*: *order-trans LIMSEQ-le-const*[*OF f*(*2*)])
finally show (λ*n*. *cdf M* (*f n*)) ⟶ *cdf M a*
by (*simp add*: *cdf-def*)
qed *simp*

lemma *cdf-at-left*: (*cdf M* ⟶ *measure M* {..<*a*}) (*at-left a*)
proof (*rule tendsto-at-left-sequentially*[*of a* − *1*])
fix *f* :: *nat* ⇒ *real* **and** *x* **assume** *f*: *incseq f f* ⟶ *a* ⋀*x*. *f x* < *a* ⋀*x*. *a* − *1* < *f x*
then have (λ*n*. *cdf M* (*f n*)) ⟶ *measure M* (⋃ *i*. {.. *f i*})
using ‹*incseq f*› **unfolding** *cdf-def*
by (*intro finite-Lim-measure-incseq*) (*auto simp*: *incseq-def*)
also have (⋃ *i*. {.. *f i*}) = {..<*a*}
by (*auto dest*!: *order-tendstoD*(*1*)[*OF f*(*2*)] *eventually-happens'*[*OF sequentially-bot*]
intro: *less-imp-le le-less-trans f*(*3*))
finally show (λ*n*. *cdf M* (*f n*)) ⟶ *measure M* {..<*a*}
by (*simp add*: *cdf-def*)
qed *auto*

lemma *isCont-cdf*: *isCont* (*cdf M*) *x* ⟷ *measure M* {*x*} = *0*
proof −
have *isCont* (*cdf M*) *x* ⟷ *cdf M x* = *measure M* {..<*x*}
by (*auto simp*: *continuous-at-split cdf-is-right-cont continuous-within*[**where** *s*={..< -}]
cdf-at-left tendsto-unique[*OF* - *cdf-at-left*])
also have *cdf M x* = *measure M* {..<*x*} ⟷ *measure M* {*x*} = *0*
unfolding *cdf-def ivl-disj-un*(*2*)[*symmetric*]
by (*subst finite-measure-Union*) *auto*
finally show *?thesis* .
qed

lemma *countable-atoms*: *countable* $\{x.\ measure\ M\ \{x\} > 0\}$
using *countable-support* **unfolding** *zero-less-measure-iff* **.**

end

locale *real-distribution* = *prob-space M* **for** *M* :: *real measure* +
assumes *events-eq-borel* [*simp*, *measurable-cong*]: *sets M* = *sets borel*
begin

lemma *finite-borel-measure-M*: *finite-borel-measure M*
by *standard auto*

sublocale *finite-borel-measure M*
by (*rule finite-borel-measure-M*)

lemma *space-eq-univ* [*simp*]: *space M* = *UNIV*
using *events-eq-borel*[*THEN sets-eq-imp-space-eq*] **by** *simp*

lemma *cdf-bounded-prob*: $\bigwedge x.$ *cdf M x* $\leq$ *1*
by (*subst prob-space* [*symmetric*], *rule cdf-bounded*)

lemma *cdf-lim-infty-prob*: $(\lambda i.$ *cdf M* (*real i*)) $\longrightarrow$ *1*
by (*subst prob-space* [*symmetric*], *rule cdf-lim-infty*)

lemma *cdf-lim-at-top-prob*: (*cdf M* $\longrightarrow$ *1*) *at-top*
by (*subst prob-space* [*symmetric*], *rule cdf-lim-at-top*)

lemma *measurable-finite-borel* [*simp*]:
f $\in$ *borel-measurable borel* $\Longrightarrow$ *f* $\in$ *borel-measurable M*
by (*rule borel-measurable-subalgebra*[**where** *N*=*borel*]) *auto*

end

lemma (**in** *prob-space*) *real-distribution-distr* [*intro*, *simp*]:
random-variable borel X $\Longrightarrow$ *real-distribution* (*distr M borel X*)
unfolding *real-distribution-def real-distribution-axioms-def* **by** (*auto intro*!: *prob-space-distr*)

2.2 Uniqueness

lemma (**in** *finite-borel-measure*) *emeasure-Ioc*:
assumes *a* $\leq$ *b* **shows** *emeasure M* $\{a <..\ b\}$ = *cdf M b* $-$ *cdf M a*
proof $-$
have $\{a <..\ b\} = \{..b\} - \{..a\}$
by *auto*
moreover have $\{..x\} \in$ *sets M* **for** *x*
using *atMost-borel*[*of x*] *M-is-borel* **by** *auto*
moreover note ‹*a* $\leq$ *b*›
ultimately show *?thesis*
by (*simp add*: *emeasure-eq-measure finite-measure-Diff cdf-def*)

qed

lemma *cdf-unique'*:
fixes *M1 M2*
assumes *finite-borel-measure M1* **and** *finite-borel-measure M2*
assumes *cdf M1 = cdf M2*
shows *M1 = M2*
proof (*rule measure-eqI-generator-eq*[**where** Ω=*UNIV*])
fix *X* **assume** *X* ∈ *range* (λ(*a*, *b*). {*a*<..*b*::*real*})
then obtain *a b* **where** *Xeq*: *X* = {*a*<..*b*} **by** *auto*
then show *emeasure M1 X = emeasure M2 X*
by (*cases a* ≤ *b*)
(*simp-all add*: *assms*(*1*,*2*)[*THEN finite-borel-measure.emeasure-Ioc*] *assms*(*3*))
next
show (⋃ *i*. {− *real* (*i*::*nat*)<..*real i*}) = *UNIV*
by (*rule UN-Ioc-eq-UNIV*)
qed (*auto simp*: *finite-borel-measure.emeasure-Ioc*[*OF assms*(*1*)]
assms(*1*,*2*)[*THEN finite-borel-measure.M-is-borel*] *borel-sigma-sets-Ioc*
Int-stable-def)

lemma *cdf-unique*:
real-distribution M1 ⟹ *real-distribution M2* ⟹ *cdf M1 = cdf M2* ⟹ *M1 = M2*
using *cdf-unique'*[*of M1 M2*] **by** (*simp add*: *real-distribution.finite-borel-measure-M*)

lemma
fixes *F* :: *real* ⇒ *real*
assumes *nondecF* : ⋀ *x y*. *x* ≤ *y* ⟹ *F x* ≤ *F y*
and *right-cont-F* : ⋀*a*. *continuous* (*at-right a*) *F*
and *lim-F-at-bot* : (*F* ⟶ *0*) *at-bot*
and *lim-F-at-top* : (*F* ⟶ *m*) *at-top*
and *m*: *0* ≤ *m*
shows *interval-measure-UNIV*: *emeasure* (*interval-measure F*) *UNIV* = *m*
and *finite-borel-measure-interval-measure*: *finite-borel-measure* (*interval-measure F*)
proof −
let *?F* = *interval-measure F*
{ have *ennreal* (*m* − *0*) = (*SUP i*. *ennreal* (*F* (*real i*) − *F* (− *real i*)))
by (*intro LIMSEQ-unique*[*OF - LIMSEQ-SUP*] *tendsto-ennrealI tendsto-intros*
lim-F-at-bot[*THEN filterlim-compose*] *lim-F-at-top*[*THEN filterlim-compose*]
lim-F-at-bot[*THEN filterlim-compose*] *filterlim-real-sequentially*
filterlim-uminus-at-top[*THEN iffD1*])
(*auto simp*: *incseq-def nondecF intro*!: *diff-mono*)
also have ... = (*SUP i*. *emeasure ?F* {− *real i*<..*real i*})
by (*subst emeasure-interval-measure-Ioc*) (*simp-all add*: *nondecF right-cont-F*)
also have ... = *emeasure ?F* (⋃ *i*::*nat*. {− *real i*<..*real i*})
by (*rule SUP-emeasure-incseq*) (*auto simp*: *incseq-def*)
also have (⋃ *i*. {− *real* (*i*::*nat*)<..*real i*}) = *space ?F*

```
    by (simp add: UN-Ioc-eq-UNIV)
  finally have emeasure ?F (space ?F) = m
    by simp }
 note ∗ = this
 then show emeasure (interval-measure F) UNIV = m
  by simp

 interpret finite-measure ?F
 proof
  show emeasure ?F (space ?F) ≠ ∞
   using ∗ by simp
 qed
 show finite-borel-measure (interval-measure F)
  proof qed simp-all
qed

lemma real-distribution-interval-measure:
 fixes F :: real ⇒ real
 assumes nondecF : ⋀ x y. x ≤ y ⟹ F x ≤ F y and
  right-cont-F : ⋀a. continuous (at-right a) F and
  lim-F-at-bot : (F ⟶ 0) at-bot and
  lim-F-at-top : (F ⟶ 1) at-top
 shows real-distribution (interval-measure F)
proof −
 let ?F = interval-measure F
 interpret prob-space ?F
  proof qed (use interval-measure-UNIV[OF assms] in simp)
 show ?thesis
  proof qed simp-all
qed

lemma
 fixes F :: real ⇒ real
 assumes nondecF : ⋀ x y. x ≤ y ⟹ F x ≤ F y and
  right-cont-F : ⋀a. continuous (at-right a) F and
  lim-F-at-bot : (F ⟶ 0) at-bot
 shows emeasure-interval-measure-Iic: emeasure (interval-measure F) {.. x} = F
x
  and measure-interval-measure-Iic: measure (interval-measure F) {.. x} = F x
 unfolding cdf-def
proof −
 have F-nonneg[simp]: 0 ≤ F y for y
  using lim-F-at-bot by (rule tendsto-upperbound) (auto simp: eventually-at-bot-linorder
nondecF intro!: exI[of - y])

 have emeasure (interval-measure F) (⋃ i::nat. {−real i <.. x}) = F x − ennreal
0
 proof (intro LIMSEQ-unique[OF Lim-emeasure-incseq])
  have (λi. F x − F (− real i)) ⟶ F x − 0
```

by (*intro tendsto-intros lim-F-at-bot*[*THEN filterlim-compose*] *filterlim-real-sequentially filterlim-uminus-at-top*[*THEN iffD1*])
from *tendsto-ennrealI*[*OF this*]
show (λ*i*. *emeasure* (*interval-measure F*) {− *real i*<..*x*}) ⟶ *F x* − *ennreal 0*
apply (*rule filterlim-cong*[*THEN iffD1*, *rotated 3*])
apply *simp*
apply *simp*
apply (*rule eventually-sequentiallyI*[**where** *c*=*nat* (*ceiling* (− *x*))])
apply (*simp add*: *emeasure-interval-measure-Ioc right-cont-F nondecF*)
done
qed (*auto simp*: *incseq-def*)
also have (⋃ *i*::*nat*. {−*real i* <.. *x*}) = {..*x*}
by *auto* (*metis minus-minus neg-less-iff-less reals-Archimedean2*)
finally show *emeasure* (*interval-measure F*) {..*x*} = *F x*
by *simp*
then show *measure* (*interval-measure F*) {..*x*} = *F x*
by (*simp add*: *measure-def*)
qed

lemma *cdf-interval-measure*:
(⋀ *x y*. *x* ≤ *y* ⟹ *F x* ≤ *F y*) ⟹ (⋀*a*. *continuous* (*at-right a*) *F*) ⟹ (*F* ⟶ *0*) *at-bot* ⟹ *cdf* (*interval-measure F*) = *F*
by (*simp add*: *cdf-def fun-eq-iff measure-interval-measure-Iic*)

end

3 Weak Convergence of Functions and Distributions

Properties of weak convergence of functions and measures, including the portmanteau theorem.

theory *Weak-Convergence*
imports *Distribution-Functions*
begin

4 Weak Convergence of Functions

definition
weak-conv :: (*nat* ⇒ (*real* ⇒ *real*)) ⇒ (*real* ⇒ *real*) ⇒ *bool*
where
weak-conv F-seq F ≡ ∀ *x*. *isCont F x* ⟶ (λ*n*. *F-seq n x*) ⟶ *F x*

5 Weak Convergence of Distributions

definition

weak-conv-m :: (*nat* ⇒ *real measure*) ⇒ *real measure* ⇒ *bool*
where
weak-conv-m M-seq M ≡ *weak-conv* (λ*n*. *cdf* (*M-seq n*)) (*cdf M*)

6 Skorohod's theorem

locale *right-continuous-mono* =
fixes *f* :: *real* ⇒ *real* **and** *a b* :: *real*
assumes *cont*: ⋀*x*. *continuous* (*at-right x*) *f*
assumes *mono*: *mono f*
assumes *bot*: (*f* ⟶ *a*) *at-bot*
assumes *top*: (*f* ⟶ *b*) *at-top*
begin

abbreviation *I* :: *real* ⇒ *real* **where**
I ω ≡ *Inf* {*x*. *ω* ≤ *f x*}

lemma *pseudoinverse*: **assumes** *a* < *ω ω* < *b* **shows** *ω* ≤ *f x* ⟷ *I ω* ≤ *x*
proof
let *?F* = {*x*. *ω* ≤ *f x*}
obtain *y* **where** *f y* < *ω*
by (*metis eventually-happens' trivial-limit-at-bot-linorder order-tendstoD*(*2*) *bot* ‹*a* < *ω*›)
with *mono* **have** *bdd*: *bdd-below ?F*
by (*auto intro*!: *bdd-belowI*[*of - y*] *elim*: *mono-invE*[*OF - less-le-trans*])

have *ne*: *?F* ≠ {}
using *order-tendstoD*(*1*)[*OF top* ‹*ω* < *b*›]
by (*auto dest*!: *eventually-happens'*[*OF trivial-limit-at-top-linorder*] *intro*: *less-imp-le*)

show *ω* ≤ *f x* ⟹ *I ω* ≤ *x*
by (*auto intro*!: *cInf-lower bdd*)

{ assume ∗: *I ω* ≤ *x*
have *ω* ≤ (*INF s*∈{*x*. *ω* ≤ *f x*}. *f s*)
by (*rule cINF-greatest*[*OF ne*]) *auto*
also have ... = *f* (*I ω*)
using *continuous-at-Inf-mono*[*OF mono cont ne bdd*] **..**
also have ... ≤ *f x*
using ∗ **by** (*rule monoD*[*OF* ‹*mono f*›])
finally show *ω* ≤ *f x* **. }**
qed

lemma *pseudoinverse'*: ∀*ω*∈{*a*<..<*b*}. ∀*x*. *ω* ≤ *f x* ⟷ *I ω* ≤ *x*
by (*intro ballI allI impI pseudoinverse*) *auto*

lemma *mono-I*: *mono-on I* {*a* <..< *b*}
unfolding *mono-on-def* **by** (*metis order.trans order.refl pseudoinverse'*)

end

locale *cdf-distribution* = *real-distribution*
begin

abbreviation $C \equiv cdf\ M$

sublocale *right-continuous-mono C 0 1*
by *standard*
(*auto intro*: *cdf-nondecreasing cdf-is-right-cont cdf-lim-at-top-prob cdf-lim-at-bot monoI*)

lemma *measurable-C*[*measurable*]: $C \in$ *borel-measurable borel*
by (*intro borel-measurable-mono mono*)

lemma *measurable-CI*[*measurable*]: $I \in$ *borel-measurable* (*restrict-space borel* {*0*<..<*1*})
by (*intro borel-measurable-mono-on-fnc mono-I*)

lemma *emeasure-distr-I*: *emeasure* (*distr* (*restrict-space lborel* {*0*<..<*1*::*real*}) *borel I*) *UNIV* = *1*
by (*simp add*: *emeasure-distr space-restrict-space emeasure-restrict-space*)

lemma *distr-I-eq-M*: *distr* (*restrict-space lborel* {*0*<..<*1*::*real*}) *borel I* = *M* (**is** *?I* = -)
proof (*intro cdf-unique ext*)
let *?*Ω = *restrict-space lborel* {*0*<..<*1*}::*real measure*
interpret Ω: *prob-space ?*Ω
by (*auto simp add*: *emeasure-restrict-space space-restrict-space intro*!: *prob-spaceI*)
show *real-distribution ?I*
by *auto*

fix *x*
have *cdf ?I x* = *measure lborel* {ω∈{*0*<..<*1*}. ω ≤ *C x*}
by (*subst cdf-def*)
(*auto simp*: *pseudoinverse*[*symmetric*] *measure-distr space-restrict-space measure-restrict-space*
intro!: *arg-cong2*[**where** *f*=*measure*])
also have ... = *measure lborel* {*0* <..< *C x*}
using *cdf-bounded-prob*[*of x*] *AE-lborel-singleton*[*of C x*]
by (*auto intro*!: *arg-cong*[**where** *f*=*enn2real*] *emeasure-eq-AE simp*: *measure-def*)
also have ... = *C x*
by (*simp add*: *cdf-nonneg*)
finally show *cdf* (*distr ?*Ω *borel I*) *x* = *C x* **.**
qed *standard*

end

context
fixes μ :: *nat* ⇒ *real measure*

```
    and M :: real measure
  assumes μ: ⋀n. real-distribution (μ n)
  assumes M: real-distribution M
  assumes μ-to-M: weak-conv-m μ M
begin


theorem Skorohod:
 ∃ (Ω :: real measure) (Y-seq :: nat ⇒ real ⇒ real) (Y :: real ⇒ real).
    prob-space Ω ∧
    (∀ n. Y-seq n ∈ measurable Ω borel) ∧
    (∀ n. distr Ω borel (Y-seq n) = μ n) ∧
    Y ∈ measurable Ω lborel ∧
    distr Ω borel Y = M ∧
    (∀ x ∈ space Ω. (λn. Y-seq n x) ⟶ Y x)
proof −
  interpret μ: cdf-distribution μ n for n
    unfolding cdf-distribution-def by (rule μ)
  interpret M: cdf-distribution M
    unfolding cdf-distribution-def by (rule M)

  have conv: measure M {x} = 0 ⟹ (λn. μ.C n x) ⟶ M.C x for x
    using μ-to-M M.isCont-cdf by (auto simp: weak-conv-m-def weak-conv-def)

  let ?Ω = restrict-space lborel {0<..<1} :: real measure
  have prob-space ?Ω
    by (auto simp: space-restrict-space emeasure-restrict-space intro!: prob-spaceI)
  interpret Ω: prob-space ?Ω
    by fact

  have Y-distr: distr ?Ω borel M.I = M
    by (rule M.distr-I-eq-M)

  have Y-cts-cnv: (λn. μ.I n ω) ⟶ M.I ω
    if ω: ω ∈ {0<..<1} isCont M.I ω for ω :: real
  proof (intro limsup-le-liminf-real)
    show liminf (λn. μ.I n ω) ≥ M.I ω
      unfolding le-Liminf-iff
    proof safe
      fix B :: ereal assume B: B < M.I ω
      then show ∀F n in sequentially. B < μ.I n ω
      proof (cases B)
        case (real r)
        with B have r: r < M.I ω
          by simp
        then obtain x where x: r < x x < M.I ω measure M {x} = 0
          using open-minus-countable[OF M.countable-support, of {r<..<M.I ω}]
by auto
        then have Fx-less: M.C x < ω
```

```
        using M.pseudoinverse' ω not-less by blast

      have ∀F n in sequentially. μ.C n x < ω
        using order-tendstoD(2)[OF conv[OF x(3)] Fx-less] .
      then have ∀F n in sequentially. x < μ.I n ω
          by eventually-elim (insert ω μ.pseudoinverse[symmetric], simp add:
not-le[symmetric])
      then show ?thesis
        by eventually-elim (insert x(1), simp add: real)
    qed auto
  qed

  have ∗: limsup (λn. μ.I n ω) ≤ M.I ω'
    if ω': 0 < ω' ω' < 1 ω < ω' for ω' :: real
  proof (rule dense-ge-bounded)
    fix B' assume ereal (M.I ω') < B' B' < ereal (M.I ω' + 1)
    then obtain B where M.I ω' < B and [simp]: B' = ereal B
      by (cases B') auto
    then obtain y where y: M.I ω' < y y < B measure M {y} = 0
      using open-minus-countable[OF M.countable-support, of {M.I ω'<..<B}]
by auto
    then have ω' ≤ M.C (M.I ω')
      using M.pseudoinverse' ω' by (metis greaterThanLessThan-iff order-refl)
    also have ... ≤ M.C y
      using M.mono y unfolding mono-def by auto
    finally have Fy-gt: ω < M.C y
      using ω'(3) by simp

    have ∀F n in sequentially. ω ≤ μ.C n y
      using order-tendstoD(1)[OF conv[OF y(3)] Fy-gt] by eventually-elim (rule
less-imp-le)
    then have 2: ∀F n in sequentially. μ.I n ω ≤ ereal y
      by simp (subst μ.pseudoinverse'[rule-format, OF ω(1), symmetric])
    then show limsup (λn. μ.I n ω) ≤ B'
      using ‹y < B›
      by (intro Limsup-bounded[rotated]) (auto intro: le-less-trans elim: eventu-
ally-mono)
  qed simp

  have ∗∗: (M.I ⟶ ereal (M.I ω)) (at-right ω)
    using ω(2) by (auto intro: tendsto-within-subset simp: continuous-at)
  show limsup (λn. μ.I n ω) ≤ M.I ω
    using ω
    by (intro tendsto-lowerbound[OF ∗∗])
      (auto intro!: exI[of - 1] ∗ simp: eventually-at-right[of - 1])
qed

let ?D = {ω∈{0<..<1}. ¬ isCont M.I ω}
have D-countable: countable ?D
```

using *mono-on-ctble-discont*[*OF M.mono-I*] **by** (*simp add*: *at-within-open*[*of* - {*0 <..< 1*}] *cong*: *conj-cong*)
hence *D*: *emeasure ?Ω ?D = 0*
using *emeasure-lborel-countable*[*OF D-countable*]
by (*subst emeasure-restrict-space*) *auto*

define *Y'* **where** *Y' ω = (if ω ∈ ?D then 0 else M.I ω)* **for** *ω*
have *Y'-AE*: *AE ω in ?Ω. Y' ω = M.I ω*
by (*rule AE-I* [*OF* - *D*]) (*auto simp*: *space-restrict-space sets-restrict-space-iff Y'-def*)

define *Y-seq'* **where** *Y-seq' n ω = (if ω ∈ ?D then 0 else μ.I n ω)* **for** *n ω*
have *Y-seq'-AE*: *⋀n. AE ω in ?Ω. Y-seq' n ω = μ.I n ω*
by (*rule AE-I* [*OF* - *D*]) (*auto simp*: *space-restrict-space sets-restrict-space-iff Y-seq'-def*)

have *Y'-cnv*: *∀ω∈{0<..<1}. (λn. Y-seq' n ω) ⟶ Y' ω*
by (*auto simp*: *Y'-def Y-seq'-def Y-cts-cnv*)

have [*simp*]: *Y-seq' n ∈ borel-measurable ?Ω* **for** *n*
by (*rule measurable-discrete-difference*[*of μ.I n - - ?D*])
(*insert μ.measurable-CI*[*of n*] *D-countable*, *auto simp*: *sets-restrict-space Y-seq'-def*)
moreover have *distr ?Ω borel* (*Y-seq' n*) *= μ n* **for** *n*
using *μ.distr-I-eq-M* [*of n*] *Y-seq'-AE* [*of n*]
by (*subst distr-cong-AE*[**where** *f = Y-seq' n* **and** *g = μ.I n*], *auto*)
moreover have [*simp*]: *Y' ∈ borel-measurable ?Ω*
by (*rule measurable-discrete-difference*[*of M.I - - ?D*])
(*insert M.measurable-CI D-countable*, *auto simp*: *sets-restrict-space Y'-def*)
moreover have *distr ?Ω borel Y' = M*
using *M.distr-I-eq-M Y'-AE*
by (*subst distr-cong-AE*[**where** *f = Y'* **and** *g = M.I*], *auto*)
ultimately have *prob-space ?Ω ∧* (*∀ n. Y-seq' n ∈ borel-measurable ?Ω*) *∧*
(*∀ n. distr ?Ω borel* (*Y-seq' n*) *= μ n*) *∧ Y' ∈ measurable ?Ω lborel ∧ distr ?Ω borel Y' = M ∧*
(*∀ x∈space ?Ω. (λn. Y-seq' n x) ⟶ Y' x*)
using *Y'-cnv* ‹*prob-space ?Ω*› **by** (*auto simp*: *space-restrict-space*)
thus *?thesis* **by** *metis*
qed

The Portmanteau theorem, that is, the equivalence of various definitions of weak convergence.

theorem *weak-conv-imp-bdd-ae-continuous-conv*:
fixes
f :: *real ⇒ 'a*::{*banach*, *second-countable-topology*}
assumes
discont-null: *M* ({*x. ¬ isCont f x*}) *= 0* **and**
f-bdd: *⋀x. norm* (*f x*) *≤ B* **and**
[*measurable*]: *f ∈ borel-measurable borel*

shows
$(\lambda\ n.\ integral^L\ (\mu\ n)\ f) \longrightarrow integral^L\ M\ f$
proof −
have $0 \leq B$
using *norm-ge-zero f-bdd* **by** (*rule order-trans*)
note *Skorohod*
then obtain *Omega Y-seq Y* **where**
ps-Omega [*simp*]: *prob-space Omega* **and**
Y-seq-measurable [*measurable*]: $\bigwedge n.$ *Y-seq n* $\in$ *borel-measurable Omega* **and**
distr-Y-seq: $\bigwedge n.$ *distr Omega borel* (*Y-seq n*) $= \mu\ n$ **and**
Y-measurable [*measurable*]: *Y* $\in$ *borel-measurable Omega* **and**
distr-Y: *distr Omega borel Y* $= M$ **and**
YnY: $\bigwedge x ::$ *real. x* $\in$ *space Omega* $\Longrightarrow$ $(\lambda n.$ *Y-seq n x*$) \longrightarrow Y\ x$ **by** *force*
interpret *prob-space Omega* **by** *fact*
have $*$: *emeasure Omega* (*Y* − ‘ $\{x. \neg$ *isCont f x*$\}$ $\cap$ *space Omega*$) = 0$
by (*subst emeasure-distr* [*symmetric*, **where** *N*=*borel*]) (*auto simp*: *distr-Y discont-null*)
have $*$: *AE x in Omega.* $(\lambda n.\ f\ (Y\text{-}seq\ n\ x)) \longrightarrow f\ (Y\ x)$
by (*rule AE-I* [*OF* - $*$]) (*auto intro*: *isCont-tendsto-compose YnY*)
show *?thesis*
by (*auto intro*!: *integral-dominated-convergence*[**where** *w*=$\lambda x.$ *B*]
simp: *f-bdd* $*$ *integral-distr distr-Y-seq* [*symmetric*] *distr-Y* [*symmetric*])
qed

theorem *weak-conv-imp-integral-bdd-continuous-conv*:
fixes *f* :: *real* $\Rightarrow$ *'a*::{*banach*, *second-countable-topology*}
assumes
$\bigwedge x.$ *isCont f x* **and**
$\bigwedge x.$ *norm* $(f\ x) \leq B$
shows
$(\lambda\ n.\ integral^L\ (\mu\ n)\ f) \longrightarrow integral^L\ M\ f$
using *assms*
by (*intro weak-conv-imp-bdd-ae-continuous-conv*)
(*auto intro*!: *borel-measurable-continuous-onI continuous-at-imp-continuous-on*)

theorem *weak-conv-imp-continuity-set-conv*:
fixes *f* :: *real* $\Rightarrow$ *real*
assumes [*measurable*]: *A* $\in$ *sets borel* **and** *M* (*frontier A*) $= 0$
shows $(\lambda n.$ *measure* $(\mu\ n)\ A) \longrightarrow$ *measure M A*
proof −
interpret *M*: *real-distribution M* **by** *fact*
interpret μ: *real-distribution* $\mu\ n$ **for** *n* **by** *fact*

have $(\lambda n.\ (\int x.\ indicator\ A\ x\ \partial\mu\ n) :: real) \longrightarrow (\int x.\ indicator\ A\ x\ \partial M)$
by (*intro weak-conv-imp-bdd-ae-continuous-conv*[**where** *B*=*1*])
(*auto intro*: *assms simp*: *isCont-indicator*)
then show *?thesis*
by *simp*
qed

end

definition
cts-step :: *real* ⇒ *real* ⇒ *real* ⇒ *real*
where
cts-step a b x ≡ *if x* ≤ *a then 1 else if x* ≥ *b then 0 else* (*b* − *x*) / (*b* − *a*)

lemma *cts-step-uniformly-continuous*:
assumes [*arith*]: *a* < *b*
shows *uniformly-continuous-on UNIV* (*cts-step a b*)
unfolding *uniformly-continuous-on-def*
proof *clarsimp*
fix *e* :: *real* **assume** [*arith*]: *0* < *e*
let *?d* = *min* (*e* ∗ (*b* − *a*)) (*b* − *a*)
have *?d* > *0*
by (*auto simp add*: *field-simps*)
moreover have *dist x′ x* < *?d* ⟹ *dist* (*cts-step a b x′*) (*cts-step a b x*) < *e* **for** *x x′*
by (*auto simp*: *dist-real-def divide-simps cts-step-def*)
ultimately show ∃ *d* > *0*. ∀ *x x′*. *dist x′ x* < *d* ⟶ *dist* (*cts-step a b x′*) (*cts-step a b x*) < *e*
by *blast*
qed

lemma (**in** *real-distribution*) *integrable-cts-step*: *a* < *b* ⟹ *integrable M* (*cts-step a b*)
by (*rule integrable-const-bound* [*of - 1*]) (*auto simp*: *cts-step-def*[*abs-def*])

lemma (**in** *real-distribution*) *cdf-cts-step*:
assumes [*arith*]: *x* < *y*
shows *cdf M x* ≤ *integral*^{L} *M* (*cts-step x y*) **and** *integral*^{L} *M* (*cts-step x y*) ≤ *cdf M y*
proof −
have *cdf M x* = *integral*^{L} *M* (*indicator* {*..x*})
by (*simp add*: *cdf-def*)
also have ... ≤ *expectation* (*cts-step x y*)
by (*intro integral-mono integrable-cts-step*)
(*auto simp*: *cts-step-def less-top*[*symmetric*] *split*: *split-indicator*)
finally show *cdf M x* ≤ *expectation* (*cts-step x y*) **.**
next
have *expectation* (*cts-step x y*) ≤ *integral*^{L} *M* (*indicator* {*..y*})
by (*intro integral-mono integrable-cts-step*)
(*auto simp*: *cts-step-def less-top*[*symmetric*] *split*: *split-indicator*)
also have ... = *cdf M y*
by (*simp add*: *cdf-def*)
finally show *expectation* (*cts-step x y*) ≤ *cdf M y* **.**
qed

context
fixes *M-seq* :: *nat* ⇒ *real measure*
and *M* :: *real measure*
assumes *distr-M-seq* [*simp*]: ⋀*n*. *real-distribution* (*M-seq n*)
assumes *distr-M* [*simp*]: *real-distribution M*
begin

theorem *continuity-set-conv-imp-weak-conv*:
fixes *f* :: *real* ⇒ *real*
assumes ∗: ⋀*A*. *A* ∈ *sets borel* ⟹ *M* (*frontier A*) = *0* ⟹ (λ *n*. (*measure* (*M-seq n*) *A*)) ⟶ *measure M A*
shows *weak-conv-m M-seq M*
proof −
interpret *real-distribution M* **by** *simp*
show *?thesis*
by (*auto intro*!: ∗ *simp*: *frontier-real-atMost isCont-cdf emeasure-eq-measure weak-conv-m-def weak-conv-def cdf-def2*)
qed

theorem *integral-cts-step-conv-imp-weak-conv*:
assumes *integral-conv*: ⋀*x y*. *x* < *y* ⟹ (λ*n*. *integral*^{L} (*M-seq n*) (*cts-step x y*)) ⟶ *integral*^{L} *M* (*cts-step x y*)
shows *weak-conv-m M-seq M*
unfolding *weak-conv-m-def weak-conv-def*
proof (*clarsimp*)
interpret *real-distribution M* **by** (*rule distr-M*)
fix *x* **assume** *isCont* (*cdf M*) *x*
hence *left-cont*: *continuous* (*at-left x*) (*cdf M*)
unfolding *continuous-at-split* **..**
{ fix *y* :: *real* **assume** [*arith*]: *x* < *y*
have *limsup* (λ*n*. *cdf* (*M-seq n*) *x*) ≤ *limsup* (λ*n*. *integral*^{L} (*M-seq n*) (*cts-step x y*))
by (*auto intro*!: *Limsup-mono always-eventually real-distribution.cdf-cts-step*)
also have ... = *integral*^{L} *M* (*cts-step x y*)
by (*intro lim-imp-Limsup*) (*auto intro*: *integral-conv*)
also have ... ≤ *cdf M y*
by (*simp add*: *cdf-cts-step*)
finally have *limsup* (λ*n*. *cdf* (*M-seq n*) *x*) ≤ *cdf M y* **.**
} note ∗ = *this*
{ fix *y* :: *real* **assume** [*arith*]: *x* > *y*
have *cdf M y* ≤ *ereal* (*integral*^{L} *M* (*cts-step y x*))
by (*simp add*: *cdf-cts-step*)
also have ... = *liminf* (λ*n*. *integral*^{L} (*M-seq n*) (*cts-step y x*))
by (*intro lim-imp-Liminf*[*symmetric*]) (*auto intro*: *integral-conv*)
also have ... ≤ *liminf* (λ*n*. *cdf* (*M-seq n*) *x*)
by (*auto intro*!: *Liminf-mono always-eventually real-distribution.cdf-cts-step*)
finally have *liminf* (λ*n*. *cdf* (*M-seq n*) *x*) ≥ *cdf M y* **.**
} note ∗∗ = *this*

have *limsup* (λn. *cdf* (*M-seq n*) *x*) $\leq$ *cdf M x*
proof (*rule tendsto-lowerbound*)
show $\forall_F$ *i in at-right x*. *limsup* (λxa. *ereal* (*cdf* (*M-seq xa*) *x*)) $\leq$ *ereal* (*cdf M i*)
by (*subst eventually-at-right*[*of - x + 1*]) (*auto simp*: $*$ *intro*: *exI* [*of - x+1*])
qed (*insert cdf-is-right-cont*, *auto simp*: *continuous-within*)
moreover have *cdf M x* $\leq$ *liminf* (λn. *cdf* (*M-seq n*) *x*)
proof (*rule tendsto-upperbound*)
show $\forall_F$ *i in at-left x*. *ereal* (*cdf M i*) $\leq$ *liminf* (λxa. *ereal* (*cdf* (*M-seq xa*) *x*))
by (*subst eventually-at-left*[*of x − 1*]) (*auto simp*: $**$ *intro*: *exI* [*of - x−1*])
qed (*insert left-cont*, *auto simp*: *continuous-within*)
ultimately show (λn. *cdf* (*M-seq n*) *x*) $\longrightarrow$ *cdf M x*
by (*elim limsup-le-liminf-real*)
qed

theorem *integral-bdd-continuous-conv-imp-weak-conv*:
assumes
$\bigwedge f$. ($\bigwedge x$. *isCont f x*) $\Longrightarrow$ ($\bigwedge x$. *abs* (*f x*) $\leq$ *1*) $\Longrightarrow$ (λn. *integral*^{L} (*M-seq n*) *f*::*real*) $\longrightarrow$ *integral*^{L} *M f*
shows
weak-conv-m M-seq M
apply (*rule integral-cts-step-conv-imp-weak-conv* [*OF assms*])
apply (*rule continuous-on-interior*)
apply (*rule uniformly-continuous-imp-continuous*)
apply (*rule cts-step-uniformly-continuous*)
apply (*auto simp*: *cts-step-def*)
done

end

end

7 The Giry monad

theory *Giry-Monad*
imports *Probability-Measure HOL−Library.Monad-Syntax*
begin

7.1 Sub-probability spaces

locale *subprob-space* = *finite-measure* +
assumes *emeasure-space-le-1*: *emeasure M* (*space M*) $\leq$ *1*
assumes *subprob-not-empty*: *space M* $\neq$ {}

lemma *subprob-spaceI*[*Pure.intro*!]:
assumes $*$: *emeasure M* (*space M*) $\leq$ *1*
assumes *space M* $\neq$ {}
shows *subprob-space M*
proof −

interpret *finite-measure M*
proof
show *emeasure M* (*space M*) ≠ ∞ **using** ∗ **by** (*auto simp*: *top-unique*)
qed
show *subprob-space M* **by** *standard fact+*
qed

lemma (**in** *subprob-space*) *emeasure-subprob-space-less-top*: *emeasure M A* ≠ *top*
using *emeasure-finite*[*of A*] **.**

lemma *prob-space-imp-subprob-space*:
prob-space M ⟹ *subprob-space M*
by (*rule subprob-spaceI*) (*simp-all add*: *prob-space.emeasure-space-1 prob-space.not-empty*)

lemma *subprob-space-imp-sigma-finite*: *subprob-space M* ⟹ *sigma-finite-measure M*
unfolding *subprob-space-def finite-measure-def* **by** *simp*

sublocale *prob-space* ⊆ *subprob-space*
by (*rule subprob-spaceI*) (*simp-all add*: *emeasure-space-1 not-empty*)

lemma *subprob-space-sigma* [*simp*]: Ω ≠ {} ⟹ *subprob-space* (*sigma* Ω *X*)
by(*rule subprob-spaceI*)(*simp-all add*: *emeasure-sigma space-measure-of-conv*)

lemma *subprob-space-null-measure*: *space M* ≠ {} ⟹ *subprob-space* (*null-measure M*)
by(*simp add*: *null-measure-def*)

lemma (**in** *subprob-space*) *subprob-space-distr*:
assumes *f*: *f* ∈ *measurable M M'* **and** *space M'* ≠ {} **shows** *subprob-space* (*distr M M' f*)
proof (*rule subprob-spaceI*)
have *f* −‘ *space M'* ∩ *space M* = *space M* **using** *f* **by** (*auto dest*: *measurable-space*)
with *f* **show** *emeasure* (*distr M M' f*) (*space* (*distr M M' f*)) ≤ *1*
by (*auto simp*: *emeasure-distr emeasure-space-le-1*)
show *space* (*distr M M' f*) ≠ {} **by** (*simp add*: *assms*)
qed

lemma (**in** *subprob-space*) *subprob-emeasure-le-1*: *emeasure M X* ≤ *1*
by (*rule order.trans*[*OF emeasure-space emeasure-space-le-1*])

lemma (**in** *subprob-space*) *subprob-measure-le-1*: *measure M X* ≤ *1*
using *subprob-emeasure-le-1*[*of X*] **by** (*simp add*: *emeasure-eq-measure*)

lemma (**in** *subprob-space*) *nn-integral-le-const*:
assumes *0* ≤ *c AE x in M*. *f x* ≤ *c*
shows (∫ $^+$*x*. *f x* ∂*M*) ≤ *c*
proof −

```
  have (∫⁺ x. f x ∂M) ≤ (∫⁺ x. c ∂M)
    by(rule nn-integral-mono-AE) fact
  also have ... ≤ c * emeasure M (space M)
    using ‹0 ≤ c› by simp
 also have ... ≤ c * 1 using emeasure-space-le-1 ‹0 ≤ c› by(rule mult-left-mono)
  finally show ?thesis by simp
qed

lemma emeasure-density-distr-interval:
  fixes h :: real ⇒ real and g :: real ⇒ real and g' :: real ⇒ real
  assumes [simp]: a ≤ b
  assumes Mf[measurable]: f ∈ borel-measurable borel
  assumes Mg[measurable]: g ∈ borel-measurable borel
  assumes Mg'[measurable]: g' ∈ borel-measurable borel
  assumes Mh[measurable]: h ∈ borel-measurable borel
  assumes prob: subprob-space (density lborel f)
  assumes nonnegf: ⋀x. f x ≥ 0
  assumes derivg: ⋀x. x ∈ {a..b} ⟹ (g has-real-derivative g' x) (at x)
  assumes contg': continuous-on {a..b} g'
  assumes mono: strict-mono-on g {a..b} and inv: ⋀x. h x ∈ {a..b} ⟹ g (h x)
= x
  assumes range: {a..b} ⊆ range h
  shows emeasure (distr (density lborel f) lborel h) {a..b} =
            emeasure (density lborel (λx. f (g x) * g' x)) {a..b}
proof (cases a < b)
  assume a < b
  from mono have inj: inj-on g {a..b} by (rule strict-mono-on-imp-inj-on)
 from mono have mono': mono-on g {a..b} by (rule strict-mono-on-imp-mono-on)
  from mono' derivg have ⋀x. x ∈ {a<..<b} ⟹ g' x ≥ 0
    by (rule mono-on-imp-deriv-nonneg) auto
  from contg' this have derivg-nonneg: ⋀x. x ∈ {a..b} ⟹ g' x ≥ 0
    by (rule continuous-ge-on-Ioo) (simp-all add: ‹a < b›)

 from derivg have contg: continuous-on {a..b} g by (rule has-real-derivative-imp-continuous-on)
  have A: h -` {a..b} = {g a..g b}
  proof (intro equalityI subsetI)
    fix x assume x: x ∈ h -` {a..b}
    hence g (h x) ∈ {g a..g b} by (auto intro: mono-onD[OF mono'])
    with inv and x show x ∈ {g a..g b} by simp
  next
    fix y assume y: y ∈ {g a..g b}
    with IVT'[OF - - - contg, of y] obtain x where x ∈ {a..b} y = g x by auto
    with range and inv show y ∈ h -` {a..b} by auto
  qed

  have prob': subprob-space (distr (density lborel f) lborel h)
    by (rule subprob-space.subprob-space-distr[OF prob]) (simp-all add: Mh)
  have B: emeasure (distr (density lborel f) lborel h) {a..b} =
           ∫⁺x. f x * indicator (h -` {a..b}) x ∂lborel
```

by (*subst emeasure-distr*) (*simp-all add*: *emeasure-density Mf Mh measurable-sets-borel*[*OF Mh*])
also note *A*
also have *emeasure* (*distr* (*density lborel f*) *lborel h*) {*a*..*b*} ≤ *1*
by (*rule subprob-space.subprob-emeasure-le-1*) (*rule prob'*)
hence *emeasure* (*distr* (*density lborel f*) *lborel h*) {*a*..*b*} ≠ ∞ **by** (*auto simp*: *top-unique*)
with *assms* **have** (∫⁺*x*. *f x* ∗ *indicator* {*g a*..*g b*} *x* ∂*lborel*) =
(∫⁺*x*. *f* (*g x*) ∗ *g'* *x* ∗ *indicator* {*a*..*b*} *x* ∂*lborel*)
by (*intro nn-integral-substitution-aux*)
(*auto simp*: *derivg-nonneg A B emeasure-density mult.commute* ‹*a* < *b*›)
also have ... = *emeasure* (*density lborel* (λ*x*. *f* (*g x*) ∗ *g'* *x*)) {*a*..*b*}
by (*simp add*: *emeasure-density*)
finally show *?thesis* **.**
next
assume ¬*a* < *b*
with ‹*a* ≤ *b*› **have** [*simp*]: *b* = *a* **by** (*simp add*: *not-less del*: ‹*a* ≤ *b*›)
from *inv* **and** *range* **have** *h* −‘ {*a*} = {*g a*} **by** *auto*
thus *?thesis* **by** (*simp-all add*: *emeasure-distr emeasure-density measurable-sets-borel*[*OF Mh*])
qed

locale *pair-subprob-space* =
pair-sigma-finite M1 M2 + *M1*: *subprob-space M1* + *M2*: *subprob-space M2* **for** *M1 M2*

sublocale *pair-subprob-space* ⊆ *P?*: *subprob-space M1* ⨂$_M$ *M2*
proof
from *mult-le-one*[*OF M1.emeasure-space-le-1 - M2.emeasure-space-le-1*]
show *emeasure* (*M1* ⨂$_M$ *M2*) (*space* (*M1* ⨂$_M$ *M2*)) ≤ *1*
by (*simp add*: *M2.emeasure-pair-measure-Times space-pair-measure*)
from *M1.subprob-not-empty* **and** *M2.subprob-not-empty* **show** *space* (*M1* ⨂$_M$ *M2*) ≠ {}
by (*simp add*: *space-pair-measure*)
qed

lemma *subprob-space-null-measure-iff*:
subprob-space (*null-measure M*) ⟷ *space M* ≠ {}
by (*auto intro*!: *subprob-spaceI dest*: *subprob-space.subprob-not-empty*)

lemma *subprob-space-restrict-space*:
assumes *M*: *subprob-space M*
and *A*: *A* ∩ *space M* ∈ *sets M A* ∩ *space M* ≠ {}
shows *subprob-space* (*restrict-space M A*)
proof(*rule subprob-spaceI*)
have *emeasure* (*restrict-space M A*) (*space* (*restrict-space M A*)) = *emeasure M* (*A* ∩ *space M*)
using *A* **by**(*simp add*: *emeasure-restrict-space space-restrict-space*)
also have ... ≤ *1* **by**(*rule subprob-space.subprob-emeasure-le-1*)(*rule M*)

finally show *emeasure* (*restrict-space M A*) (*space* (*restrict-space M A*)) ≤ *1* **.**
next
show *space* (*restrict-space M A*) ≠ {}
using *A* **by**(*simp add*: *space-restrict-space*)
qed

definition *subprob-algebra* :: *'a measure* ⇒ *'a measure measure* **where**
subprob-algebra K =
(*SUP A* ∈ *sets K*. *vimage-algebra* {*M*. *subprob-space M* ∧ *sets M* = *sets K*}
(λ*M*. *emeasure M A*) *borel*)

lemma *space-subprob-algebra*: *space* (*subprob-algebra A*) = {*M*. *subprob-space M*
∧ *sets M* = *sets A*}
by (*auto simp add*: *subprob-algebra-def space-Sup-eq-UN*)

lemma *subprob-algebra-cong*: *sets M* = *sets N* ⟹ *subprob-algebra M* = *subprob-algebra N*
by (*simp add*: *subprob-algebra-def*)

lemma *measurable-emeasure-subprob-algebra*[*measurable*]:
a ∈ *sets A* ⟹ (λ*M*. *emeasure M a*) ∈ *borel-measurable* (*subprob-algebra A*)
by (*auto intro*!: *measurable-Sup1 measurable-vimage-algebra1 simp*: *subprob-algebra-def*)

lemma *measurable-measure-subprob-algebra*[*measurable*]:
a ∈ *sets A* ⟹ (λ*M*. *measure M a*) ∈ *borel-measurable* (*subprob-algebra A*)
unfolding *measure-def* **by** *measurable*

lemma *subprob-measurableD*:
assumes *N*: *N* ∈ *measurable M* (*subprob-algebra S*) **and** *x*: *x* ∈ *space M*
shows *space* (*N x*) = *space S*
and *sets* (*N x*) = *sets S*
and *measurable* (*N x*) *K* = *measurable S K*
and *measurable K* (*N x*) = *measurable K S*
using *measurable-space*[*OF N x*]
by (*auto simp*: *space-subprob-algebra intro*!: *measurable-cong-sets dest*: *sets-eq-imp-space-eq*)

ML ‹

fun subprob-cong thm ctxt = (
let
val thm' = *Thm.transfer' ctxt thm*
val free = *thm'* |> *Thm.concl-of* |> *HOLogic.dest-Trueprop* |> *dest-comb* |> *fst*
|>
dest-comb |> *snd* |> *strip-abs-body* |> *head-of* |> *is-Free*
in
if free then ([], *Measurable.add-local-cong* (*thm' RS* @{*thm subprob-measurableD*(*2*)})
ctxt)
else ([], *ctxt*)
end

handle THM - => ([], ctxt) | TERM - => ([], ctxt))

›

setup ‹
Context.theory-map (Measurable.add-preprocessor subprob-cong subprob-cong)
›

context
fixes *K M N* **assumes** *K*: *K ∈ measurable M (subprob-algebra N)*
begin

lemma *subprob-space-kernel*: *a ∈ space M ⟹ subprob-space (K a)*
using *measurable-space*[*OF K*] **by** (*simp add*: *space-subprob-algebra*)

lemma *sets-kernel*: *a ∈ space M ⟹ sets (K a) = sets N*
using *measurable-space*[*OF K*] **by** (*simp add*: *space-subprob-algebra*)

lemma *measurable-emeasure-kernel*[*measurable*]:
A ∈ sets N ⟹ (λa. emeasure (K a) A) ∈ borel-measurable M
using *measurable-compose*[*OF K measurable-emeasure-subprob-algebra*] **.**

end

lemma *measurable-subprob-algebra*:
(⋀a. a ∈ space M ⟹ subprob-space (K a)) ⟹
(⋀a. a ∈ space M ⟹ sets (K a) = sets N) ⟹
(⋀A. A ∈ sets N ⟹ (λa. emeasure (K a) A) ∈ borel-measurable M) ⟹
K ∈ measurable M (subprob-algebra N)
by (*auto intro*!: *measurable-Sup2 measurable-vimage-algebra2 simp*: *subprob-algebra-def*)

lemma *measurable-submarkov*:
K ∈ measurable M (subprob-algebra M) ⟷
(∀ x∈space M. subprob-space (K x) ∧ sets (K x) = sets M) ∧
(∀ A∈sets M. (λx. emeasure (K x) A) ∈ measurable M borel)
proof
assume *(∀ x∈space M. subprob-space (K x) ∧ sets (K x) = sets M) ∧*
(∀ A∈sets M. (λx. emeasure (K x) A) ∈ borel-measurable M)
then show *K ∈ measurable M (subprob-algebra M)*
by (*intro measurable-subprob-algebra*) *auto*
next
assume *K ∈ measurable M (subprob-algebra M)*
then show *(∀ x∈space M. subprob-space (K x) ∧ sets (K x) = sets M) ∧*
(∀ A∈sets M. (λx. emeasure (K x) A) ∈ borel-measurable M)
by (*auto dest*: *subprob-space-kernel sets-kernel*)
qed

lemma *measurable-subprob-algebra-generated*:
assumes *eq*: *sets N = sigma-sets Ω G* **and** *Int-stable G G ⊆ Pow Ω*

assumes *subsp*: $\bigwedge a.\ a \in space\ M \Longrightarrow subprob\text{-}space\ (K\ a)$
assumes *sets*: $\bigwedge a.\ a \in space\ M \Longrightarrow sets\ (K\ a) = sets\ N$
assumes $\bigwedge A.\ A \in G \Longrightarrow (\lambda a.\ emeasure\ (K\ a)\ A) \in borel\text{-}measurable\ M$
assumes Ω: $(\lambda a.\ emeasure\ (K\ a)\ \Omega) \in borel\text{-}measurable\ M$
shows $K \in measurable\ M\ (subprob\text{-}algebra\ N)$
proof (*rule measurable-subprob-algebra*)
fix *a* **assume** $a \in space\ M$ **then show** *subprob-space* $(K\ a)$ *sets* $(K\ a) = sets\ N$
by *fact+*
next
interpret *G*: *sigma-algebra* Ω *sigma-sets* $\Omega\ G$
using ‹$G \subseteq Pow\ \Omega$› **by** (*rule sigma-algebra-sigma-sets*)
fix *A* **assume** $A \in sets\ N$ **with** *assms*(*2*,*3*) **show** $(\lambda a.\ emeasure\ (K\ a)\ A) \in$ *borel-measurable M*
unfolding ‹*sets N* = *sigma-sets* $\Omega\ G$›
proof (*induction rule*: *sigma-sets-induct-disjoint*)
case (*basic A*) **then show** *?case* **by** *fact*
next
case *empty* **then show** *?case* **by** *simp*
next
case (*compl A*)
have $(\lambda a.\ emeasure\ (K\ a)\ (\Omega - A)) \in borel\text{-}measurable\ M \longleftrightarrow$
$(\lambda a.\ emeasure\ (K\ a)\ \Omega - emeasure\ (K\ a)\ A) \in borel\text{-}measurable\ M$
using *G.top G.sets-into-space sets eq compl subprob-space.emeasure-subprob-space-less-top*[*OF subsp*]
by (*intro measurable-cong emeasure-Diff*) *auto*
with *compl* Ω **show** *?case*
by *simp*
next
case (*union F*)
moreover have $(\lambda a.\ emeasure\ (K\ a)\ (\bigcup i.\ F\ i)) \in borel\text{-}measurable\ M \longleftrightarrow$
$(\lambda a.\ \sum i.\ emeasure\ (K\ a)\ (F\ i)) \in borel\text{-}measurable\ M$
using *sets union eq*
by (*intro measurable-cong suminf-emeasure*[*symmetric*]) *auto*
ultimately show *?case*
by *auto*
qed
qed

lemma *space-subprob-algebra-empty-iff*:
$space\ (subprob\text{-}algebra\ N) = \{\} \longleftrightarrow space\ N = \{\}$
proof
have $\bigwedge x.\ x \in space\ N \Longrightarrow density\ N\ (\lambda\text{-}.\ 0) \in space\ (subprob\text{-}algebra\ N)$
by (*auto simp*: *space-subprob-algebra emeasure-density intro*!: *subprob-spaceI*)
then show $space\ (subprob\text{-}algebra\ N) = \{\} \Longrightarrow space\ N = \{\}$
by *auto*
next
assume $space\ N = \{\}$
hence $sets\ N = \{\{\}\}$ **by** (*simp add*: *space-empty-iff*)
moreover have $\bigwedge M.\ subprob\text{-}space\ M \Longrightarrow sets\ M \neq \{\{\}\}$

by (*simp add*: *subprob-space.subprob-not-empty space-empty-iff*[*symmetric*])
ultimately show *space* (*subprob-algebra N*) = {} **by** (*auto simp*: *space-subprob-algebra*)
qed

lemma *nn-integral-measurable-subprob-algebra*[*measurable*]:
assumes *f*: *f* ∈ *borel-measurable N*
shows (λ*M*. *integral*^{N} *M f*) ∈ *borel-measurable* (*subprob-algebra N*) (**is** - ∈ *?B*)
using *f*
proof *induct*
case (*cong f g*)
moreover have (λ*M'*. ∫$^+$*M''*. *f M'' ∂M'*) ∈ *?B* ⟷ (λ*M'*. ∫$^+$*M''*. *g M'' ∂M'*) ∈ *?B*
by (*intro measurable-cong nn-integral-cong cong*)
(*auto simp*: *space-subprob-algebra dest*!: *sets-eq-imp-space-eq*)
ultimately show *?case* **by** *simp*
next
case (*set B*)
then have (λ*M'*. ∫$^+$*M''*. *indicator B M'' ∂M'*) ∈ *?B* ⟷ (λ*M'*. *emeasure M' B*) ∈ *?B*
by (*intro measurable-cong nn-integral-indicator*) (*simp add*: *space-subprob-algebra*)
with *set* **show** *?case*
by (*simp add*: *measurable-emeasure-subprob-algebra*)
next
case (*mult f c*)
then have (λ*M'*. ∫$^+$*M''*. *c* ∗ *f M'' ∂M'*) ∈ *?B* ⟷ (λ*M'*. *c* ∗ ∫$^+$*M''*. *f M'' ∂M'*) ∈ *?B*
by (*intro measurable-cong nn-integral-cmult*) (*auto simp add*: *space-subprob-algebra*)
with *mult* **show** *?case*
by *simp*
next
case (*add f g*)
then have (λ*M'*. ∫$^+$*M''*. *f M'' + g M'' ∂M'*) ∈ *?B* ⟷ (λ*M'*. (∫$^+$*M''*. *f M'' ∂M'*) + (∫$^+$*M''*. *g M'' ∂M'*)) ∈ *?B*
by (*intro measurable-cong nn-integral-add*) (*auto simp add*: *space-subprob-algebra*)
with *add* **show** *?case*
by (*simp add*: *ac-simps*)
next
case (*seq F*)
then have (λ*M'*. ∫$^+$*M''*. (*SUP i*. *F i*) *M'' ∂M'*) ∈ *?B* ⟷ (λ*M'*. *SUP i*. (∫$^+$*M''*. *F i M'' ∂M'*)) ∈ *?B*
unfolding *SUP-apply*
by (*intro measurable-cong nn-integral-monotone-convergence-SUP*) (*auto simp add*: *space-subprob-algebra*)
with *seq* **show** *?case*
by (*simp add*: *ac-simps*)
qed

lemma *measurable-distr*:
assumes [*measurable*]: *f* ∈ *measurable M N*

shows (*λM′. distr M′ N f*) ∈ *measurable* (*subprob-algebra M*) (*subprob-algebra N*)
proof (*cases space N = {}*)
assume *not-empty*: *space N ≠ {}*
show *?thesis*
proof (*rule measurable-subprob-algebra*)
fix *A* **assume** *A*: *A ∈ sets N*
then have (*λM′. emeasure* (*distr M′ N f*) *A*) ∈ *borel-measurable* (*subprob-algebra M*) ⟷
(*λM′. emeasure M′* (*f −‘ A ∩ space M*)) ∈ *borel-measurable* (*subprob-algebra M*)
by (*intro measurable-cong*)
(*auto simp*: *emeasure-distr space-subprob-algebra*
intro!: *arg-cong2*[**where** *f*=*emeasure*] *sets-eq-imp-space-eq arg-cong2*[**where** *f*=(∩)])
also have . . .
using *A* **by** (*intro measurable-emeasure-subprob-algebra*) *simp*
finally show (*λM′. emeasure* (*distr M′ N f*) *A*) ∈ *borel-measurable* (*subprob-algebra M*) **.**
qed (*auto intro*!: *subprob-space.subprob-space-distr simp*: *space-subprob-algebra not-empty cong*: *measurable-cong-sets*)
qed (*insert assms*, *auto simp*: *measurable-empty-iff space-subprob-algebra-empty-iff*)

lemma *emeasure-space-subprob-algebra*[*measurable*]:
(*λa. emeasure a* (*space a*)) ∈ *borel-measurable* (*subprob-algebra N*)
proof−
have (*λa. emeasure a* (*space N*)) ∈ *borel-measurable* (*subprob-algebra N*) (**is** *?f* ∈ *?M*)
by (*rule measurable-emeasure-subprob-algebra*) *simp*
also have *?f* ∈ *?M* ⟷ (*λa. emeasure a* (*space a*)) ∈ *?M*
by (*rule measurable-cong*) (*auto simp*: *space-subprob-algebra dest*: *sets-eq-imp-space-eq*)
finally show *?thesis* **.**
qed

lemma *integrable-measurable-subprob-algebra*[*measurable*]:
fixes *f* :: *′a* ⇒ *′b*::{*banach*, *second-countable-topology*}
assumes [*measurable*]: *f ∈ borel-measurable N*
shows *Measurable.pred* (*subprob-algebra N*) (*λM. integrable M f*)
proof (*rule measurable-cong*[*THEN iffD2*])
show *M ∈ space* (*subprob-algebra N*) ⟹ *integrable M f* ⟷ (∫ ⁺*x. norm* (*f x*) ∂*M*) < ∞ **for** *M*
by (*auto simp*: *space-subprob-algebra integrable-iff-bounded*)
qed *measurable*

lemma *integral-measurable-subprob-algebra*[*measurable*]:
fixes *f* :: *′a* ⇒ *′b*::{*banach*, *second-countable-topology*}
assumes *f* [*measurable*]: *f ∈ borel-measurable N*
shows (*λM. integral*^{L} *M f*) ∈ *subprob-algebra N* →$_M$ *borel*
proof −

from *borel-measurable-implies-sequence-metric*[*OF f, of 0*]
obtain *F* **where** *F*: $\bigwedge i$. *simple-function N* (*F i*)
$\bigwedge x.\ x \in$ *space N* $\Longrightarrow$ ($\lambda i.$ *F i x*) $\longrightarrow$ *f x*
$\bigwedge i\ x.\ x \in$ *space N* $\Longrightarrow$ *norm* (*F i x*) $\leq$ *2* $*$ *norm* (*f x*)
unfolding *norm-conv-dist* **by** *blast*

have [*measurable*]: *F i* $\in$ *N* $\rightarrow_M$ *count-space UNIV* **for** *i*
using *F*(*1*) **by** (*rule measurable-simple-function*)

define *F'* **where** [*abs-def*]:
F' M i = (*if integrable M f then integral*^{L} *M* (*F i*) *else 0*) **for** *M i*

have ($\lambda M.$ *F' M i*) $\in$ *subprob-algebra N* $\rightarrow_M$ *borel* **for** *i*
proof (*rule measurable-cong*[*THEN iffD2*])
fix *M* **assume** *M* $\in$ *space* (*subprob-algebra N*)
then have [*simp*]: *sets M* = *sets N space M* = *space N subprob-space M*
by (*auto simp*: *space-subprob-algebra intro*!: *sets-eq-imp-space-eq*)
interpret *subprob-space M* **by** *fact*
have *F' M i* = (*if integrable M f then Bochner-Integration.simple-bochner-integral M* (*F i*) *else 0*)
using *F*(*1*)
by (*subst simple-bochner-integrable-eq-integral*)
(*auto simp*: *simple-bochner-integrable.simps simple-function-def F'-def*)
then show *F' M i* = (*if integrable M f then* $\sum y \in F\ i$ ‘ *space N. measure M* {*x*$\in$*space N. F i x* = *y*} $*_R$ *y else 0*)
unfolding *simple-bochner-integral-def* **by** *simp*
qed *measurable*
moreover
have *F' M* $\longrightarrow$ *integral*^{L} *M f* **if** *M*: *M* $\in$ *space* (*subprob-algebra N*) **for** *M*
proof *cases*
from *M* **have** [*simp*]: *sets M* = *sets N space M* = *space N*
by (*auto simp*: *space-subprob-algebra intro*!: *sets-eq-imp-space-eq*)
assume *integrable M f* **then show** *?thesis*
unfolding *F'-def* **using** *F*(*1*)[*THEN borel-measurable-simple-function*] *F*
by (*auto intro*!: *integral-dominated-convergence*[**where** *w*=$\lambda x.$ *2* $*$ *norm* (*f x*)]
cong: *measurable-cong-sets*)
qed (*auto simp*: *F'-def not-integrable-integral-eq*)
ultimately show *?thesis*
by (*rule borel-measurable-LIMSEQ-metric*)
qed

lemma *measurable-pair-measure*:
assumes *f*: *f* $\in$ *measurable M* (*subprob-algebra N*)
assumes *g*: *g* $\in$ *measurable M* (*subprob-algebra L*)
shows ($\lambda x.$ *f x* $\bigotimes_M$ *g x*) $\in$ *measurable M* (*subprob-algebra* (*N* $\bigotimes_M$ *L*))
proof (*rule measurable-subprob-algebra*)
{ fix *x* **assume** *x* $\in$ *space M*
with *measurable-space*[*OF f*] *measurable-space*[*OF g*]

have *fx*: $f\ x \in$ *space* (*subprob-algebra N*) **and** *gx*: $g\ x \in$ *space* (*subprob-algebra L*)
by *auto*
interpret *F*: *subprob-space f x*
using *fx* **by** (*simp add*: *space-subprob-algebra*)
interpret *G*: *subprob-space g x*
using *gx* **by** (*simp add*: *space-subprob-algebra*)

interpret *pair-subprob-space f x g x* **..**
show *subprob-space* ($f\ x \bigotimes_M g\ x$) **by** *unfold-locales*
show *sets-eq*: *sets* ($f\ x \bigotimes_M g\ x$) = *sets* ($N \bigotimes_M L$)
using *fx gx* **by** (*simp add*: *space-subprob-algebra*)

have *1*: $\bigwedge A\ B.\ A \in$ *sets* $N \Longrightarrow B \in$ *sets* $L \Longrightarrow$ *emeasure* ($f\ x \bigotimes_M g\ x$) ($A \times B$) = *emeasure* ($f\ x$) A * *emeasure* ($g\ x$) B
using *fx gx* **by** (*intro G.emeasure-pair-measure-Times*) (*auto simp*: *space-subprob-algebra*)
have *emeasure* ($f\ x \bigotimes_M g\ x$) (*space* ($f\ x \bigotimes_M g\ x$)) =
emeasure ($f\ x$) (*space* ($f\ x$)) * *emeasure* ($g\ x$) (*space* ($g\ x$))
by (*subst G.emeasure-pair-measure-Times*[*symmetric*]) (*simp-all add*: *space-pair-measure*)
hence *2*: $\bigwedge A.\ A \in$ *sets* ($N \bigotimes_M L$) $\Longrightarrow$ *emeasure* ($f\ x \bigotimes_M g\ x$) (*space* $N \times$ *space* $L - A$) =
... − *emeasure* ($f\ x \bigotimes_M g\ x$) A
using *emeasure-compl*[*simplified*, *OF - P.emeasure-finite*]
unfolding *sets-eq*
unfolding *sets-eq-imp-space-eq*[*OF sets-eq*]
by (*simp add*: *space-pair-measure G.emeasure-pair-measure-Times*)
note *1 2 sets-eq* **}**
note *Times* = *this*(*1*) **and** *Compl* = *this*(*2*) **and** *sets-eq* = *this*(*3*)

fix *A* **assume** *A*: $A \in$ *sets* ($N \bigotimes_M L$)
show ($\lambda a.$ *emeasure* ($f\ a \bigotimes_M g\ a$) A) $\in$ *borel-measurable M*
using *Int-stable-pair-measure-generator pair-measure-closed A*
unfolding *sets-pair-measure*
proof (*induct A rule*: *sigma-sets-induct-disjoint*)
case (*basic A*) **then show** *?case*
by (*auto intro*!: *borel-measurable-times-ennreal simp*: *Times cong*: *measurable-cong*)
(*auto intro*!: *measurable-emeasure-kernel f g*)
next
case (*compl A*)
then have *A*: $A \in$ *sets* ($N \bigotimes_M L$)
by (*auto simp*: *sets-pair-measure*)
have ($\lambda x.$ *emeasure* ($f\ x$) (*space* ($f\ x$)) * *emeasure* ($g\ x$) (*space* ($g\ x$)) −
emeasure ($f\ x \bigotimes_M g\ x$) A) $\in$ *borel-measurable M* (**is** *?f* $\in$ *?M*)
using *compl*(*2*) *f g* **by** *measurable*
thus *?case* **by** (*simp add*: *Compl A cong*: *measurable-cong*)
next
case (*union A*)
then have *range A* $\subseteq$ *sets* ($N \bigotimes_M L$) *disjoint-family A*

```
    by (auto simp: sets-pair-measure)
  then have (λa. emeasure (f a ⨂M g a) (⋃i. A i)) ∈ borel-measurable M ⟷
    (λa. ∑i. emeasure (f a ⨂M g a) (A i)) ∈ borel-measurable M
    by (intro measurable-cong suminf-emeasure[symmetric])
       (auto simp: sets-eq)
  also have ...
    using union by auto
  finally show ?case .
 qed simp
qed

lemma restrict-space-measurable:
 assumes X: X ≠ {} X ∈ sets K
 assumes N: N ∈ measurable M (subprob-algebra K)
 shows (λx. restrict-space (N x) X) ∈ measurable M (subprob-algebra (restrict-space
K X))
proof (rule measurable-subprob-algebra)
 fix a assume a: a ∈ space M
 from N[THEN measurable-space, OF this]
 have subprob-space (N a) and [simp]: sets (N a) = sets K space (N a) = space
K
   by (auto simp add: space-subprob-algebra dest: sets-eq-imp-space-eq)
 then interpret subprob-space N a
   by simp
 show subprob-space (restrict-space (N a) X)
 proof
   show space (restrict-space (N a) X) ≠ {}
     using X by (auto simp add: space-restrict-space)
   show emeasure (restrict-space (N a) X) (space (restrict-space (N a) X)) ≤ 1
       using X by (simp add: emeasure-restrict-space space-restrict-space sub-
prob-emeasure-le-1)
 qed
 show sets (restrict-space (N a) X) = sets (restrict-space K X)
   by (intro sets-restrict-space-cong) fact
next
 fix A assume A: A ∈ sets (restrict-space K X)
 show (λa. emeasure (restrict-space (N a) X) A) ∈ borel-measurable M
 proof (subst measurable-cong)
   fix a assume a ∈ space M
   from N[THEN measurable-space, OF this]
   have [simp]: sets (N a) = sets K space (N a) = space K
     by (auto simp add: space-subprob-algebra dest: sets-eq-imp-space-eq)
   show emeasure (restrict-space (N a) X) A = emeasure (N a) (A ∩ X)
    using X A by (subst emeasure-restrict-space) (auto simp add: sets-restrict-space
ac-simps)
 next
   show (λw. emeasure (N w) (A ∩ X)) ∈ borel-measurable M
     using A X
     by (intro measurable-compose[OF N measurable-emeasure-subprob-algebra])
```

(*auto simp*: *sets-restrict-space*)
qed
qed

7.2 Properties of "return"

definition *return* :: *'a measure* ⇒ *'a* ⇒ *'a measure* **where**
return R x = *measure-of* (*space R*) (*sets R*) (λ*A*. *indicator A x*)

lemma *space-return*[*simp*]: *space* (*return M x*) = *space M*
by (*simp add*: *return-def*)

lemma *sets-return*[*simp*]: *sets* (*return M x*) = *sets M*
by (*simp add*: *return-def*)

lemma *measurable-return1*[*simp*]: *measurable* (*return N x*) *L* = *measurable N L*
by (*simp cong*: *measurable-cong-sets*)

lemma *measurable-return2*[*simp*]: *measurable L* (*return N x*) = *measurable L N*
by (*simp cong*: *measurable-cong-sets*)

lemma *return-sets-cong*: *sets M* = *sets N* ⟹ *return M* = *return N*
by (*auto dest*: *sets-eq-imp-space-eq simp*: *fun-eq-iff return-def*)

lemma *return-cong*: *sets A* = *sets B* ⟹ *return A x* = *return B x*
by (*auto simp add*: *return-def dest*: *sets-eq-imp-space-eq*)

lemma *emeasure-return*[*simp*]:
assumes *A* ∈ *sets M*
shows *emeasure* (*return M x*) *A* = *indicator A x*
proof (*rule emeasure-measure-of*[*OF return-def*])
show *sets M* ⊆ *Pow* (*space M*) **by** (*rule sets.space-closed*)
show *positive* (*sets* (*return M x*)) (λ*A*. *indicator A x*) **by** (*simp add*: *positive-def*)
from *assms* **show** *A* ∈ *sets* (*return M x*) **unfolding** *return-def* **by** *simp*
show *countably-additive* (*sets* (*return M x*)) (λ*A*. *indicator A x*)
by (*auto intro*!: *countably-additiveI suminf-indicator*)
qed

lemma *prob-space-return*: *x* ∈ *space M* ⟹ *prob-space* (*return M x*)
by *rule simp*

lemma *subprob-space-return*: *x* ∈ *space M* ⟹ *subprob-space* (*return M x*)
by (*intro prob-space-return prob-space-imp-subprob-space*)

lemma *subprob-space-return-ne*:
assumes *space M* ≠ {} **shows** *subprob-space* (*return M x*)
proof
show *emeasure* (*return M x*) (*space* (*return M x*)) ≤ *1*
by (*subst emeasure-return*) (*auto split*: *split-indicator*)

qed (*simp*, *fact*)

lemma *measure-return*: **assumes** *X*: $X \in sets\ M$ **shows** *measure* (*return M x*) *X* = *indicator X x*
unfolding *measure-def emeasure-return*[*OF X*, *of x*] **by** (*simp split*: *split-indicator*)

lemma *AE-return*:
assumes [*simp*]: $x \in space\ M$ **and** [*measurable*]: *Measurable.pred M P*
shows (*AE y in return M x*. *P y*) $\longleftrightarrow$ *P x*
proof −
have (*AE y in return M x*. $y \notin \{x \in space\ M.\ \neg\ P\ x\}$) $\longleftrightarrow$ *P x*
by (*subst AE-iff-null-sets*[*symmetric*]) (*simp-all add*: *null-sets-def split*: *split-indicator*)
also have (*AE y in return M x*. $y \notin \{x \in space\ M.\ \neg\ P\ x\}$) $\longleftrightarrow$ (*AE y in return M x*. *P y*)
by (*rule AE-cong*) *auto*
finally show *?thesis* **.**
qed

lemma *nn-integral-return*:
assumes $x \in space\ M$ $g \in$ *borel-measurable M*
shows ($\int^+ a.\ g\ a\ \partial return\ M\ x$) = *g x*
proof−
interpret *prob-space return M x* **by** (*rule prob-space-return*[*OF* ‹$x \in space\ M$›])
have ($\int^+ a.\ g\ a\ \partial return\ M\ x$) = ($\int^+ a.\ g\ x\ \partial return\ M\ x$) **using** *assms*
by (*intro nn-integral-cong-AE*) (*auto simp*: *AE-return*)
also have ... = *g x*
using *nn-integral-const*[*of return M x*] *emeasure-space-1* **by** *simp*
finally show *?thesis* **.**
qed

lemma *integral-return*:
fixes *g* :: - ⇒ *'a* :: {*banach*, *second-countable-topology*}
assumes $x \in space\ M$ $g \in$ *borel-measurable M*
shows ($\int a.\ g\ a\ \partial return\ M\ x$) = *g x*
proof−
interpret *prob-space return M x* **by** (*rule prob-space-return*[*OF* ‹$x \in space\ M$›])
have ($\int a.\ g\ a\ \partial return\ M\ x$) = ($\int a.\ g\ x\ \partial return\ M\ x$) **using** *assms*
by (*intro integral-cong-AE*) (*auto simp*: *AE-return*)
then show *?thesis*
using *prob-space* **by** *simp*
qed

lemma *return-measurable*[*measurable*]: *return N* ∈ *measurable N* (*subprob-algebra N*)
by (*rule measurable-subprob-algebra*) (*auto simp*: *subprob-space-return*)

lemma *distr-return*:
assumes *f* ∈ *measurable M N* **and** $x \in space\ M$
shows *distr* (*return M x*) *N f* = *return N* (*f x*)

using *assms* **by** (*intro measure-eqI*) (*simp-all add*: *indicator-def emeasure-distr*)

lemma *return-restrict-space*:
Ω ∈ *sets M* ⟹ *return* (*restrict-space M* Ω) *x* = *restrict-space* (*return M x*) Ω
by (*auto intro*!: *measure-eqI simp*: *sets-restrict-space emeasure-restrict-space*)

lemma *measurable-distr2*:
assumes *f*[*measurable*]: *case-prod f* ∈ *measurable* (*L* ⨂$_M$ *M*) *N*
assumes *g*[*measurable*]: *g* ∈ *measurable L* (*subprob-algebra M*)
shows (λ*x*. *distr* (*g x*) *N* (*f x*)) ∈ *measurable L* (*subprob-algebra N*)
proof −
have (λ*x*. *distr* (*g x*) *N* (*f x*)) ∈ *measurable L* (*subprob-algebra N*)
⟷ (λ*x*. *distr* (*return L x* ⨂$_M$ *g x*) *N* (*case-prod f*)) ∈ *measurable L* (*subprob-algebra N*)
proof (*rule measurable-cong*)
fix *x* **assume** *x*: *x* ∈ *space L*
have *gx*: *g x* ∈ *space* (*subprob-algebra M*)
using *measurable-space*[*OF g x*] **.**
then have [*simp*]: *sets* (*g x*) = *sets M*
by (*simp add*: *space-subprob-algebra*)
then have [*simp*]: *space* (*g x*) = *space M*
by (*rule sets-eq-imp-space-eq*)
let *?R* = *return L x*
from *measurable-compose-Pair1*[*OF x f*] **have** *f-M*′: *f x* ∈ *measurable M N*
by *simp*
interpret *subprob-space g x*
using *gx* **by** (*simp add*: *space-subprob-algebra*)
have *space-pair-M*′[*simp*]: ⋀*X*. *space* (*X* ⨂$_M$ *g x*) = *space* (*X* ⨂$_M$ *M*)
by (*simp add*: *space-pair-measure*)
show *distr* (*g x*) *N* (*f x*) = *distr* (*?R* ⨂$_M$ *g x*) *N* (*case-prod f*) (**is** *?l* = *?r*)
proof (*rule measure-eqI*)
show *sets ?l* = *sets ?r*
by *simp*
next
fix *A* **assume** *A* ∈ *sets ?l*
then have *A*[*measurable*]: *A* ∈ *sets N*
by *simp*
then have *emeasure ?r A* = *emeasure* (*?R* ⨂$_M$ *g x*) ((λ(*x*, *y*). *f x y*) −‘ *A* ∩ *space* (*?R* ⨂$_M$ *g x*))
by (*auto simp add*: *emeasure-distr f-M*′ *cong*: *measurable-cong-sets*)
also have ... = (∫$^+$*M*′′. *emeasure* (*g x*) (*f M*′′ −‘ *A* ∩ *space M*) ∂*?R*)
apply (*subst emeasure-pair-measure-alt*)
apply (*rule measurable-sets*[*OF* - *A*])
apply (*auto simp add*: *f-M*′ *cong*: *measurable-cong-sets*)
apply (*intro nn-integral-cong arg-cong*[**where** *f*=*emeasure* (*g x*)])
apply (*auto simp*: *space-subprob-algebra space-pair-measure*)
done
also have ... = *emeasure* (*g x*) (*f x* −‘ *A* ∩ *space M*)
by (*subst nn-integral-return*)

```
        (auto simp: x intro!: measurable-emeasure)
    also have ... = emeasure ?l A
      by (simp add: emeasure-distr f-M' cong: measurable-cong-sets)
    finally show emeasure ?l A = emeasure ?r A ..
  qed
 qed
 also have ...
  apply (intro measurable-compose[OF measurable-pair-measure measurable-distr])
   apply (rule return-measurable)
   apply measurable
   done
 finally show ?thesis .
qed

lemma nn-integral-measurable-subprob-algebra2:
 assumes f[measurable]: (λ(x, y). f x y) ∈ borel-measurable (M ⨂M N)
 assumes N[measurable]: L ∈ measurable M (subprob-algebra N)
 shows (λx. integral^N (L x) (f x)) ∈ borel-measurable M
proof −
 note nn-integral-measurable-subprob-algebra[measurable]
 note measurable-distr2[measurable]
 have (λx. integral^N (distr (L x) (M ⨂M N) (λy. (x, y))) (λ(x, y). f x y)) ∈
borel-measurable M
   by measurable
 then show (λx. integral^N (L x) (f x)) ∈ borel-measurable M
   by (rule measurable-cong[THEN iffD1, rotated])
     (simp add: nn-integral-distr)
qed

lemma emeasure-measurable-subprob-algebra2:
 assumes A[measurable]: (SIGMA x:space M. A x) ∈ sets (M ⨂M N)
 assumes L[measurable]: L ∈ measurable M (subprob-algebra N)
 shows (λx. emeasure (L x) (A x)) ∈ borel-measurable M
proof −
 { fix x assume x ∈ space M
   then have Pair x −‘ Sigma (space M) A = A x
     by auto
   with sets-Pair1[OF A, of x] have A x ∈ sets N
     by auto }
 note ∗∗ = this

 have ∗: ⋀x. fst x ∈ space M ⟹ snd x ∈ A (fst x) ⟷ x ∈ (SIGMA x:space M.
A x)
   by (auto simp: fun-eq-iff)
 have (λ(x, y). indicator (A x) y::ennreal) ∈ borel-measurable (M ⨂M N)
   apply measurable
   apply (subst measurable-cong)
   apply (rule ∗)
   apply (auto simp: space-pair-measure)
```

```
    done
  then have (λx. integral^N (L x) (indicator (A x))) ∈ borel-measurable M
    by (intro nn-integral-measurable-subprob-algebra2[where N=N] L)
  then show (λx. emeasure (L x) (A x)) ∈ borel-measurable M
    apply (rule measurable-cong[THEN iffD1, rotated])
    apply (rule nn-integral-indicator)
    apply (simp add: subprob-measurableD[OF L] **)
    done
qed

lemma measure-measurable-subprob-algebra2:
  assumes A[measurable]: (SIGMA x:space M. A x) ∈ sets (M ⨂_M N)
  assumes L[measurable]: L ∈ measurable M (subprob-algebra N)
  shows (λx. measure (L x) (A x)) ∈ borel-measurable M
  unfolding measure-def
 by (intro borel-measurable-enn2real emeasure-measurable-subprob-algebra2[OF assms])

definition select-sets M = (SOME N. sets M = sets (subprob-algebra N))

lemma select-sets1:
  sets M = sets (subprob-algebra N) ⟹ sets M = sets (subprob-algebra (select-sets
M))
  unfolding select-sets-def by (rule someI)

lemma sets-select-sets[simp]:
  assumes sets: sets M = sets (subprob-algebra N)
  shows sets (select-sets M) = sets N
  unfolding select-sets-def
proof (rule someI2)
  show sets M = sets (subprob-algebra N)
    by fact
next
  fix L assume sets M = sets (subprob-algebra L)
  with sets have eq: space (subprob-algebra N) = space (subprob-algebra L)
    by (intro sets-eq-imp-space-eq) simp
  show sets L = sets N
  proof cases
    assume space (subprob-algebra N) = {}
     with space-subprob-algebra-empty-iff[of N] space-subprob-algebra-empty-iff[of
L]
    show ?thesis
      by (simp add: eq space-empty-iff)
  next
    assume space (subprob-algebra N) ≠ {}
    with eq show ?thesis
      by (fastforce simp add: space-subprob-algebra)
  qed
qed
```

lemma *space-select-sets*[*simp*]:
sets M = sets (subprob-algebra N) ⟹ space (select-sets M) = space N
by (*intro sets-eq-imp-space-eq sets-select-sets*)

7.3 Join

definition *join* :: *'a measure measure ⇒ 'a measure* **where**
join M = measure-of (space (select-sets M)) (sets (select-sets M)) (λB. ∫⁺ M'. emeasure M' B ∂M)

lemma
shows *space-join*[*simp*]: *space (join M) = space (select-sets M)*
and *sets-join*[*simp*]: *sets (join M) = sets (select-sets M)*
by (*simp-all add*: *join-def*)

lemma *emeasure-join*:
assumes *M*[*simp*, *measurable-cong*]: *sets M = sets (subprob-algebra N)* **and** *A*: *A ∈ sets N*
shows *emeasure (join M) A = (∫⁺ M'. emeasure M' A ∂M)*
proof (*rule emeasure-measure-of*[*OF join-def*])
show *countably-additive (sets (join M)) (λB. ∫⁺ M'. emeasure M' B ∂M)*
proof (*rule countably-additiveI*)
fix *A* :: *nat ⇒ 'a set* **assume** *A*: *range A ⊆ sets (join M) disjoint-family A*
have *(∑ i. ∫⁺ M'. emeasure M' (A i) ∂M) = (∫⁺M'. (∑ i. emeasure M' (A i)) ∂M)*
using *A* **by** (*subst nn-integral-suminf*) (*auto simp*: *measurable-emeasure-subprob-algebra*)
also have *... = (∫⁺M'. emeasure M' (⋃ i. A i) ∂M)*
proof (*rule nn-integral-cong*)
fix *M'* **assume** *M' ∈ space M*
then show *(∑ i. emeasure M' (A i)) = emeasure M' (⋃ i. A i)*
using *A sets-eq-imp-space-eq*[*OF M*] **by** (*simp add*: *suminf-emeasure space-subprob-algebra*)
qed
finally show *(∑ i. ∫⁺M'. emeasure M' (A i) ∂M) = (∫⁺M'. emeasure M' (⋃ i. A i) ∂M)* **.**
qed
qed (*auto simp*: *A sets.space-closed positive-def*)

lemma *measurable-join*:
join ∈ measurable (subprob-algebra (subprob-algebra N)) (subprob-algebra N)
proof (*cases space N ≠ {}*, *rule measurable-subprob-algebra*)
fix *A* **assume** *A ∈ sets N*
let *?B = borel-measurable (subprob-algebra (subprob-algebra N))*
have *(λM'. emeasure (join M') A) ∈ ?B ⟷ (λM'. (∫⁺ M''. emeasure M'' A ∂M')) ∈ ?B*
proof (*rule measurable-cong*)
fix *M'* **assume** *M' ∈ space (subprob-algebra (subprob-algebra N))*
then show *emeasure (join M') A = (∫⁺ M''. emeasure M'' A ∂M')*
by (*intro emeasure-join*) (*auto simp*: *space-subprob-algebra ‹A∈sets N›*)

qed
also have ($\lambda M'$. $\int^+ M''$. *emeasure* M'' A $\partial M'$) $\in$ *?B*
using *measurable-emeasure-subprob-algebra*[*OF* ‹A∈*sets* N›]
by (*rule nn-integral-measurable-subprob-algebra*)
finally show ($\lambda M'$. *emeasure* (*join* M') A) $\in$ *borel-measurable* (*subprob-algebra* (*subprob-algebra* N)) **.**
next
assume [*simp*]: *space* $N \neq \{\}$
fix M **assume** M: $M \in$ *space* (*subprob-algebra* (*subprob-algebra* N))
then have ($\int^+ M'$. *emeasure* M' (*space* N) ∂M) $\leq$ ($\int^+ M'$. 1 ∂M)
apply (*intro nn-integral-mono*)
apply (*auto simp*: *space-subprob-algebra*
dest!: *sets-eq-imp-space-eq subprob-space.emeasure-space-le-1*)
done
with M **show** *subprob-space* (*join* M)
by (*intro subprob-spaceI*)
(*auto simp*: *emeasure-join space-subprob-algebra M dest*: *subprob-space.emeasure-space-le-1*)
next
assume ¬(*space* $N \neq \{\}$)
thus *?thesis* **by** (*simp add*: *measurable-empty-iff space-subprob-algebra-empty-iff*)
qed (*auto simp*: *space-subprob-algebra*)

lemma *nn-integral-join*:
assumes f: $f \in$ *borel-measurable* N
and M[*measurable-cong*]: *sets* M = *sets* (*subprob-algebra* N)
shows ($\int^+ x$. f x ∂*join* M) = ($\int^+ M'$. $\int^+ x$. f x $\partial M'$ ∂M)
using f
proof *induct*
case (*cong f g*)
moreover have *integral*^{N} (*join* M) f = *integral*^{N} (*join* M) g
by (*intro nn-integral-cong cong*) (*simp add*: M)
moreover from M **have** ($\int^+ M'$. *integral*^{N} M' f ∂M) = ($\int^+ M'$. *integral*^{N} M' g ∂M)
by (*intro nn-integral-cong cong*)
(*auto simp add*: *space-subprob-algebra dest*!: *sets-eq-imp-space-eq*)
ultimately show *?case*
by *simp*
next
case (*set A*)
with M **have** ($\int^+ M'$. *integral*^{N} M' (*indicator* A) ∂M) = ($\int^+ M'$. *emeasure* M' A ∂M)
by (*intro nn-integral-cong nn-integral-indicator*)
(*auto simp*: *space-subprob-algebra dest*!: *sets-eq-imp-space-eq*)
with *set* **show** *?case*
using M **by** (*simp add*: *emeasure-join*)
next
case (*mult f c*)
have ($\int^+ M'$. $\int^+ x$. $c * f$ x $\partial M'$ ∂M) = ($\int^+ M'$. $c * \int^+ x$. f x $\partial M'$ ∂M)
using *mult* M M[*THEN sets-eq-imp-space-eq*]

by (*intro nn-integral-cong nn-integral-cmult*) (*auto simp add*: *space-subprob-algebra*)
also have ... = $c * (\int^+ M'.\ \int^+ x.\ f\ x\ \partial M'\ \partial M)$
using *nn-integral-measurable-subprob-algebra*[*OF mult*(*2*)]
by (*intro nn-integral-cmult mult*) (*simp add*: *M*)
also have ... = $c * (integral^N\ (join\ M)\ f)$
by (*simp add*: *mult*)
also have ... = $(\int^+ x.\ c * f\ x\ \partial join\ M)$
using *mult*(*2*,*3*) **by** (*intro nn-integral-cmult*[*symmetric*] *mult*) (*simp add*: *M cong*: *measurable-cong-sets*)
finally show *?case* **by** *simp*
next
case (*add f g*)
have $(\int^+ M'.\ \int^+ x.\ f\ x + g\ x\ \partial M'\ \partial M) = (\int^+ M'.\ (\int^+ x.\ f\ x\ \partial M') + (\int^+ x.\ g\ x\ \partial M')\ \partial M)$
using *add M M*[*THEN sets-eq-imp-space-eq*]
by (*intro nn-integral-cong nn-integral-add*) (*auto simp add*: *space-subprob-algebra*)
also have ... = $(\int^+ M'.\ \int^+ x.\ f\ x\ \partial M'\ \partial M) + (\int^+ M'.\ \int^+ x.\ g\ x\ \partial M'\ \partial M)$
using *nn-integral-measurable-subprob-algebra*[*OF add*(*1*)]
using *nn-integral-measurable-subprob-algebra*[*OF add*(*4*)]
by (*intro nn-integral-add add*) (*simp-all add*: *M*)
also have ... = $(integral^N\ (join\ M)\ f) + (integral^N\ (join\ M)\ g)$
by (*simp add*: *add*)
also have ... = $(\int^+ x.\ f\ x + g\ x\ \partial join\ M)$
using *add* **by** (*intro nn-integral-add*[*symmetric*] *add*) (*simp-all add*: *M cong*: *measurable-cong-sets*)
finally show *?case* **by** (*simp add*: *ac-simps*)
next
case (*seq F*)
have $(\int^+ M'.\ \int^+ x.\ (SUP\ i.\ F\ i)\ x\ \partial M'\ \partial M) = (\int^+ M'.\ (SUP\ i.\ \int^+ x.\ F\ i\ x\ \partial M')\ \partial M)$
using *seq M M*[*THEN sets-eq-imp-space-eq*] **unfolding** *SUP-apply*
by (*intro nn-integral-cong nn-integral-monotone-convergence-SUP*)
(*auto simp add*: *space-subprob-algebra*)
also have ... = $(SUP\ i.\ \int^+ M'.\ \int^+ x.\ F\ i\ x\ \partial M'\ \partial M)$
using *nn-integral-measurable-subprob-algebra*[*OF seq*(*1*)] *seq*
by (*intro nn-integral-monotone-convergence-SUP*)
(*simp-all add*: *M incseq-nn-integral incseq-def le-fun-def nn-integral-mono*)
also have ... = $(SUP\ i.\ integral^N\ (join\ M)\ (F\ i))$
by (*simp add*: *seq*)
also have ... = $(\int^+ x.\ (SUP\ i.\ F\ i\ x)\ \partial join\ M)$
using *seq* **by** (*intro nn-integral-monotone-convergence-SUP*[*symmetric*] *seq*)
(*simp-all add*: *M cong*: *measurable-cong-sets*)
finally show *?case* **by** (*simp add*: *ac-simps image-comp*)
qed

lemma *measurable-join1*:
$[\![f \in measurable\ N\ K;\ sets\ M = sets\ (subprob\text{-}algebra\ N)]\!]$
$\Longrightarrow f \in measurable\ (join\ M)\ K$
by(*simp add*: *measurable-def*)

lemma
fixes f :: - ⇒ *real*
assumes *f-measurable* [*measurable*]: $f \in$ *borel-measurable* N
and *f-bounded*: $\bigwedge x.\ x \in$ *space* $N \Longrightarrow |f\,x| \leq B$
and M [*measurable-cong*]: *sets* M = *sets* (*subprob-algebra* N)
and *fin*: *finite-measure* M
and *M-bounded*: *AE* M' *in* M. *emeasure* M' (*space* M') ≤ *ennreal* B'
shows *integrable-join*: *integrable* (*join* M) f (**is** *?integrable*)
and *integral-join*: *integral*^{L} (*join* M) $f = \int\ M'.$ *integral*^{L} M' f ∂M (**is** *?integral*)
proof(*case-tac* [!] *space* $N = \{\}$)
assume ∗: *space* $N = \{\}$
show *?integrable*
using M ∗ **by**(*simp add*: *real-integrable-def measurable-def nn-integral-empty*)
have $(\int\ M'.$ *integral*^{L} M' f $\partial M) = (\int\ M'.\ 0\ \partial M)$
proof(*rule Bochner-Integration.integral-cong*)
fix M'
assume $M' \in$ *space* M
with *sets-eq-imp-space-eq*[*OF* M] **have** *space* M' = *space* N
by(*auto simp add*: *space-subprob-algebra dest*: *sets-eq-imp-space-eq*)
with ∗ **show** $(\int\ x.\ f\,x\ \partial M') = 0$ **by**(*simp add*: *Bochner-Integration.integral-empty*)
qed *simp*
then show *?integral*
using M ∗ **by**(*simp add*: *Bochner-Integration.integral-empty*)
next
assume ∗: *space* $N \neq \{\}$

from ∗ **have** B [*simp*]: $0 \leq B$ **by**(*auto dest*: *f-bounded*)

have [*measurable*]: $f \in$ *borel-measurable* (*join* M) **using** *f-measurable* M
by(*rule measurable-join1*)

{ fix f M'
assume [*measurable*]: $f \in$ *borel-measurable* N
and *f-bounded*: $\bigwedge x.\ x \in$ *space* $N \Longrightarrow f\,x \leq B$
and $M' \in$ *space* M *emeasure* M' (*space* M') ≤ *ennreal* B'
have *AE* x *in* M'. *ennreal* $(f\,x) \leq$ *ennreal* B
proof(*rule AE-I2*)
fix x
assume $x \in$ *space* M'
with ‹$M' \in$ *space* M› *sets-eq-imp-space-eq*[*OF* M]
have $x \in$ *space* N **by**(*auto simp add*: *space-subprob-algebra dest*: *sets-eq-imp-space-eq*)
from *f-bounded*[*OF this*] **show** *ennreal* $(f\,x) \leq$ *ennreal* B **by** *simp*
qed
then have $(\int^+ x.$ *ennreal* $(f\,x)\ \partial M') \leq (\int^+ x.$ *ennreal* $B\ \partial M')$
by(*rule nn-integral-mono-AE*)
also have ... = *ennreal* B ∗ *emeasure* M' (*space* M') **by**(*simp*)
also have ... ≤ *ennreal* B ∗ *ennreal* B' **by**(*rule mult-left-mono*)(*fact*, *simp*)
also have ... ≤ *ennreal* B ∗ *ennreal* $|B'|$ **by**(*rule mult-left-mono*)(*simp-all*)

finally have $(\int^+ x.\ ennreal\ (f\ x)\ \partial M') \leq ennreal\ (B * |B'|)$ **by** (*simp add*: *ennreal-mult*) **}**
note *bounded1* = *this*

have *bounded*:
$\bigwedge f.\ [\![\ f \in borel\text{-}measurable\ N;\ \bigwedge x.\ x \in space\ N \Longrightarrow f\ x \leq B\ ]\!]$
$\Longrightarrow (\int^+ x.\ ennreal\ (f\ x)\ \partial join\ M) \neq top$
proof −
fix *f*
assume [*measurable*]: $f \in borel\text{-}measurable\ N$
and *f-bounded*: $\bigwedge x.\ x \in space\ N \Longrightarrow f\ x \leq B$
have $(\int^+ x.\ ennreal\ (f\ x)\ \partial join\ M) = (\int^+ M'.\ \int^+ x.\ ennreal\ (f\ x)\ \partial M'\ \partial M)$
by(*rule nn-integral-join*[*OF* - *M*]) *simp*
also have $\ldots \leq \int^+ M'.\ B * |B'|\ \partial M$
using *bounded1*[*OF* ‹$f \in borel\text{-}measurable\ N$› *f-bounded*]
by(*rule nn-integral-mono-AE*[*OF AE-mp*[*OF M-bounded AE-I2*], *rule-format*])
also have $\ldots = B * |B'| * emeasure\ M\ (space\ M)$ **by** *simp*
also have $\ldots < \infty$
using *finite-measure.finite-emeasure-space*[*OF fin*]
by(*simp add*: *ennreal-mult-less-top less-top*)
finally show *?thesis f* **by** *simp*
qed
have *f-pos*: $(\int^+ x.\ ennreal\ (f\ x)\ \partial join\ M) \neq \infty$
and *f-neg*: $(\int^+ x.\ ennreal\ (-\ f\ x)\ \partial join\ M) \neq \infty$
using *f-bounded* **by**(*auto del*: *notI intro*!: *bounded simp add*: *abs-le-iff*)

show *?integrable* **using** *f-pos f-neg* **by**(*simp add*: *real-integrable-def*)

note [*measurable*] = *nn-integral-measurable-subprob-algebra*

have *int-f*: $(\int^+ x.\ f\ x\ \partial join\ M) = \int^+ M'.\ \int^+ x.\ f\ x\ \partial M'\ \partial M$
by(*simp add*: *nn-integral-join*[*OF* - *M*])
have *int-mf*: $(\int^+ x.\ -\ f\ x\ \partial join\ M) = (\int^+ M'.\ \int^+ x.\ -\ f\ x\ \partial M'\ \partial M)$
by(*simp add*: *nn-integral-join*[*OF* - *M*])

have *pos-finite*: *AE M' in M*. $(\int^+ x.\ f\ x\ \partial M') \neq \infty$
using *AE-space M-bounded*
proof *eventually-elim*
fix *M'* **assume** $M' \in space\ M$ *emeasure M'* $(space\ M') \leq ennreal\ B'$
then have $(\int^+ x.\ ennreal\ (f\ x)\ \partial M') \leq ennreal\ (B * |B'|)$
using *f-measurable* **by**(*auto intro*!: *bounded1 dest*: *f-bounded*)
then show $(\int^+ x.\ ennreal\ (f\ x)\ \partial M') \neq \infty$
by (*auto simp*: *top-unique*)
qed
hence [*simp*]: $(\int^+ M'.\ ennreal\ (enn2real\ (\int^+ x.\ f\ x\ \partial M'))\ \partial M) = (\int^+ M'.\ \int^+ x.\ f\ x\ \partial M'\ \partial M)$
by (*rule nn-integral-cong-AE*[*OF AE-mp*]) (*simp add*: *less-top*)
from *f-pos* **have** [*simp*]: *integrable M* $(\lambda M'.\ enn2real\ (\int^+ x.\ f\ x\ \partial M'))$
by(*simp add*: *int-f real-integrable-def nn-integral-0-iff-AE*[*THEN iffD2*] *en-*

nreal-neg enn2real-nonneg)

have *neg-finite*: *AE M' in M*. $(\int^+ x. -f\,x\,\partial M') \neq \infty$
using *AE-space M-bounded*
proof *eventually-elim*
fix *M'* **assume** *M' ∈ space M emeasure M' (space M') ≤ ennreal B'*
then have $(\int^+ x.$ *ennreal* $(-f\,x)\,\partial M') \leq$ *ennreal* $(B * |B'|)$
using *f-measurable* **by**(*auto intro*!: *bounded1 dest*: *f-bounded*)
then show $(\int^+ x.$ *ennreal* $(-f\,x)\,\partial M') \neq \infty$
by (*auto simp*: *top-unique*)
qed
hence [*simp*]: $(\int^+ M'.$ *ennreal* (*enn2real* $(\int^+ x. -f\,x\,\partial M'))\,\partial M) = (\int^+ M'.$ $\int^+ x. -f\,x\,\partial M'\,\partial M)$
by (*rule nn-integral-cong-AE*[*OF AE-mp*]) (*simp add*: *less-top*)
from *f-neg* **have** [*simp*]: *integrable M* $(\lambda M'.$ *enn2real* $(\int^+ x. -f\,x\,\partial M'))$
by(*simp add*: *int-mf real-integrable-def nn-integral-0-iff-AE*[*THEN iffD2*] *ennreal-neg enn2real-nonneg*)

have $(\int x.\, f\,x\,\partial$*join M*$) =$ *enn2real* $(\int^+ N. \int^+ x.\, f\,x\,\partial N\,\partial M) -$ *enn2real* $(\int^+ N. \int^+ x. -f\,x\,\partial N\,\partial M)$
unfolding *real-lebesgue-integral-def*[*OF ‹?integrable›*] **by** (*simp add*: *nn-integral-join*[*OF - M*])
also have $\ldots = (\int N.$ *enn2real* $(\int^+ x.\, f\,x\,\partial N)\,\partial M) - (\int N.$ *enn2real* $(\int^+ x. -f\,x\,\partial N)\,\partial M)$
using *pos-finite neg-finite* **by** (*subst* (*1 2*) *integral-eq-nn-integral*) (*auto simp*: *enn2real-nonneg*)
also have $\ldots = (\int N.$ *enn2real* $(\int^+ x.\, f\,x\,\partial N) -$ *enn2real* $(\int^+ x. -f\,x\,\partial N)\,\partial M)$
by *simp*
also have $\ldots = \int M'. \int x.\, f\,x\,\partial M'\,\partial M$
proof (*rule integral-cong-AE*)
show *AE x in M*.
enn2real $(\int^+ x.$ *ennreal* $(f\,x)\,\partial x) -$ *enn2real* $(\int^+ x.$ *ennreal* $(-f\,x)\,\partial x) =$ $integral^L\ x\ f$
using *AE-space M-bounded*
proof *eventually-elim*
fix *M'* **assume** *M' ∈ space M emeasure M' (space M') ≤ B'*
then interpret *subprob-space M'*
by (*auto simp*: *M*[*THEN sets-eq-imp-space-eq*] *space-subprob-algebra*)

from *‹M' ∈ space M›* *sets-eq-imp-space-eq*[*OF M*]
have [*measurable-cong*]: *sets M' = sets N* **by**(*simp add*: *space-subprob-algebra*)
hence [*simp*]: *space M' = space N* **by**(*rule sets-eq-imp-space-eq*)
have *integrable M' f*
by(*rule integrable-const-bound*[**where** *B=B*])(*auto simp add*: *f-bounded*)
then show *enn2real* $(\int^+ x.\, f\,x\,\partial M') -$ *enn2real* $(\int^+ x. -f\,x\,\partial M') = \int x.$ $f\,x\,\partial M'$
by(*simp add*: *real-lebesgue-integral-def*)
qed

qed *simp-all*
finally show *?integral* **by** *simp*
qed

lemma *join-assoc*:
assumes *M*[*measurable-cong*]: *sets M* = *sets* (*subprob-algebra* (*subprob-algebra N*))
shows *join* (*distr M* (*subprob-algebra N*) *join*) = *join* (*join M*)
proof (*rule measure-eqI*)
fix *A* **assume** *A* ∈ *sets* (*join* (*distr M* (*subprob-algebra N*) *join*))
then have *A*: *A* ∈ *sets N* **by** *simp*
show *emeasure* (*join* (*distr M* (*subprob-algebra N*) *join*)) *A* = *emeasure* (*join* (*join M*)) *A*
using *measurable-join*[*of N*]
by (*auto simp*: *M A nn-integral-distr emeasure-join measurable-emeasure-subprob-algebra sets-eq-imp-space-eq*[*OF M*] *space-subprob-algebra nn-integral-join*[*OF - M*]
intro!: *nn-integral-cong emeasure-join*)
qed (*simp add*: *M*)

lemma *join-return*:
assumes *sets M* = *sets N* **and** *subprob-space M*
shows *join* (*return* (*subprob-algebra N*) *M*) = *M*
by (*rule measure-eqI*)
(*simp-all add*: *emeasure-join space-subprob-algebra measurable-emeasure-subprob-algebra nn-integral-return assms*)

lemma *join-return'*:
assumes *sets N* = *sets M*
shows *join* (*distr M* (*subprob-algebra N*) (*return N*)) = *M*
apply (*rule measure-eqI*)
apply (*simp add*: *assms*)
apply (*subgoal-tac return N* ∈ *measurable M* (*subprob-algebra N*))
apply (*simp add*: *emeasure-join nn-integral-distr measurable-emeasure-subprob-algebra assms*)
apply (*subst measurable-cong-sets*, *rule assms*[*symmetric*], *rule refl*, *rule return-measurable*)
done

lemma *join-distr-distr*:
fixes *f* :: *'a* ⇒ *'b* **and** *M* :: *'a measure measure* **and** *N* :: *'b measure*
assumes *sets M* = *sets* (*subprob-algebra R*) **and** *f* ∈ *measurable R N*
shows *join* (*distr M* (*subprob-algebra N*) (λ*M*. *distr M N f*)) = *distr* (*join M*) *N f* (**is** *?r* = *?l*)
proof (*rule measure-eqI*)
fix *A* **assume** *A* ∈ *sets ?r*
hence *A-in-N*: *A* ∈ *sets N* **by** *simp*

from *assms* **have** *f* ∈ *measurable* (*join M*) *N*
by (*simp cong*: *measurable-cong-sets*)

moreover from *assms* **and** *A-in-N* **have** $f-`A \cap space\ R \in sets\ R$
by (*intro measurable-sets*) *simp-all*
ultimately have *emeasure* (*distr* (*join M*) *N f*) *A* = $\int^+ M'$. *emeasure M'* ($f-`A \cap space\ R$) *∂M*
by (*simp-all add*: *A-in-N emeasure-distr emeasure-join assms*)

also have ... = $\int^+$ *x*. *emeasure* (*distr x N f*) *A ∂M* **using** *A-in-N*
proof (*intro nn-integral-cong*, *subst emeasure-distr*)
fix *M'* **assume** *M'* ∈ *space M*
from *assms* **have** *space M* = *space* (*subprob-algebra R*)
using *sets-eq-imp-space-eq* **by** *blast*
with ‹*M'* ∈ *space M*› **have** [*simp*]: *sets M'* = *sets R* **using** *space-subprob-algebra* **by** *blast*
show *f* ∈ *measurable M' N* **by** (*simp cong*: *measurable-cong-sets add*: *assms*)
have *space M'* = *space R* **by** (*rule sets-eq-imp-space-eq*) *simp*
thus *emeasure M'* (*f* −` *A* ∩ *space R*) = *emeasure M'* (*f* −` *A* ∩ *space M'*) **by** *simp*
qed

also have (λ*M*. *distr M N f*) ∈ *measurable M* (*subprob-algebra N*)
by (*simp cong*: *measurable-cong-sets add*: *assms measurable-distr*)
hence ($\int^+$ *x*. *emeasure* (*distr x N f*) *A ∂M*) =
emeasure (*join* (*distr M* (*subprob-algebra N*) (λ*M*. *distr M N f*))) *A*
by (*simp-all add*: *emeasure-join assms A-in-N nn-integral-distr measurable-emeasure-subprob-algebra*)
finally show *emeasure ?r A* = *emeasure ?l A* **..**
qed *simp*

definition *bind* :: *'a measure* ⇒ (*'a* ⇒ *'b measure*) ⇒ *'b measure* **where**
bind M f = (*if space M* = {} *then count-space* {} *else*
join (*distr M* (*subprob-algebra* (*f* (*SOME x*. *x* ∈ *space M*))) *f*))

adhoc-overloading *Monad-Syntax.bind bind*

lemma *bind-empty*:
space M = {} ⟹ *bind M f* = *count-space* {}
by (*simp add*: *bind-def*)

lemma *bind-nonempty*:
space M ≠ {} ⟹ *bind M f* = *join* (*distr M* (*subprob-algebra* (*f* (*SOME x*. *x* ∈ *space M*))) *f*)
by (*simp add*: *bind-def*)

lemma *sets-bind-empty*: *sets M* = {} ⟹ *sets* (*bind M f*) = {{}}
by (*auto simp*: *bind-def*)

lemma *space-bind-empty*: *space M* = {} ⟹ *space* (*bind M f*) = {}
by (*simp add*: *bind-def*)

lemma *sets-bind*[*simp*, *measurable-cong*]:
assumes *f*: $\bigwedge x.\ x \in space\ M \Longrightarrow sets\ (f\ x) = sets\ N$ **and** *M*: $space\ M \neq \{\}$
shows $sets\ (bind\ M\ f) = sets\ N$
using *f* [*of SOME x. x ∈ space M*] **by** (*simp add*: *bind-nonempty M some-in-eq*)

lemma *space-bind*[*simp*]:
assumes $\bigwedge x.\ x \in space\ M \Longrightarrow sets\ (f\ x) = sets\ N$ **and** $space\ M \neq \{\}$
shows $space\ (bind\ M\ f) = space\ N$
using *assms* **by** (*intro sets-eq-imp-space-eq sets-bind*)

lemma *bind-cong-All*:
assumes $\forall x \in space\ M.\ f\ x = g\ x$
shows $bind\ M\ f = bind\ M\ g$
proof (*cases space M = {}*)
assume $space\ M \neq \{\}$
hence $(SOME\ x.\ x \in space\ M) \in space\ M$ **by** (*rule-tac someI-ex*) *blast*
with *assms* **have** $f\ (SOME\ x.\ x \in space\ M) = g\ (SOME\ x.\ x \in space\ M)$ **by** *blast*
with ‹$space\ M \neq \{\}$› **and** *assms* **show** *?thesis* **by** (*simp add*: *bind-nonempty cong*: *distr-cong*)
qed (*simp add*: *bind-empty*)

lemma *bind-cong*:
$M = N \Longrightarrow (\bigwedge x.\ x \in space\ M \Longrightarrow f\ x = g\ x) \Longrightarrow bind\ M\ f = bind\ N\ g$
using *bind-cong-All*[*of M f g*] **by** *auto*

lemma *bind-nonempty'*:
assumes $f \in measurable\ M\ (subprob\text{-}algebra\ N)$ $x \in space\ M$
shows $bind\ M\ f = join\ (distr\ M\ (subprob\text{-}algebra\ N)\ f)$
using *assms*
apply (*subst bind-nonempty*, *blast*)
apply (*subst subprob-algebra-cong*[*OF sets-kernel*[*OF assms*(*1*) *someI-ex*]], *blast*)
apply (*simp add*: *subprob-algebra-cong*[*OF sets-kernel*[*OF assms*]])
done

lemma *bind-nonempty''*:
assumes $f \in measurable\ M\ (subprob\text{-}algebra\ N)$ $space\ M \neq \{\}$
shows $bind\ M\ f = join\ (distr\ M\ (subprob\text{-}algebra\ N)\ f)$
using *assms* **by** (*auto intro*: *bind-nonempty'*)

lemma *emeasure-bind*:
$[\![space\ M \neq \{\}; f \in measurable\ M\ (subprob\text{-}algebra\ N); X \in sets\ N]\!]$
$\Longrightarrow emeasure\ (M \gg\!= f)\ X = \int^+ x.\ emeasure\ (f\ x)\ X\ \partial M$
by (*simp add*: *bind-nonempty'' emeasure-join nn-integral-distr measurable-emeasure-subprob-algebra*)

lemma *nn-integral-bind*:
assumes *f*: $f \in borel\text{-}measurable\ B$
assumes *N*: $N \in measurable\ M\ (subprob\text{-}algebra\ B)$
shows $(\int^+ x.\ f\ x\ \partial(M \gg\!= N)) = (\int^+ x.\ \int^+ y.\ f\ y\ \partial N\ x\ \partial M)$

proof *cases*
assume *M*: *space M* ≠ {} **show** *?thesis*
unfolding *bind-nonempty'*[*OF N M*] *nn-integral-join*[*OF f sets-distr*]
by (*rule nn-integral-distr*[*OF N*])
(*simp add*: *f nn-integral-measurable-subprob-algebra*)
qed (*simp add*: *bind-empty space-empty*[*of M*] *nn-integral-count-space*)

lemma *AE-bind*:
assumes *N*[*measurable*]: *N* ∈ *measurable M* (*subprob-algebra B*)
assumes *P*[*measurable*]: *Measurable.pred B P*
shows (*AE x in M* ⤜ *N*. *P x*) ⟷ (*AE x in M*. *AE y in N x*. *P y*)
proof *cases*
assume *M*: *space M* = {} **show** *?thesis*
unfolding *bind-empty*[*OF M*] **unfolding** *space-empty*[*OF M*] **by** (*simp add*: *AE-count-space*)
next
assume *M*: *space M* ≠ {}
note *sets-kernel*[*OF N*, *simp*]
have ∗: (∫⁺*x*. *indicator* {*x*. ¬ *P x*} *x* ∂(*M* ⤜ *N*)) = (∫⁺*x*. *indicator* {*x*∈*space B*. ¬ *P x*} *x* ∂(*M* ⤜ *N*))
by (*intro nn-integral-cong*) (*simp add*: *space-bind*[*OF - M*] *split*: *split-indicator*)

have (*AE x in M* ⤜ *N*. *P x*) ⟷ (∫⁺ *x*. *integral*^{N} (*N x*) (*indicator* {*x* ∈ *space B*. ¬ *P x*}) ∂*M*) = *0*
by (*simp add*: *AE-iff-nn-integral sets-bind*[*OF - M*] *space-bind*[*OF - M*] ∗ *nn-integral-bind*[**where** *B*=*B*]
del: *nn-integral-indicator*)
also have ... = (*AE x in M*. *AE y in N x*. *P y*)
apply (*subst nn-integral-0-iff-AE*)
apply (*rule measurable-compose*[*OF N nn-integral-measurable-subprob-algebra*])
apply *measurable*
apply (*intro eventually-subst AE-I2*)
apply (*auto simp add*: *subprob-measurableD*(*1*)[*OF N*]
intro!: *AE-iff-measurable*[*symmetric*])
done
finally show *?thesis* **.**
qed

lemma *measurable-bind'*:
assumes *M1*: *f* ∈ *measurable M* (*subprob-algebra N*) **and**
M2: *case-prod g* ∈ *measurable* (*M* ⨂$_M$ *N*) (*subprob-algebra R*)
shows (λ*x*. *bind* (*f x*) (*g x*)) ∈ *measurable M* (*subprob-algebra R*)
proof (*subst measurable-cong*)
fix *x* **assume** *x-in-M*: *x* ∈ *space M*
with *assms* **have** *space* (*f x*) ≠ {}
by (*blast dest*: *subprob-space-kernel subprob-space.subprob-not-empty*)
moreover from *M2 x-in-M* **have** *g x* ∈ *measurable* (*f x*) (*subprob-algebra R*)
by (*subst measurable-cong-sets*[*OF sets-kernel*[*OF M1 x-in-M*] *refl*])
(*auto dest*: *measurable-Pair2*)

ultimately show *bind* (*f x*) (*g x*) = *join* (*distr* (*f x*) (*subprob-algebra R*) (*g x*))
by (*simp-all add*: *bind-nonempty''*)
next
show (λ*w*. *join* (*distr* (*f w*) (*subprob-algebra R*) (*g w*))) ∈ *measurable M* (*subprob-algebra R*)
apply (*rule measurable-compose*[*OF* - *measurable-join*])
apply (*rule measurable-distr2*[*OF M2 M1*])
done
qed

lemma *measurable-bind*[*measurable* (*raw*)]:
assumes *M1*: *f* ∈ *measurable M* (*subprob-algebra N*) **and**
M2: (λ*x*. *g* (*fst x*) (*snd x*)) ∈ *measurable* (*M* ⨂$_M$ *N*) (*subprob-algebra R*)
shows (λ*x*. *bind* (*f x*) (*g x*)) ∈ *measurable M* (*subprob-algebra R*)
using *assms* **by** (*auto intro*: *measurable-bind' simp*: *measurable-split-conv*)

lemma *measurable-bind2*:
assumes *f* ∈ *measurable M* (*subprob-algebra N*) **and** *g* ∈ *measurable N* (*subprob-algebra R*)
shows (λ*x*. *bind* (*f x*) *g*) ∈ *measurable M* (*subprob-algebra R*)
using *assms* **by** (*intro measurable-bind' measurable-const*) *auto*

lemma *subprob-space-bind*:
assumes *subprob-space M f* ∈ *measurable M* (*subprob-algebra N*)
shows *subprob-space* (*M* ⤜ *f*)
proof (*rule subprob-space-kernel*[*of* λ*x*. *x* ⤜ *f*])
show (λ*x*. *x* ⤜ *f*) ∈ *measurable* (*subprob-algebra M*) (*subprob-algebra N*)
by (*rule measurable-bind*, *rule measurable-ident-sets*, *rule refl*,
rule measurable-compose[*OF measurable-snd assms*(*2*)])
from *assms*(*1*) **show** *M* ∈ *space* (*subprob-algebra M*)
by (*simp add*: *space-subprob-algebra*)
qed

lemma
fixes *f* :: - ⇒ *real*
assumes *f-measurable* [*measurable*]: *f* ∈ *borel-measurable K*
and *f-bounded*: ⋀*x*. *x* ∈ *space K* ⟹ |*f x*| ≤ *B*
and *N* [*measurable*]: *N* ∈ *measurable M* (*subprob-algebra K*)
and *fin*: *finite-measure M*
and *M-bounded*: *AE x in M*. *emeasure* (*N x*) (*space* (*N x*)) ≤ *ennreal B'*
shows *integrable-bind*: *integrable* (*bind M N*) *f* (**is** *?integrable*)
and *integral-bind*: *integral*^{L} (*bind M N*) *f* = ∫ *x*. *integral*^{L} (*N x*) *f* ∂*M* (**is** *?integral*)
proof(*case-tac* [!] *space M* = {})
assume [*simp*]: *space M* ≠ {}
interpret *finite-measure M* **by**(*rule fin*)

have *integrable* (*join* (*distr M* (*subprob-algebra K*) *N*)) *f*
using *f-measurable f-bounded*

by(*rule integrable-join*[**where** *B′=B‴*])(*simp-all add*: *finite-measure-distr AE-distr-iff M-bounded*)
then show *?integrable* **by**(*simp add*: *bind-nonempty″*[**where** *N=K*])

have $integral^L$ (*join* (*distr M* (*subprob-algebra K*) *N*)) *f* = ∫ *M′*. $integral^L$ *M′ f* ∂*distr M* (*subprob-algebra K*) *N*
using *f-measurable f-bounded*
by(*rule integral-join*[**where** *B′=B‴*])(*simp-all add*: *finite-measure-distr AE-distr-iff M-bounded*)
also have ... = ∫ *x*. $integral^L$ (*N x*) *f* ∂*M*
by(*rule integral-distr*)(*simp-all add*: *integral-measurable-subprob-algebra*[*OF* -])
finally show *?integral* **by**(*simp add*: *bind-nonempty″*[**where** *N=K*])
qed(*simp-all add*: *bind-def integrable-count-space lebesgue-integral-count-space-finite Bochner-Integration.integral-empty*)

lemma (**in** *prob-space*) *prob-space-bind*:
assumes *ae*: *AE x in M*. *prob-space* (*N x*)
and *N*[*measurable*]: *N* ∈ *measurable M* (*subprob-algebra S*)
shows *prob-space* (*M* ⤜ *N*)
proof
have *emeasure* (*M* ⤜ *N*) (*space* (*M* ⤜ *N*)) = (∫⁺*x*. *emeasure* (*N x*) (*space* (*N x*)) ∂*M*)
by (*subst emeasure-bind*[**where** *N=S*])
(*auto simp*: *not-empty space-bind*[*OF sets-kernel*] *subprob-measurableD*[*OF N*] *intro*!: *nn-integral-cong*)
also have ... = (∫⁺*x*. *1* ∂*M*)
using *ae* **by** (*intro nn-integral-cong-AE*, *eventually-elim*) (*rule prob-space.emeasure-space-1*)
finally show *emeasure* (*M* ⤜ *N*) (*space* (*M* ⤜ *N*)) = *1*
by (*simp add*: *emeasure-space-1*)
qed

lemma (**in** *subprob-space*) *bind-in-space*:
A ∈ *measurable M* (*subprob-algebra N*) ⟹ (*M* ⤜ *A*) ∈ *space* (*subprob-algebra N*)
by (*auto simp add*: *space-subprob-algebra subprob-not-empty sets-kernel intro*!: *subprob-space-bind*)
unfold-locales

lemma (**in** *subprob-space*) *measure-bind*:
assumes *f*: *f* ∈ *measurable M* (*subprob-algebra N*) **and** *X*: *X* ∈ *sets N*
shows *measure* (*M* ⤜ *f*) *X* = ∫ *x*. *measure* (*f x*) *X* ∂*M*
proof −
interpret *Mf*: *subprob-space M* ⤜ *f*
by (*rule subprob-space-bind*[*OF* - *f*]) *unfold-locales*

{ fix *x* **assume** *x* ∈ *space M*
from *f*[*THEN measurable-space*, *OF this*] **interpret** *subprob-space f x*
by (*simp add*: *space-subprob-algebra*)
have *emeasure* (*f x*) *X* = *ennreal* (*measure* (*f x*) *X*) *measure* (*f x*) *X* ≤ *1*

by (*auto simp*: *emeasure-eq-measure subprob-measure-le-1*) **}**
note *this*[*simp*]

have *emeasure* (*M* ⤜ *f*) *X* = ∫⁺*x*. *emeasure* (*f x*) *X* ∂*M*
using *subprob-not-empty f X* **by** (*rule emeasure-bind*)
also have ... = ∫⁺*x*. *ennreal* (*measure* (*f x*) *X*) ∂*M*
by (*intro nn-integral-cong*) *simp*
also have ... = ∫ *x*. *measure* (*f x*) *X* ∂*M*
by (*intro nn-integral-eq-integral integrable-const-bound*[**where** *B=1*]
measure-measurable-subprob-algebra2[*OF - f*] *pair-measureI X*)
(*auto simp*: *measure-nonneg*)
finally show *?thesis*
by (*simp add*: *Mf.emeasure-eq-measure measure-nonneg integral-nonneg*)
qed

lemma *emeasure-bind-const*:
space M ≠ {} ⟹ *X* ∈ *sets N* ⟹ *subprob-space N* ⟹
emeasure (*M* ⤜ (λ*x*. *N*)) *X* = *emeasure N X* ∗ *emeasure M* (*space M*)
by (*simp add*: *bind-nonempty emeasure-join nn-integral-distr*
space-subprob-algebra measurable-emeasure-subprob-algebra)

lemma *emeasure-bind-const′*:
assumes *subprob-space M subprob-space N*
shows *emeasure* (*M* ⤜ (λ*x*. *N*)) *X* = *emeasure N X* ∗ *emeasure M* (*space M*)
using *assms*
proof (*case-tac X* ∈ *sets N*)
fix *X* **assume** *X* ∈ *sets N*
thus *emeasure* (*M* ⤜ (λ*x*. *N*)) *X* = *emeasure N X* ∗ *emeasure M* (*space M*)
using *assms*
by (*subst emeasure-bind-const*)
(*simp-all add*: *subprob-space.subprob-not-empty subprob-space.emeasure-space-le-1*)
next
fix *X* **assume** *X* ∉ *sets N*
with *assms* **show** *emeasure* (*M* ⤜ (λ*x*. *N*)) *X* = *emeasure N X* ∗ *emeasure M*
(*space M*)
by (*simp add*: *sets-bind*[*of - - N*] *subprob-space.subprob-not-empty*
space-subprob-algebra emeasure-notin-sets)
qed

lemma *emeasure-bind-const-prob-space*:
assumes *prob-space M subprob-space N*
shows *emeasure* (*M* ⤜ (λ*x*. *N*)) *X* = *emeasure N X*
using *assms* **by** (*simp add*: *emeasure-bind-const′ prob-space-imp-subprob-space*
prob-space.emeasure-space-1)

lemma *bind-return*:
assumes *f* ∈ *measurable M* (*subprob-algebra N*) **and** *x* ∈ *space M*
shows *bind* (*return M x*) *f* = *f x*
using *sets-kernel*[*OF assms*] *assms*

by (*simp-all add*: *distr-return join-return subprob-space-kernel bind-nonempty'*
cong: *subprob-algebra-cong*)

lemma *bind-return'*:
shows *bind M* (*return M*) = *M*
by (*cases space M* = {})
(*simp-all add*: *bind-empty space-empty*[*symmetric*] *bind-nonempty join-return'*
cong: *subprob-algebra-cong*)

lemma *distr-bind*:
assumes *N*: *N* ∈ *measurable M* (*subprob-algebra K*) *space M* ≠ {}
assumes *f*: *f* ∈ *measurable K R*
shows *distr* (*M* ⤜ *N*) *R f* = (*M* ⤜ (λ*x*. *distr* (*N x*) *R f*))
unfolding *bind-nonempty''*[*OF N*]
apply (*subst bind-nonempty''*[*OF measurable-compose*[*OF N*(*1*) *measurable-distr*]
N(*2*)])
apply (*rule f*)
apply (*simp add*: *join-distr-distr*[*OF - f*, *symmetric*])
apply (*subst distr-distr*[*OF measurable-distr*, *OF f N*(*1*)])
apply (*simp add*: *comp-def*)
done

lemma *bind-distr*:
assumes *f*[*measurable*]: *f* ∈ *measurable M X*
assumes *N*[*measurable*]: *N* ∈ *measurable X* (*subprob-algebra K*) **and** *space M* ≠
{}
shows (*distr M X f* ⤜ *N*) = (*M* ⤜ (λ*x*. *N* (*f x*)))
proof −
have *space X* ≠ {} *space M* ≠ {}
using ‹*space M* ≠ {}› *f*[*THEN measurable-space*] **by** *auto*
then show *?thesis*
by (*simp add*: *bind-nonempty''*[**where** *N*=*K*] *distr-distr comp-def*)
qed

lemma *bind-count-space-singleton*:
assumes *subprob-space* (*f x*)
shows *count-space* {*x*} ⤜ *f* = *f x*
proof−
have *A*: ⋀*A*. *A* ⊆ {*x*} ⟹ *A* = {} ∨ *A* = {*x*} **by** *auto*
have *count-space* {*x*} = *return* (*count-space* {*x*}) *x*
by (*intro measure-eqI*) (*auto dest*: *A*)
also have ... ⤜ *f* = *f x*
by (*subst bind-return*[*of - - f x*]) (*auto simp*: *space-subprob-algebra assms*)
finally show *?thesis* **.**
qed

lemma *restrict-space-bind*:
assumes *N*: *N* ∈ *measurable M* (*subprob-algebra K*)
assumes *space M* ≠ {}

assumes X[*simp*]: $X \in$ *sets* K $X \neq \{\}$
shows *restrict-space* (*bind* M N) X = *bind* M (λx. *restrict-space* (N x) X)
proof (*rule measure-eqI*)
note *N-sets* = *sets-bind*[*OF sets-kernel*[*OF* N] *assms*(*2*), *simp*]
note *N-space* = *sets-eq-imp-space-eq*[*OF N-sets*, *simp*]
show *sets* (*restrict-space* (*bind* M N) X) = *sets* (*bind* M (λx. *restrict-space* (N x) X))
by (*simp add*: *sets-restrict-space assms*(*2*) *sets-bind*[*OF sets-kernel*[*OF restrict-space-measurable*[*OF assms*(*4*,*3*,*1*)]]])
fix A **assume** $A \in$ *sets* (*restrict-space* ($M \gg\!= N$) X)
with X **have** $A \in$ *sets* K $A \subseteq X$
by (*auto simp*: *sets-restrict-space*)
then show *emeasure* (*restrict-space* ($M \gg\!= N$) X) A = *emeasure* ($M \gg\!= (\lambda x.$ *restrict-space* (N x) X)) A
using *assms*
apply (*subst emeasure-restrict-space*)
apply (*simp-all add*: *emeasure-bind*[*OF assms*(*2*,*1*)])
apply (*subst emeasure-bind*[*OF* - *restrict-space-measurable*[*OF* - - *N*]])
apply (*auto simp*: *sets-restrict-space emeasure-restrict-space space-subprob-algebra*
intro!: *nn-integral-cong dest*!: *measurable-space*)
done
qed

lemma *bind-restrict-space*:
assumes A: $A \cap$ *space* $M \neq \{\}$ $A \cap$ *space* $M \in$ *sets* M
and f: $f \in$ *measurable* (*restrict-space* M A) (*subprob-algebra* N)
shows *restrict-space* M $A \gg\!= f$ = $M \gg\!= (\lambda x.$ *if* $x \in A$ *then* f x *else null-measure* (f (*SOME* x. $x \in A \wedge x \in$ *space* M)))
(**is** *?lhs* = *?rhs* **is** - = $M \gg\!=$ *?f*)
proof −
let *?P* = $\lambda x.$ $x \in A \wedge x \in$ *space* M
let *?x* = *Eps ?P*
let *?c* = *null-measure* (f *?x*)
from A **have** *?P ?x* **by**−(*rule someI-ex*, *blast*)
hence *?x* ∈ *space* (*restrict-space* M A) **by**(*simp add*: *space-restrict-space*)
with f **have** f *?x* ∈ *space* (*subprob-algebra* N) **by**(*rule measurable-space*)
hence *sps*: *subprob-space* (f *?x*)
and *sets*: *sets* (f *?x*) = *sets* N
by(*simp-all add*: *space-subprob-algebra*)
have *space* (f *?x*) ≠ {} **using** *sps* **by**(*rule subprob-space.subprob-not-empty*)
moreover have *sets ?c* = *sets* N **by**(*simp add*: *null-measure-def*)(*simp add*: *sets*)
ultimately have *c*: *?c* ∈ *space* (*subprob-algebra* N)
by(*simp add*: *space-subprob-algebra subprob-space-null-measure*)
from f A c **have** f': *?f* ∈ *measurable* M (*subprob-algebra* N)
by(*simp add*: *measurable-restrict-space-iff*)

from A **have** [*simp*]: *space* $M \neq \{\}$ **by** *blast*

have *?lhs* = *join* (*distr* (*restrict-space M A*) (*subprob-algebra N*) *f*)
using *assms* **by**(*simp add*: *space-restrict-space bind-nonempty''*)
also have ... = *join* (*distr M* (*subprob-algebra N*) *?f*)
by(*rule measure-eqI*)(*auto simp add*: *emeasure-join nn-integral-distr nn-integral-restrict-space f f' A intro*: *nn-integral-cong*)
also have ... = *?rhs* **using** *f'* **by**(*simp add*: *bind-nonempty''*)
finally show *?thesis* **.**
qed

lemma *bind-const'*: ⟦*prob-space M*; *subprob-space N*⟧ ⟹ *M* ⤜ (λ*x*. *N*) = *N*
by (*intro measure-eqI*)
(*simp-all add*: *space-subprob-algebra prob-space.not-empty emeasure-bind-const-prob-space*)

lemma *bind-return-distr*:
space M ≠ {} ⟹ *f* ∈ *measurable M N* ⟹ *bind M* (*return N* ∘ *f*) = *distr M N f*
apply (*simp add*: *bind-nonempty*)
apply (*subst subprob-algebra-cong*)
apply (*rule sets-return*)
apply (*subst distr-distr*[*symmetric*])
apply (*auto intro*!: *return-measurable simp*: *distr-distr*[*symmetric*] *join-return'*)
done

lemma *bind-return-distr'*:
space M ≠ {} ⟹ *f* ∈ *measurable M N* ⟹ *bind M* (λ*x*. *return N* (*f x*)) = *distr M N f*
using *bind-return-distr*[*of M f N*] **by** (*simp add*: *comp-def*)

lemma *bind-assoc*:
fixes *f* :: *'a* ⇒ *'b measure* **and** *g* :: *'b* ⇒ *'c measure*
assumes *M1*: *f* ∈ *measurable M* (*subprob-algebra N*) **and** *M2*: *g* ∈ *measurable N* (*subprob-algebra R*)
shows *bind* (*bind M f*) *g* = *bind M* (λ*x*. *bind* (*f x*) *g*)
proof (*cases space M* = {})
assume [*simp*]: *space M* ≠ {}
from *assms* **have** [*simp*]: *space N* ≠ {} *space R* ≠ {}
by (*auto simp*: *measurable-empty-iff space-subprob-algebra-empty-iff*)
from *assms* **have** *sets-fx*[*simp*]: ⋀*x*. *x* ∈ *space M* ⟹ *sets* (*f x*) = *sets N*
by (*simp add*: *sets-kernel*)
have *ex-in*: ⋀*A*. *A* ≠ {} ⟹ ∃ *x*. *x* ∈ *A* **by** *blast*
note *sets-some*[*simp*] = *sets-kernel*[*OF M1 someI-ex*[*OF ex-in*[*OF* ‹*space M* ≠ {}›]]]
sets-kernel[*OF M2 someI-ex*[*OF ex-in*[*OF* ‹*space N* ≠ {}›]]]
note *space-some*[*simp*] = *sets-eq-imp-space-eq*[*OF this*(*1*)] *sets-eq-imp-space-eq*[*OF this*(*2*)]

have *bind M* (λ*x*. *bind* (*f x*) *g*) =
join (*distr M* (*subprob-algebra R*) (*join* ∘ (λ*x*. (*distr x* (*subprob-algebra R*) *g*)) ∘ *f*))

by (*simp add*: *sets-eq-imp-space-eq*[*OF sets-fx*] *bind-nonempty o-def*
cong: *subprob-algebra-cong distr-cong*)
also have *distr M* (*subprob-algebra R*) (*join* ∘ (λ*x*. (*distr x* (*subprob-algebra R*) *g*)) ∘ *f*) =
distr (*distr* (*distr M* (*subprob-algebra N*) *f*)
(*subprob-algebra* (*subprob-algebra R*))
(λ*x*. *distr x* (*subprob-algebra R*) *g*))
(*subprob-algebra R*) *join*
apply (*subst distr-distr*,
(*blast intro*: *measurable-comp measurable-distr measurable-join M1 M2*)+)+
apply (*simp add*: *o-assoc*)
done
also have *join* ... = *bind* (*bind M f*) *g*
by (*simp add*: *join-assoc join-distr-distr M2 bind-nonempty cong*: *subprob-algebra-cong*)
finally show *?thesis* **..**
qed (*simp add*: *bind-empty*)

lemma *double-bind-assoc*:
assumes *Mg*: *g* ∈ *measurable N* (*subprob-algebra N'*)
assumes *Mf*: *f* ∈ *measurable M* (*subprob-algebra M'*)
assumes *Mh*: *case-prod h* ∈ *measurable* (*M* ⨂*M M'*) *N*
shows *do* {*x* ← *M*; *y* ← *f x*; *g* (*h x y*)} = *do* {*x* ← *M*; *y* ← *f x*; *return N* (*h x y*)} ⤜ *g*
proof−
have *do* {*x* ← *M*; *y* ← *f x*; *return N* (*h x y*)} ⤜ *g* =
do {*x* ← *M*; *do* {*y* ← *f x*; *return N* (*h x y*)} ⤜ *g*}
using *Mh* **by** (*auto intro*!: *bind-assoc measurable-bind'*[*OF Mf*] *Mf Mg*
measurable-compose[*OF* - *return-measurable*] *simp*: *measurable-split-conv*)
also from *Mh* **have** ⋀*x*. *x* ∈ *space M* ⟹ *h x* ∈ *measurable M' N* **by** *measurable*
hence *do* {*x* ← *M*; *do* {*y* ← *f x*; *return N* (*h x y*)} ⤜ *g*} =
do {*x* ← *M*; *y* ← *f x*; *return N* (*h x y*) ⤜ *g*}
apply (*intro ballI bind-cong refl bind-assoc*)
apply (*subst measurable-cong-sets*[*OF sets-kernel*[*OF Mf*] *refl*], *simp*)
apply (*rule measurable-compose*[*OF* - *return-measurable*], *auto intro*: *Mg*)
done
also have ⋀*x*. *x* ∈ *space M* ⟹ *space* (*f x*) = *space M'*
by (*intro sets-eq-imp-space-eq sets-kernel*[*OF Mf*])
with *measurable-space*[*OF Mh*]
have *do* {*x* ← *M*; *y* ← *f x*; *return N* (*h x y*) ⤜ *g*} = *do* {*x* ← *M*; *y* ← *f x*; *g* (*h x y*)}
by (*intro ballI bind-cong bind-return*[*OF Mg*]) (*auto simp*: *space-pair-measure*)
finally show *?thesis* **..**
qed

lemma (**in** *prob-space*) *M-in-subprob*[*measurable* (*raw*)]: *M* ∈ *space* (*subprob-algebra M*)
by (*simp add*: *space-subprob-algebra*) *unfold-locales*

lemma (**in** *pair-prob-space*) *pair-measure-eq-bind*:
(*M1* $\bigotimes_M$ *M2*) = (*M1* $\gg\!=$ (λ*x*. *M2* $\gg\!=$ (λ*y*. *return* (*M1* $\bigotimes_M$ *M2*) (*x*, *y*))))
proof (*rule measure-eqI*)
have *ps-M2*: *prob-space M2* **by** *unfold-locales*
note *return-measurable*[*measurable*]
show *sets* (*M1* $\bigotimes_M$ *M2*) = *sets* (*M1* $\gg\!=$ (λ*x*. *M2* $\gg\!=$ (λ*y*. *return* (*M1* $\bigotimes_M$ *M2*) (*x*, *y*))))
by (*simp-all add*: *M1.not-empty M2.not-empty*)
fix *A* **assume** [*measurable*]: *A* ∈ *sets* (*M1* $\bigotimes_M$ *M2*)
show *emeasure* (*M1* $\bigotimes_M$ *M2*) *A* = *emeasure* (*M1* $\gg\!=$ (λ*x*. *M2* $\gg\!=$ (λ*y*. *return* (*M1* $\bigotimes_M$ *M2*) (*x*, *y*)))) *A*
by (*auto simp*: *M2.emeasure-pair-measure M1.not-empty M2.not-empty emeasure-bind*[**where** *N*=*M1* $\bigotimes_M$ *M2*]
intro!: *nn-integral-cong*)
qed

lemma (**in** *pair-prob-space*) *bind-rotate*:
assumes *C*[*measurable*]: (λ(*x*, *y*). *C x y*) ∈ *measurable* (*M1* $\bigotimes_M$ *M2*) (*subprob-algebra N*)
shows (*M1* $\gg\!=$ (λ*x*. *M2* $\gg\!=$ (λ*y*. *C x y*))) = (*M2* $\gg\!=$ (λ*y*. *M1* $\gg\!=$ (λ*x*. *C x y*)))
proof −
interpret *swap*: *pair-prob-space M2 M1* **by** *unfold-locales*
note *measurable-bind*[**where** *N*=*M2*, *measurable*]
note *measurable-bind*[**where** *N*=*M1*, *measurable*]
have [*simp*]: *M1* ∈ *space* (*subprob-algebra M1*) *M2* ∈ *space* (*subprob-algebra M2*)
by (*auto simp*: *space-subprob-algebra*) *unfold-locales*
have (*M1* $\gg\!=$ (λ*x*. *M2* $\gg\!=$ (λ*y*. *C x y*))) =
(*M1* $\gg\!=$ (λ*x*. *M2* $\gg\!=$ (λ*y*. *return* (*M1* $\bigotimes_M$ *M2*) (*x*, *y*)))) $\gg\!=$ (λ(*x*, *y*). *C x y*)
by (*auto intro*!: *bind-cong simp*: *bind-return*[**where** *N*=*N*] *space-pair-measure bind-assoc*[**where** *N*=*M1* $\bigotimes_M$ *M2* **and** *R*=*N*])
also have ... = (*distr* (*M2* $\bigotimes_M$ *M1*) (*M1* $\bigotimes_M$ *M2*) (λ(*x*, *y*). (*y*, *x*))) $\gg\!=$ (λ(*x*, *y*). *C x y*)
unfolding *pair-measure-eq-bind*[*symmetric*] *distr-pair-swap*[*symmetric*] **..**
also have ... = (*M2* $\gg\!=$ (λ*x*. *M1* $\gg\!=$ (λ*y*. *return* (*M2* $\bigotimes_M$ *M1*) (*x*, *y*)))) $\gg\!=$ (λ(*y*, *x*). *C x y*)
unfolding *swap.pair-measure-eq-bind*[*symmetric*]
by (*auto simp add*: *space-pair-measure M1.not-empty M2.not-empty bind-distr*[*OF - C*] *intro*!: *bind-cong*)
also have ... = (*M2* $\gg\!=$ (λ*y*. *M1* $\gg\!=$ (λ*x*. *C x y*)))
by (*auto intro*!: *bind-cong simp*: *bind-return*[**where** *N*=*N*] *space-pair-measure bind-assoc*[**where** *N*=*M2* $\bigotimes_M$ *M1* **and** *R*=*N*])
finally show *?thesis* **.**
qed

lemma *bind-return''*: *sets M* = *sets N* ⟹ *M* $\gg\!=$ *return N* = *M*
by (*cases space M* = {})
(*simp-all add*: *bind-empty space-empty*[*symmetric*] *bind-nonempty join-return'*

cong: *subprob-algebra-cong*)

lemma (**in** *prob-space*) *distr-const*[*simp*]:
c ∈ *space N* ⟹ *distr M N* (λ*x*. *c*) = *return N c*
by (*rule measure-eqI*) (*auto simp*: *emeasure-distr emeasure-space-1*)

lemma *return-count-space-eq-density*:
return (*count-space M*) *x* = *density* (*count-space M*) (*indicator* {*x*})
by (*rule measure-eqI*)
(*auto simp*: *indicator-inter-arith*[*symmetric*] *emeasure-density split*: *split-indicator*)

lemma *null-measure-in-space-subprob-algebra* [*simp*]:
null-measure M ∈ *space* (*subprob-algebra M*) ⟷ *space M* ≠ {}
by(*simp add*: *space-subprob-algebra subprob-space-null-measure-iff*)

7.4 Giry monad on probability spaces

definition *prob-algebra* :: *'a measure* ⇒ *'a measure measure* **where**
prob-algebra K = *restrict-space* (*subprob-algebra K*) {*M*. *prob-space M*}

lemma *space-prob-algebra*: *space* (*prob-algebra M*) = {*N*. *sets N* = *sets M* ∧ *prob-space N*}
unfolding *prob-algebra-def* **by** (*auto simp*: *space-subprob-algebra space-restrict-space prob-space-imp-subprob-space*)

lemma *measurable-measure-prob-algebra*[*measurable*]:
a ∈ *sets A* ⟹ (λ*M*. *Sigma-Algebra.measure M a*) ∈ *prob-algebra A* →$_M$ *borel*
unfolding *prob-algebra-def* **by** (*intro measurable-restrict-space1 measurable-measure-subprob-algebra*)

lemma *measurable-prob-algebraD*:
f ∈ *N* →$_M$ *prob-algebra M* ⟹ *f* ∈ *N* →$_M$ *subprob-algebra M*
unfolding *prob-algebra-def measurable-restrict-space2-iff* **by** *auto*

lemma *measure-measurable-prob-algebra2*:
Sigma (*space M*) *A* ∈ *sets* (*M* ⨂$_M$ *N*) ⟹ *L* ∈ *M* →$_M$ *prob-algebra N* ⟹
(λ*x*. *Sigma-Algebra.measure* (*L x*) (*A x*)) ∈ *borel-measurable M*
using *measure-measurable-subprob-algebra2*[*of M A N L*] **by** (*auto intro*: *measurable-prob-algebraD*)

lemma *measurable-prob-algebraI*:
(⋀*x*. *x* ∈ *space N* ⟹ *prob-space* (*f x*)) ⟹ *f* ∈ *N* →$_M$ *subprob-algebra M* ⟹
f ∈ *N* →$_M$ *prob-algebra M*
unfolding *prob-algebra-def* **by** (*intro measurable-restrict-space2*) *auto*

lemma *measurable-distr-prob-space*:
assumes *f*: *f* ∈ *M* →$_M$ *N*
shows (λ*M'*. *distr M' N f*) ∈ *prob-algebra M* →$_M$ *prob-algebra N*
unfolding *prob-algebra-def measurable-restrict-space2-iff*
proof (*intro conjI measurable-restrict-space1 measurable-distr f*)

show ($\lambda M'$. *distr* M' *N f*) ∈ *space* (*restrict-space* (*subprob-algebra M*) (*Collect prob-space*)) → *Collect prob-space*
using *f* **by** (*auto simp*: *space-restrict-space space-subprob-algebra intro*!: *prob-space.prob-space-distr*)
qed

lemma *measurable-return-prob-space*[*measurable*]: *return N* ∈ *N* →$_M$ *prob-algebra N*
by (*rule measurable-prob-algebraI*) (*auto simp*: *prob-space-return*)

lemma *measurable-distr-prob-space2*[*measurable* (*raw*)]:
assumes *f*: *g* ∈ *L* →$_M$ *prob-algebra M* ($\lambda(x, y)$. *f x y*) ∈ *L* ⨂$_M$ *M* →$_M$ *N*
shows (λx. *distr* (*g x*) *N* (*f x*)) ∈ *L* →$_M$ *prob-algebra N*
unfolding *prob-algebra-def measurable-restrict-space2-iff*
proof (*intro conjI measurable-restrict-space1 measurable-distr2*[**where** *M*=*M*] *f measurable-prob-algebraD*)
show (λx. *distr* (*g x*) *N* (*f x*)) ∈ *space L* → *Collect prob-space*
using *f subprob-measurableD*[*OF measurable-prob-algebraD*[*OF f*(*1*)]]
by (*auto simp*: *measurable-restrict-space2-iff prob-algebra-def*
intro!: *prob-space.prob-space-distr*)
qed

lemma *measurable-bind-prob-space*:
assumes *f*: *f* ∈ *M* →$_M$ *prob-algebra N* **and** *g*: *g* ∈ *N* →$_M$ *prob-algebra R*
shows (λx. *bind* (*f x*) *g*) ∈ *M* →$_M$ *prob-algebra R*
unfolding *prob-algebra-def measurable-restrict-space2-iff*
proof (*intro conjI measurable-restrict-space1 measurable-bind2*[**where** *N*=*N*] *f g measurable-prob-algebraD*)
show (λx. *f x* ⤜ *g*) ∈ *space M* → *Collect prob-space*
using *g f subprob-measurableD*[*OF measurable-prob-algebraD*[*OF f*]]
by (*auto simp*: *measurable-restrict-space2-iff prob-algebra-def*
intro!: *prob-space.prob-space-bind*[**where** *S*=*R*] *AE-I2*)
qed

lemma *measurable-bind-prob-space2*[*measurable* (*raw*)]:
assumes *f*: *f* ∈ *M* →$_M$ *prob-algebra N* **and** *g*: ($\lambda(x, y)$. *g x y*) ∈ (*M* ⨂$_M$ *N*) →$_M$ *prob-algebra R*
shows (λx. *bind* (*f x*) (*g x*)) ∈ *M* →$_M$ *prob-algebra R*
unfolding *prob-algebra-def measurable-restrict-space2-iff*
proof (*intro conjI measurable-restrict-space1 measurable-bind*[**where** *N*=*N*] *f g measurable-prob-algebraD*)
show (λx. *f x* ⤜ *g x*) ∈ *space M* → *Collect prob-space*
using *g f subprob-measurableD*[*OF measurable-prob-algebraD*[*OF f*]]
using *measurable-space*[*OF g*]
by (*auto simp*: *measurable-restrict-space2-iff prob-algebra-def space-pair-measure Pi-iff*
intro!: *prob-space.prob-space-bind*[**where** *S*=*R*] *AE-I2*)
qed (*insert g*, *simp*)

lemma *measurable-prob-algebra-generated*:
assumes *eq*: *sets N* = *sigma-sets* Ω *G* **and** *Int-stable G G* ⊆ *Pow* Ω
assumes *subsp*: ⋀*a*. *a* ∈ *space M* ⟹ *prob-space* (*K a*)
assumes *sets*: ⋀*a*. *a* ∈ *space M* ⟹ *sets* (*K a*) = *sets N*
assumes ⋀*A*. *A* ∈ *G* ⟹ (λ*a*. *emeasure* (*K a*) *A*) ∈ *borel-measurable M*
shows *K* ∈ *measurable M* (*prob-algebra N*)
unfolding *measurable-restrict-space2-iff prob-algebra-def*
proof
show *K* ∈ *M* →$_M$ *subprob-algebra N*
proof (*rule measurable-subprob-algebra-generated*[*OF assms*(*1*,*2*,*3*) - *assms*(*5*,*6*)])
fix *a* **assume** *a* ∈ *space M* **then show** *subprob-space* (*K a*)
using *subsp*[*of a*] **by** (*intro prob-space-imp-subprob-space*)
next
have (λ*a*. *emeasure* (*K a*) Ω) ∈ *borel-measurable M* ⟷ (λ*a*. *1*::*ennreal*) ∈ *borel-measurable M*
using *sets-eq-imp-space-eq*[*of sigma* Ω *G N*] ‹*G* ⊆ *Pow* Ω› *eq sets-eq-imp-space-eq*[*OF sets*]
prob-space.emeasure-space-1[*OF subsp*]
by (*intro measurable-cong*) *auto*
then show (λ*a*. *emeasure* (*K a*) Ω) ∈ *borel-measurable M* **by** *simp*
qed
qed (*insert subsp*, *auto*)

lemma *in-space-prob-algebra*:
x ∈ *space* (*prob-algebra M*) ⟹ *emeasure x* (*space M*) = *1*
unfolding *prob-algebra-def space-restrict-space space-subprob-algebra*
by (*auto dest*!: *prob-space.emeasure-space-1 sets-eq-imp-space-eq*)

lemma *prob-space-pair*:
assumes *prob-space M prob-space N* **shows** *prob-space* (*M* ⨂$_M$ *N*)
proof −
interpret *M*: *prob-space M* **by** *fact*
interpret *N*: *prob-space N* **by** *fact*
interpret *P*: *pair-prob-space M N* **proof qed**
show *?thesis*
by *unfold-locales*
qed

lemma *measurable-pair-prob*[*measurable*]:
f ∈ *M* →$_M$ *prob-algebra N* ⟹ *g* ∈ *M* →$_M$ *prob-algebra L* ⟹ (λ*x*. *f x* ⨂$_M$ *g x*) ∈ *M* →$_M$ *prob-algebra* (*N* ⨂$_M$ *L*)
unfolding *prob-algebra-def measurable-restrict-space2-iff*
by (*auto intro*!: *measurable-pair-measure prob-space-pair*)

lemma *emeasure-bind-prob-algebra*:
assumes *A*: *A* ∈ *space* (*prob-algebra N*)
assumes *B*: *B* ∈ *N* →$_M$ *prob-algebra L*
assumes *X*: *X* ∈ *sets L*
shows *emeasure* (*bind A B*) *X* = (∫$^+$*x*. *emeasure* (*B x*) *X* ∂*A*)

using *A B*
by (*intro emeasure-bind*[*OF - - X*])
(*auto simp*: *space-prob-algebra measurable-prob-algebraD cong*: *measurable-cong-sets intro*!: *prob-space.not-empty*)

lemma *prob-space-bind'*:
assumes *A*: $A \in$ *space* (*prob-algebra M*) **and** *B*: $B \in M \rightarrow_M$ *prob-algebra N*
shows *prob-space* ($A \gg\!= B$)
using *measurable-bind-prob-space*[*OF measurable-const*, *OF A B*, *THEN measurable-space*, *of undefined count-space UNIV*]
by (*simp add*: *space-prob-algebra*)

lemma *sets-bind'*:
assumes *A*: $A \in$ *space* (*prob-algebra M*) **and** *B*: $B \in M \rightarrow_M$ *prob-algebra N*
shows *sets* ($A \gg\!= B$) = *sets N*
using *measurable-bind-prob-space*[*OF measurable-const*, *OF A B*, *THEN measurable-space*, *of undefined count-space UNIV*]
by (*simp add*: *space-prob-algebra*)

lemma *bind-cong-AE'*:
assumes *M*: $M \in$ *space* (*prob-algebra L*)
and *f*: $f \in L \rightarrow_M$ *prob-algebra N* **and** *g*: $g \in L \rightarrow_M$ *prob-algebra N*
and *ae*: *AE x in M*. *f x* = *g x*
shows *bind M f* = *bind M g*
proof (*rule measure-eqI*)
show *sets* ($M \gg\!= f$) = *sets* ($M \gg\!= g$)
unfolding *sets-bind'*[*OF M f*] *sets-bind'*[*OF M g*] **..**
show $A \in$ *sets* ($M \gg\!= f$) $\Longrightarrow$ *emeasure* ($M \gg\!= f$) *A* = *emeasure* ($M \gg\!= g$) *A*
for *A*
unfolding *sets-bind'*[*OF M f*]
using *emeasure-bind-prob-algebra*[*OF M f*, *of A*] *emeasure-bind-prob-algebra*[*OF M g*, *of A*] *ae*
by (*auto intro*: *nn-integral-cong-AE*)
qed

lemma *density-discrete*:
countable A $\Longrightarrow$ *sets N* = *Set.Pow A* $\Longrightarrow$ ($\bigwedge x$. *f x* $\geq$ *0*) $\Longrightarrow$ ($\bigwedge x$. *x* $\in$ *A* $\Longrightarrow$ *f x* = *emeasure N* {*x*}) $\Longrightarrow$
density (*count-space A*) *f* = *N*
by (*rule measure-eqI-countable*[*of - A*]) (*auto simp*: *emeasure-density*)

lemma *distr-density-discrete*:
fixes *f'*
assumes *countable A*
assumes *f'* $\in$ *borel-measurable M*
assumes *g* $\in$ *measurable M* (*count-space A*)
defines *f* $\equiv \lambda x. \int^+ t.$ (*if g t* = *x then 1 else 0*) $*$ *f' t* ∂M
assumes $\bigwedge x.$ *x* $\in$ *space M* $\Longrightarrow$ *g x* $\in$ *A*
shows *density* (*count-space A*) ($\lambda x.$ *f x*) = *distr* (*density M f'*) (*count-space A*) *g*

proof (*rule density-discrete*)
fix x **assume** x: $x \in A$
have $f\ x = \int^+ t.\ indicator\ (g\ -`\ \{x\} \cap space\ M)\ t * f'\ t\ \partial M$ (**is** - = *?I*)
unfolding *f-def*
by (*intro nn-integral-cong*) (*simp split*: *split-indicator*)
also from x **have** *in-sets*: $g\ -`\ \{x\} \cap space\ M \in sets\ M$
by (*intro measurable-sets*[*OF assms*(*3*)]) *simp*
have *?I* = *emeasure* (*density M f'*) ($g\ -`\ \{x\} \cap space\ M$) **unfolding** *f-def*
by (*subst emeasure-density*[*OF assms*(*2*) *in-sets*], *subst mult.commute*) (*rule refl*)
also from *assms*(*3*) x **have** ... = *emeasure* (*distr* (*density M f'*) (*count-space A*) *g*) {x}
by (*subst emeasure-distr*) *simp-all*
finally show $f\ x$ = *emeasure* (*distr* (*density M f'*) (*count-space A*) *g*) {x} .
qed (*insert assms*, *auto*)

lemma *bind-cong-AE*:
assumes $M = N$
assumes f: $f \in measurable\ N\ (subprob\text{-}algebra\ B)$
assumes g: $g \in measurable\ N\ (subprob\text{-}algebra\ B)$
assumes *ae*: *AE x in N*. $f\ x = g\ x$
shows *bind M f* = *bind N g*
proof *cases*
assume *space N* = {} **then show** *?thesis*
using ‹$M = N$› **by** (*simp add*: *bind-empty*)
next
assume *space N* ≠ {}
show *?thesis* **unfolding** ‹$M = N$›
proof (*rule measure-eqI*)
have ∗: *sets* (*N* ⤜ *f*) = *sets B*
using *sets-bind*[*OF sets-kernel*[*OF f*] ‹*space N* ≠ {}›] **by** *simp*
then show *sets* (*N* ⤜ *f*) = *sets* (*N* ⤜ *g*)
using *sets-bind*[*OF sets-kernel*[*OF g*] ‹*space N* ≠ {}›] **by** *auto*
fix *A* **assume** *A* ∈ *sets* (*N* ⤜ *f*)
then have *A* ∈ *sets B*
unfolding ∗ .
with *ae f g* ‹*space N* ≠ {}› **show** *emeasure* (*N* ⤜ *f*) *A* = *emeasure* (*N* ⤜ *g*) *A*
by (*subst* (*1 2*) *emeasure-bind*[**where** *N*=*B*]) (*auto intro*!: *nn-integral-cong-AE*)
qed
qed

lemma *bind-cong-simp*: $M = N \Longrightarrow (\bigwedge x.\ x \in space\ M \Rightarrow f\ x = g\ x) \Longrightarrow bind\ M\ f = bind\ N\ g$
by (*auto simp*: *simp-implies-def intro*!: *bind-cong*)

lemma *sets-bind-measurable*:
assumes f: $f \in measurable\ M\ (subprob\text{-}algebra\ B)$
assumes M: $space\ M \neq \{\}$

shows *sets* $(M \gg\!= f) = sets\ B$
using M **by** (*intro sets-bind*[*OF sets-kernel*[*OF f*]]) *auto*

lemma *space-bind-measurable*:
assumes *f*: $f \in measurable\ M\ (subprob\text{-}algebra\ B)$
assumes *M*: $space\ M \neq \{\}$
shows $space\ (M \gg\!= f) = space\ B$
using M **by** (*intro space-bind*[*OF sets-kernel*[*OF f*]]) *auto*

lemma *bind-distr-return*:
$f \in M \to_M N \Longrightarrow g \in N \to_M L \Longrightarrow space\ M \neq \{\} \Longrightarrow$
distr M N f $\gg\!=$ $(\lambda x.\ return\ L\ (g\ x)) = distr\ M\ L\ (\lambda x.\ g\ (f\ x))$
by (*subst bind-distr*[*OF - measurable-compose*[*OF - return-measurable*]])
(*auto intro*!: *bind-return-distr'*)

lemma (**in** *prob-space*) *AE-eq-constD*:
assumes $AE\ x\ in\ M.\ x = y$
shows $M = return\ M\ y\ y \in space\ M$
proof −
have $AE\ x\ in\ M.\ x \in space\ M$
by *auto*
with *assms* **have** $AE\ x\ in\ M.\ y \in space\ M$
by *eventually-elim auto*
thus $y \in space\ M$
by *simp*

show $M = return\ M\ y$
proof (*rule measure-eqI*)
fix *X* **assume** *X*: $X \in sets\ M$
have $AE\ x\ in\ M.\ (x \in X) = (x \in (if\ y \in X\ then\ space\ M\ else\ \{\}))$
using *assms* **by** *eventually-elim* (*use X* ‹$y \in space\ M$› **in** *auto*)
hence $emeasure\ M\ X = emeasure\ M\ (if\ y \in X\ then\ space\ M\ else\ \{\})$
using *X* **by** (*intro emeasure-eq-AE*) *auto*
also have $\ldots = emeasure\ (return\ M\ y)\ X$
using *X* **by** (*auto simp*: *emeasure-space-1*)
finally show $emeasure\ M\ X = \ldots$ **.**
qed *auto*
qed

end

8 Projective Family

theory *Projective-Family*
imports *Giry-Monad*
begin

lemma *vimage-restrict-preseve-mono*:
assumes *J*: $J \subseteq I$

and *sets*: $A \subseteq (\Pi_E\ i{\in}J.\ S\ i)\ B \subseteq (\Pi_E\ i{\in}J.\ S\ i)$ **and** *ne*: $(\Pi_E\ i{\in}I.\ S\ i) \neq \{\}$
and *eq*: $(\lambda x.\ restrict\ x\ J) -` A \cap (\Pi_E\ i{\in}I.\ S\ i) \subseteq (\lambda x.\ restrict\ x\ J) -` B \cap (\Pi_E\ i{\in}I.\ S\ i)$
shows $A \subseteq B$
proof (*intro subsetI*)
fix x **assume** $x \in A$
from *ne* **obtain** y **where** y: $\bigwedge i.\ i \in I \Longrightarrow y\ i \in S\ i$ **by** *auto*
have $J \cap (I - J) = \{\}$ **by** *auto*
show $x \in B$
proof *cases*
assume x: $x \in (\Pi_E\ i{\in}J.\ S\ i)$
have *merge* $J\ (I - J)\ (x,y) \in (\lambda x.\ restrict\ x\ J) -` A \cap (\Pi_E\ i{\in}I.\ S\ i)$
using $y\ x$ ‹$J \subseteq I$› *PiE-cancel-merge*[*of* $J\ I - J\ x\ y\ S$] ‹$x{\in}A$›
by (*auto simp del*: *PiE-cancel-merge simp add*: *Un-absorb1*)
also have $\ldots \subseteq (\lambda x.\ restrict\ x\ J) -` B \cap (\Pi_E\ i{\in}I.\ S\ i)$ **by** *fact*
finally show $x \in B$
using $y\ x$ ‹$J \subseteq I$› *PiE-cancel-merge*[*of* $J\ I - J\ x\ y\ S$] ‹$x{\in}A$› *eq*
by (*auto simp del*: *PiE-cancel-merge simp add*: *Un-absorb1*)
qed (*insert* ‹$x{\in}A$› *sets*, *auto*)
qed

locale *projective-family* =
fixes $I :: 'i\ set$ **and** $P :: 'i\ set \Rightarrow ('i \Rightarrow 'a)\ measure$ **and** $M :: 'i \Rightarrow 'a\ measure$
assumes P: $\bigwedge J\ H.\ J \subseteq H \Longrightarrow finite\ H \Longrightarrow H \subseteq I \Longrightarrow P\ J = distr\ (P\ H)\ (PiM\ J\ M)\ (\lambda f.\ restrict\ f\ J)$
assumes *prob-space-P*: $\bigwedge J.\ finite\ J \Longrightarrow J \subseteq I \Longrightarrow prob\text{-}space\ (P\ J)$
begin

lemma *sets-P*: $finite\ J \Longrightarrow J \subseteq I \Longrightarrow sets\ (P\ J) = sets\ (PiM\ J\ M)$
by (*subst P*[*of J J*]) *simp-all*

lemma *space-P*: $finite\ J \Longrightarrow J \subseteq I \Longrightarrow space\ (P\ J) = space\ (PiM\ J\ M)$
using *sets-P* **by** (*rule sets-eq-imp-space-eq*)

lemma *not-empty-M*: $i \in I \Longrightarrow space\ (M\ i) \neq \{\}$
using *prob-space-P*[*THEN prob-space.not-empty*] **by** (*auto simp*: *space-PiM space-P*)

lemma *not-empty*: $space\ (PiM\ I\ M) \neq \{\}$
by (*simp add*: *not-empty-M*)

abbreviation
$emb\ L\ K \equiv prod\text{-}emb\ L\ M\ K$

lemma *emb-preserve-mono*:
assumes $J \subseteq L\ L \subseteq I$ **and** *sets*: $X \in sets\ (Pi_M\ J\ M)\ Y \in sets\ (Pi_M\ J\ M)$
assumes $emb\ L\ J\ X \subseteq emb\ L\ J\ Y$
shows $X \subseteq Y$
proof (*rule vimage-restrict-preseve-mono*)
show $X \subseteq (\Pi_E\ i{\in}J.\ space\ (M\ i))\ Y \subseteq (\Pi_E\ i{\in}J.\ space\ (M\ i))$

using *sets*[*THEN sets.sets-into-space*] **by** (*auto simp*: *space-PiM*)
show $(\Pi_E\ i{\in}L.\ space\ (M\ i)) \neq \{\}$
using ‹$L \subseteq I$› **by** (*auto simp add*: *not-empty-M space-PiM*[*symmetric*])
show $(\lambda x.\ restrict\ x\ J) -` X \cap (\Pi_E\ i{\in}L.\ space\ (M\ i)) \subseteq (\lambda x.\ restrict\ x\ J) -` Y \cap (\Pi_E\ i{\in}L.\ space\ (M\ i))$
using ‹*prod-emb L M J X* ⊆ *prod-emb L M J Y*› **by** (*simp add*: *prod-emb-def*)
qed *fact*

lemma *emb-injective*:
assumes *L*: $J \subseteq L\ L \subseteq I$ **and** *X*: $X \in sets\ (Pi_M\ J\ M)$ **and** *Y*: $Y \in sets\ (Pi_M\ J\ M)$
shows $emb\ L\ J\ X = emb\ L\ J\ Y \Longrightarrow X = Y$
by (*intro antisym emb-preserve-mono*[*OF L X Y*] *emb-preserve-mono*[*OF L Y X*]) *auto*

lemma *emeasure-P*: $J \subseteq K \Longrightarrow finite\ K \Longrightarrow K \subseteq I \Longrightarrow X \in sets\ (PiM\ J\ M) \Longrightarrow P\ K\ (emb\ K\ J\ X) = P\ J\ X$
by (*auto intro*!: *emeasure-distr-restrict*[*symmetric*] *simp*: *P sets-P*)

inductive-set *generator* :: $('i \Rightarrow 'a)\ set\ set$ **where**
$finite\ J \Longrightarrow J \subseteq I \Longrightarrow X \in sets\ (Pi_M\ J\ M) \Longrightarrow emb\ I\ J\ X \in generator$

lemma *algebra-generator*: *algebra* (*space* (*PiM I M*)) *generator*
unfolding *algebra-iff-Int*
proof (*safe elim*!: *generator.cases*)
fix *J X* **assume** *J*: *finite J J* ⊆ *I* **and** *X*: $X \in sets\ (PiM\ J\ M)$

from *X*[*THEN sets.sets-into-space*] *J* **show** $x \in emb\ I\ J\ X \Longrightarrow x \in space\ (PiM\ I\ M)$ **for** *x*
by (*auto simp*: *prod-emb-def space-PiM*)

have *emb I J* (*space* (*PiM J M*) − *X*) ∈ *generator*
by (*intro generator.intros J sets.Diff sets.top X*)
with *J* **show** $space\ (Pi_M\ I\ M) - emb\ I\ J\ X \in generator$
by (*simp add*: *space-PiM prod-emb-PiE*)

fix *K Y* **assume** *K*: *finite K K* ⊆ *I* **and** *Y*: $Y \in sets\ (PiM\ K\ M)$
have *emb I* (*J* ∪ *K*) (*emb* (*J* ∪ *K*) *J X*) ∩ *emb I* (*J* ∪ *K*) (*emb* (*J* ∪ *K*) *K Y*) ∈ *generator*
unfolding *prod-emb-Int*[*symmetric*]
by (*intro generator.intros sets.Int measurable-prod-emb*) (*auto intro*!: *J K X Y*)
with *J K X Y* **show** *emb I J X* ∩ *emb I K Y* ∈ *generator*
by *simp*
qed (*force simp*: *generator.simps prod-emb-empty*[*symmetric*])

interpretation *generator*: *algebra space* (*PiM I M*) *generator*
by (*rule algebra-generator*)

lemma *sets-PiM-generator*: *sets* (*PiM I M*) = *sigma-sets* (*space* (*PiM I M*)) *gen-*

erator
proof (*intro antisym sets.sigma-sets-subset*)
show *sets* (*PiM I M*) ⊆ *sigma-sets* (*space* (*PiM I M*)) *generator*
unfolding *sets-PiM-single space-PiM*[*symmetric*]
proof (*intro sigma-sets-mono'*, *safe*)
fix *i A* **assume** *i* ∈ *I* **and** *A*: *A* ∈ *sets* (*M i*)
then have {*f* ∈ *space* (*Pi*$_M$ *I M*). *f i* ∈ *A*} = *emb I* {*i*} (Π$_E$ *j*∈{*i*}. *A*)
by (*auto simp*: *prod-emb-def space-PiM restrict-def Pi-iff PiE-iff extensional-def*)
with ‹*i*∈*I*› *A* **show** {*f* ∈ *space* (*Pi*$_M$ *I M*). *f i* ∈ *A*} ∈ *generator*
by (*auto intro*!: *generator.intros sets-PiM-I-finite*)
qed
qed (*auto elim*!: *generator.cases*)

definition *mu-G* (*μG*) **where**
μG A = (*THE x*. ∀ *J*⊆*I*. *finite J* ⟶ (∀ *X*∈*sets* (*Pi*$_M$ *J M*). *A* = *emb I J X* ⟶ *x* = *emeasure* (*P J*) *X*))

definition *lim* :: ('*i* ⇒ '*a*) *measure* **where**
lim = *extend-measure* (*space* (*PiM I M*)) *generator* (λ*x*. *x*) *μG*

lemma *space-lim*[*simp*]: *space lim* = *space* (*PiM I M*)
using *generator.space-closed*
unfolding *lim-def* **by** (*intro space-extend-measure*) *simp*

lemma *sets-lim*[*simp*, *measurable*]: *sets lim* = *sets* (*PiM I M*)
using *generator.space-closed* **by** (*simp add*: *lim-def sets-PiM-generator sets-extend-measure*)

lemma *mu-G-spec*:
assumes *J*: *finite J J* ⊆ *I X* ∈ *sets* (*Pi*$_M$ *J M*)
shows *μG* (*emb I J X*) = *emeasure* (*P J*) *X*
unfolding *mu-G-def*
proof (*intro the-equality allI impI ballI*)
fix *K Y* **assume** *K*: *finite K K* ⊆ *I Y* ∈ *sets* (*Pi*$_M$ *K M*)
and [*simp*]: *emb I J X* = *emb I K Y*
have *emeasure* (*P K*) *Y* = *emeasure* (*P* (*K* ∪ *J*)) (*emb* (*K* ∪ *J*) *K Y*)
using *K J* **by** (*simp add*: *emeasure-P*)
also have *emb* (*K* ∪ *J*) *K Y* = *emb* (*K* ∪ *J*) *J X*
using *K J* **by** (*simp add*: *emb-injective*[*of K* ∪ *J I*])
also have *emeasure* (*P* (*K* ∪ *J*)) (*emb* (*K* ∪ *J*) *J X*) = *emeasure* (*P J*) *X*
using *K J* **by** (*subst emeasure-P*) *simp-all*
finally show *emeasure* (*P J*) *X* = *emeasure* (*P K*) *Y* **..**
qed (*insert J*, *force*)

lemma *positive-mu-G*: *positive generator μG*
proof −
show *?thesis*
proof (*safe intro*!: *positive-def*[*THEN iffD2*])
obtain *J* **where** *finite J J* ⊆ *I* **by** *auto*

then have $\mu G\ (emb\ I\ J\ \{\}) = 0$
by (*subst mu-G-spec*) *auto*
then show $\mu G\ \{\} = 0$ **by** *simp*
qed
qed

lemma *additive-mu-G*: *additive generator* μG
proof (*safe intro*!: *additive-def*[*THEN iffD2*] *elim*!: *generator.cases*)
fix $J\ X\ K\ Y$ **assume** *J*: *finite* $J\ J \subseteq I\ X \in sets\ (PiM\ J\ M)$
and *K*: *finite* $K\ K \subseteq I\ Y \in sets\ (PiM\ K\ M)$
and $emb\ I\ J\ X \cap emb\ I\ K\ Y = \{\}$
then have *JK-disj*: $emb\ (J \cup K)\ J\ X \cap emb\ (J \cup K)\ K\ Y = \{\}$
by (*intro emb-injective*[*of* $J \cup K\ I$ - $\{\}$]) (*auto simp*: *sets.Int prod-emb-Int*)
have $\mu G\ (emb\ I\ J\ X \cup emb\ I\ K\ Y) = \mu G\ (emb\ I\ (J \cup K)\ (emb\ (J \cup K)\ J\ X \cup emb\ (J \cup K)\ K\ Y))$
using $J\ K$ **by** *simp*
also have $\ldots = emeasure\ (P\ (J \cup K))\ (emb\ (J \cup K)\ J\ X \cup emb\ (J \cup K)\ K\ Y)$
using $J\ K$ **by** (*simp add*: *mu-G-spec sets.Un del*: *prod-emb-Un*)
also have $\ldots = \mu G\ (emb\ I\ J\ X) + \mu G\ (emb\ I\ K\ Y)$
using $J\ K$ *JK-disj* **by** (*simp add*: *plus-emeasure*[*symmetric*] *mu-G-spec emeasure-P sets-P*)
finally show $\mu G\ (emb\ I\ J\ X \cup emb\ I\ K\ Y) = \mu G\ (emb\ I\ J\ X) + \mu G\ (emb\ I\ K\ Y)$ **.**
qed

lemma *emeasure-lim*:
assumes *JX*: *finite* $J\ J \subseteq I\ X \in sets\ (PiM\ J\ M)$
assumes *cont*: $\bigwedge J\ X.\ (\bigwedge i.\ J\ i \subseteq I) \Longrightarrow incseq\ J \Longrightarrow (\bigwedge i.\ finite\ (J\ i)) \Longrightarrow (\bigwedge i.\ X\ i \in sets\ (PiM\ (J\ i)\ M)) \Longrightarrow$
$decseq\ (\lambda i.\ emb\ I\ (J\ i)\ (X\ i)) \Longrightarrow 0 < (INF\ i.\ P\ (J\ i)\ (X\ i)) \Longrightarrow (\bigcap i.\ emb\ I\ (J\ i)\ (X\ i)) \neq \{\}$
shows $emeasure\ lim\ (emb\ I\ J\ X) = P\ J\ X$
proof −
have $\exists \mu.\ (\forall s \in generator.\ \mu\ s = \mu G\ s) \wedge$
$measure\text{-}space\ (space\ (PiM\ I\ M))\ (sigma\text{-}sets\ (space\ (PiM\ I\ M))\ generator)\ \mu$
proof (*rule generator.caratheodory-empty-continuous*[*OF positive-mu-G additive-mu-G*])
show $\bigwedge A.\ A \in generator \Longrightarrow \mu G\ A \neq \infty$
proof (*clarsimp elim*!: *generator.cases simp*: *mu-G-spec del*: *notI*)
fix J **assume** *finite* $J\ J \subseteq I$
then interpret *prob-space* $P\ J$ **by** (*rule prob-space-P*)
show $\bigwedge X.\ X \in sets\ (Pi_M\ J\ M) \Longrightarrow emeasure\ (P\ J)\ X \neq top$
by *simp*
qed
next
fix A **assume** *range* $A \subseteq generator$ **and** *decseq* $A\ (\bigcap i.\ A\ i) = \{\}$
then have $\forall i.\ \exists J\ X.\ A\ i = emb\ I\ J\ X \wedge finite\ J \wedge J \subseteq I \wedge X \in sets\ (PiM\ J\ M)$

unfolding *image-subset-iff generator.simps* **by** *blast*
then obtain $J\ X$ **where** A: $\bigwedge i.\ A\ i = emb\ I\ (J\ i)\ (X\ i)$
and $*$: $\bigwedge i.\ finite\ (J\ i)\ \bigwedge i.\ J\ i \subseteq I\ \bigwedge i.\ X\ i \in sets\ (PiM\ (J\ i)\ M)$
by *metis*
have $(INF\ i.\ P\ (J\ i)\ (X\ i)) = 0$
proof (*rule ccontr*)
assume *INF-P*: $(INF\ i.\ P\ (J\ i)\ (X\ i)) \neq 0$
have $(\bigcap i.\ emb\ I\ (\bigcup n{\leq}i.\ J\ n)\ (emb\ (\bigcup n{\leq}i.\ J\ n)\ (J\ i)\ (X\ i))) \neq \{\}$
proof (*rule cont*)
show *decseq* $(\lambda i.\ emb\ I\ (\bigcup n{\leq}i.\ J\ n)\ (emb\ (\bigcup n{\leq}i.\ J\ n)\ (J\ i)\ (X\ i)))$
using $*$ ‹*decseq A*› **by** (*subst prod-emb-trans*) (*auto simp*: *A*[*abs-def*])
show $0 < (INF\ i.\ P\ (\bigcup n{\leq}i.\ J\ n)\ (emb\ (\bigcup n{\leq}i.\ J\ n)\ (J\ i)\ (X\ i)))$
using $*$ *INF-P* **by** (*subst emeasure-P*) (*auto simp*: *less-le intro*!: *INF-greatest*)
show *incseq* $(\lambda i.\ \bigcup n{\leq}i.\ J\ n)$
by (*force simp*: *incseq-def*)
qed (*insert* $*$, *auto*)
with ‹$(\bigcap i.\ A\ i) = \{\}$› $*$ **show** *False*
by (*subst* (*asm*) *prod-emb-trans*) (*auto simp*: *A*[*abs-def*])
qed
moreover have $(\lambda i.\ P\ (J\ i)\ (X\ i)) \longrightarrow (INF\ i.\ P\ (J\ i)\ (X\ i))$
proof (*intro LIMSEQ-INF antimonoI*)
fix $x\ y :: nat$ **assume** $x \leq y$
have $P\ (J\ y \cup J\ x)\ (emb\ (J\ y \cup J\ x)\ (J\ y)\ (X\ y)) \leq P\ (J\ y \cup J\ x)\ (emb\ (J\ y \cup J\ x)\ (J\ x)\ (X\ x))$
using ‹*decseq A*›[*THEN decseqD*, *OF* ‹$x{\leq}y$›] $*$
by (*auto simp*: *A sets-P del*: *subsetI intro*!: *emeasure-mono* ‹$x \leq y$› *emb-preserve-mono*[*of J y* $\cup$ *J x I*, **where** $X{=}emb\ (J\ y \cup J\ x)\ (J\ y)\ (X\ y)$])
then show $P\ (J\ y)\ (X\ y) \leq P\ (J\ x)\ (X\ x)$
using $*$ **by** (*simp add*: *emeasure-P*)
qed
ultimately show $(\lambda i.\ \mu G\ (A\ i)) \longrightarrow 0$
by (*auto simp*: *A*[*abs-def*] *mu-G-spec* $*$)
qed
then obtain μ **where** *eq*: $\forall s{\in}generator.\ \mu\ s = \mu G\ s$
and *ms*: *measure-space* $(space\ (PiM\ I\ M))\ (sets\ (PiM\ I\ M))\ \mu$
by (*metis sets-PiM-generator*)
show *?thesis*
proof (*subst emeasure-extend-measure*[*OF lim-def*])
show $A \in generator \Longrightarrow \mu\ A = \mu G\ A$ **for** A
using *eq* **by** *simp*
show *positive* $(sets\ lim)\ \mu$ *countably-additive* $(sets\ lim)\ \mu$
using *ms* **by** (*auto simp add*: *measure-space-def*)
show $(\lambda x.\ x)$ ‘ $generator \subseteq Pow\ (space\ (Pi_M\ I\ M))$
using *generator.space-closed* **by** *simp*
show $emb\ I\ J\ X \in generator\ \mu G\ (emb\ I\ J\ X) = emeasure\ (P\ J)\ X$
using *JX* **by** (*auto intro*: *generator.intros simp*: *mu-G-spec*)
qed

qed

end

sublocale *product-prob-space* ⊆ *projective-family I* λ*J. PiM J M M*
  unfolding *projective-family-def*
proof (*intro conjI allI impI distr-restrict*)
  show ⋀*J. finite J* ⟹ *prob-space* (*Pi*$_M$ *J M*)
    by (*intro prob-spaceI*) (*simp add*: *space-PiM emeasure-PiM emeasure-space-1*)
qed *auto*

Proof due to Ionescu Tulcea.

locale *Ionescu-Tulcea* =
  fixes *P* :: *nat* ⇒ (*nat* ⇒ *'a*) ⇒ *'a measure* **and** *M* :: *nat* ⇒ *'a measure*
  assumes *P*[*measurable*]: ⋀*i. P i* ∈ *measurable* (*PiM* {*0..<i*} *M*) (*subprob-algebra* (*M i*))
  assumes *prob-space-P*: ⋀*i x. x* ∈ *space* (*PiM* {*0..<i*} *M*) ⟹ *prob-space* (*P i x*)
begin

lemma *non-empty*[*simp*]: *space* (*M i*) ≠ {}
proof (*induction i rule*: *less-induct*)
  case (*less i*)
  then obtain *x* **where** ⋀*j. j* < *i* ⟹ *x j* ∈ *space* (*M j*)
    unfolding *ex-in-conv*[*symmetric*] **by** *metis*
  then have ∗: *restrict x* {*0..<i*} ∈ *space* (*PiM* {*0..<i*} *M*)
    by (*auto simp*: *space-PiM PiE-iff*)
  then interpret *prob-space P i* (*restrict x* {*0..<i*})
    by (*rule prob-space-P*)
  show *?case*
    using *not-empty subprob-measurableD*(*1*)[*OF P*, *OF* ∗] **by** *simp*
qed

lemma *space-PiM-not-empty*[*simp*]: *space* (*PiM UNIV M*) ≠ {}
  unfolding *space-PiM-empty-iff* **by** *auto*

lemma *space-P*: *x* ∈ *space* (*PiM* {*0..<n*} *M*) ⟹ *space* (*P n x*) = *space* (*M n*)
  by (*simp add*: *P*[*THEN subprob-measurableD*(*1*)])

lemma *sets-P*[*measurable-cong*]: *x* ∈ *space* (*PiM* {*0..<n*} *M*) ⟹ *sets* (*P n x*) = *sets* (*M n*)
  by (*simp add*: *P*[*THEN subprob-measurableD*(*2*)])

definition *eP* :: *nat* ⇒ (*nat* ⇒ *'a*) ⇒ (*nat* ⇒ *'a*) *measure* **where**
  eP n ω = *distr* (*P n ω*) (*PiM* {*0..<Suc n*} *M*) (*fun-upd ω n*)

lemma *measurable-eP*[*measurable*]:
  eP n ∈ *measurable* (*PiM* {*0..< n*} *M*) (*subprob-algebra* (*PiM* {*0..<Suc n*} *M*))
  by (*auto simp*: *eP-def*[*abs-def*] *measurable-split-conv*

intro!: *measurable-fun-upd*[**where** *J*={*0*..<*n*}] *measurable-distr2*[*OF* - *P*])

lemma *space-eP*:
x ∈ *space* (*PiM* {*0*..<*n*} *M*) ⟹ *space* (*eP n x*) = *space* (*PiM* {*0*..<*Suc n*} *M*)
by (*simp add*: *measurable-eP*[*THEN subprob-measurableD*(*1*)])

lemma *sets-eP*[*measurable*]:
x ∈ *space* (*PiM* {*0*..<*n*} *M*) ⟹ *sets* (*eP n x*) = *sets* (*PiM* {*0*..<*Suc n*} *M*)
by (*simp add*: *measurable-eP*[*THEN subprob-measurableD*(*2*)])

lemma *prob-space-eP*: *x* ∈ *space* (*PiM* {*0*..<*n*} *M*) ⟹ *prob-space* (*eP n x*)
unfolding *eP-def*
by (*intro prob-space.prob-space-distr prob-space-P measurable-fun-upd*[**where** *J*={*0*..<*n*}])
auto

lemma *nn-integral-eP*:
ω ∈ *space* (*PiM* {*0*..<*n*} *M*) ⟹ *f* ∈ *borel-measurable* (*PiM* {*0*..<*Suc n*} *M*)
⟹
(∫⁺*x*. *f x* ∂*eP n ω*) = (∫⁺*x*. *f* (*ω*(*n* := *x*)) ∂*P n ω*)
unfolding *eP-def*
by (*subst nn-integral-distr*) (*auto intro*!: *measurable-fun-upd*[**where** *J*={*0*..<*n*}]
simp: *space-PiM PiE-iff*)

lemma *emeasure-eP*:
assumes *ω*[*simp*]: *ω* ∈ *space* (*PiM* {*0*..<*n*} *M*) **and** *A*[*measurable*]: *A* ∈ *sets*
(*PiM* {*0*..<*Suc n*} *M*)
shows *eP n ω A* = *P n ω* ((λ*x*. *ω*(*n* := *x*)) −‘ *A* ∩ *space* (*M n*))
using *nn-integral-eP*[*of ω n indicator A*]
apply (*simp add*: *sets-eP nn-integral-indicator*[*symmetric*] *sets-P del*: *nn-integral-indicator*)
apply (*subst nn-integral-indicator*[*symmetric*])
using *measurable-sets*[*OF measurable-fun-upd*[*OF* - *measurable-const*[*OF ω*] *measurable-id*] *A*, *of n*]
apply (*auto simp add*: *sets-P atLeastLessThanSuc space-P simp del*: *nn-integral-indicator*
intro!: *nn-integral-cong split*: *split-indicator*)
done

primrec *C* :: *nat* ⇒ *nat* ⇒ (*nat* ⇒ *'a*) ⇒ (*nat* ⇒ *'a*) *measure* **where**
C n 0 ω = *return* (*PiM* {*0*..<*n*} *M*) *ω*
| *C n* (*Suc m*) *ω* = *C n m ω* ⤜ *eP* (*n* + *m*)

lemma *measurable-C*[*measurable*]:
C n m ∈ *measurable* (*PiM* {*0*..<*n*} *M*) (*subprob-algebra* (*PiM* {*0*..<*n* + *m*} *M*))
by (*induction m*) *auto*

lemma *space-C*:
x ∈ *space* (*PiM* {*0*..<*n*} *M*) ⟹ *space* (*C n m x*) = *space* (*PiM* {*0*..<*n* + *m*}
M)
by (*simp add*: *measurable-C*[*THEN subprob-measurableD*(*1*)])

lemma *sets-C*[*measurable-cong*]:
  $x \in space\ (PiM\ \{0..<n\}\ M) \Longrightarrow sets\ (C\ n\ m\ x) = sets\ (PiM\ \{0..<n + m\}\ M)$
  by (*simp add*: *measurable-C*[*THEN subprob-measurableD*(*2*)])

lemma *prob-space-C*: $x \in space\ (PiM\ \{0..<n\}\ M) \Longrightarrow prob\text{-}space\ (C\ n\ m\ x)$
proof (*induction m*)
  case (*Suc m*) **then show** *?case*
   by (*auto intro*!: *prob-space.prob-space-bind*[**where** *S*=*PiM* $\{0..<Suc\ (n + m)\}$ *M*]
            simp: *space-C prob-space-eP*)
qed (*auto intro*!: *prob-space-return simp*: *space-PiM*)

lemma *split-C*:
  assumes ω: $\omega \in space\ (PiM\ \{0..<n\}\ M)$ **shows** $(C\ n\ m\ \omega \gg\!= C\ (n + m)\ l) = C\ n\ (m + l)\ \omega$
proof (*induction l*)
  case *0*
  with ω **show** *?case*
   by (*simp add*: *bind-return-distr'* *prob-space-C*[*THEN prob-space.not-empty*]
            distr-cong[*OF refl sets-C*[*symmetric*, *OF* ω]])
next
  case (*Suc l*) **with** ω **show** *?case*
   by (*simp add*: *bind-assoc*[*symmetric*, *OF - measurable-eP*]) (*simp add*: *ac-simps*)
qed

lemma *nn-integral-C*:
  assumes $m \le m'$ **and** *f*[*measurable*]: $f \in borel\text{-}measurable\ (PiM\ \{0..<n+m\}\ M)$
   and *nonneg*: $\bigwedge x.\ x \in space\ (PiM\ \{0..<n+m\}\ M) \Longrightarrow 0 \le f\ x$
   and *x*: $x \in space\ (PiM\ \{0..<n\}\ M)$
  shows $(\int^+ x.\ f\ x\ \partial C\ n\ m\ x) = (\int^+ x.\ f\ (restrict\ x\ \{0..<n+m\})\ \partial C\ n\ m'\ x)$
  using ‹$m \le m'$›
proof (*induction rule*: *dec-induct*)
  case (*step i*)
  let *?E* = $\lambda x.\ f\ (restrict\ x\ \{0..<n + m\})$ **and** *?C* = $\lambda i\ f.\ \int^+ x.\ f\ x\ \partial C\ n\ i\ x$
  from ‹$m \le i$› *x* **have** *?C i ?E* = *?C* (*Suc i*) *?E*
   by (*auto simp*: *nn-integral-bind*[**where** *B*=*PiM* $\{0 ..< Suc\ (n + i)\}$ *M*] *space-C nn-integral-eP*
            intro!: *nn-integral-cong*)
         (*simp add*: *space-PiM PiE-iff nonneg prob-space.emeasure-space-1*[*OF prob-space-P*])
  with *step* **show** *?case* **by** (*simp del*: *restrict-apply*)
qed (*auto simp*: *space-PiM space-C*[*OF x*] *simp del*: *restrict-apply intro*!: *nn-integral-cong*)

lemma *emeasure-C*:
  assumes $m \le m'$ **and** *A*[*measurable*]: $A \in sets\ (PiM\ \{0..<n+m\}\ M)$ **and** [*simp*]: $x \in space\ (PiM\ \{0..<n\}\ M)$
   shows $emeasure\ (C\ n\ m'\ x)\ (prod\text{-}emb\ \{0..<n + m'\}\ M\ \{0..<n+m\}\ A) =$

emeasure (*C n m x*) *A*
using *assms*
by (*subst* (*1 2*) *nn-integral-indicator*[*symmetric*])
(*auto intro*!: *nn-integral-cong split*: *split-indicator simp del*: *nn-integral-indicator*
simp: *nn-integral-C*[*of m m'* - *n*] *prod-emb-iff space-PiM PiE-iff sets-C space-C*)

lemma *distr-C*:
assumes $m \leq m'$ **and** [*simp*]: $x \in$ *space* (*PiM* {*0*..<*n*} *M*)
shows *C n m x* = *distr* (*C n m' x*) (*PiM* {*0*..<*n*+*m*} *M*) (λ*x*. *restrict x* {*0*..<*n*+*m*})
proof (*rule measure-eqI*)
fix *A* **assume** $A \in$ *sets* (*C n m x*)
with ‹$m \leq m'$› **show** *emeasure* (*C n m x*) *A* = *emeasure* (*distr* (*C n m' x*) (*Pi*$_M$ {*0*..<*n* + *m*} *M*) (λ*x*. *restrict x* {*0*..<*n* + *m*})) *A*
by (*subst emeasure-C*[*symmetric*, *OF* ‹$m \leq m'$›]) (*auto intro*!: *emeasure-distr-restrict*[*symmetric*] *simp*: *sets-C*)
qed (*simp add*: *sets-C*)

definition *up-to* :: *nat set* ⇒ *nat* **where**
up-to J = (*LEAST n*. ∀ *i*≥*n*. *i* ∉ *J*)

lemma *up-to-less*: *finite J* ⟹ *i* ∈ *J* ⟹ *i* < *up-to J*
unfolding *up-to-def*
by (*rule LeastI2*[*of* - *Suc* (*Max J*)]) (*auto simp*: *Suc-le-eq not-le*[*symmetric*])

lemma *up-to-iff*: *finite J* ⟹ *up-to J* ≤ *n* ⟷ (∀ *i*∈*J*. *i* < *n*)
proof *safe*
show *finite J* ⟹ *up-to J* ≤ *n* ⟹ *i* ∈ *J* ⟹ *i* < *n* **for** *i*
using *up-to-less*[*of J i*] **by** *auto*
qed (*auto simp*: *up-to-def intro*!: *Least-le*)

lemma *up-to-iff-Ico*: *finite J* ⟹ *up-to J* ≤ *n* ⟷ *J* ⊆ {*0*..<*n*}
by (*auto simp*: *up-to-iff*)

lemma *up-to*: *finite J* ⟹ *J* ⊆ {*0*..< *up-to J*}
by (*auto simp*: *up-to-less*)

lemma *up-to-mono*: *J* ⊆ *H* ⟹ *finite H* ⟹ *up-to J* ≤ *up-to H*
by (*auto simp add*: *up-to-iff finite-subset up-to-less*)

definition *CI* :: *nat set* ⇒ (*nat* ⇒ *'a*) *measure* **where**
CI J = *distr* (*C 0* (*up-to J*) (λ*x*. *undefined*)) (*PiM J M*) (λ*f*. *restrict f J*)

sublocale *PF*: *projective-family UNIV CI*
unfolding *projective-family-def*
proof *safe*
show *finite J* ⟹ *prob-space* (*CI J*) **for** *J*
using *up-to*[*of J*] **unfolding** *CI-def*

by (*intro prob-space.prob-space-distr prob-space-C measurable-restrict*) *auto*
note *measurable-cong-sets*[*OF sets-C*, *simp*]
have [*simp*]: $J \subseteq H \Longrightarrow (\lambda f.\ restrict\ f\ J) \in measurable\ (Pi_M\ H\ M)\ (Pi_M\ J\ M)$ **for** $H\ J$
by (*auto intro*!: *measurable-restrict*)

show $J \subseteq H \Longrightarrow finite\ H \Longrightarrow CI\ J = distr\ (CI\ H)\ (Pi_M\ J\ M)\ (\lambda f.\ restrict\ f\ J)$ **for** $J\ H$
by (*simp add*: *CI-def distr-C*[*OF up-to-mono*[*of J H*]] *up-to up-to-mono distr-distr comp-def*
inf.absorb2 finite-subset)
qed

lemma *emeasure-CI'*:
finite J $\Longrightarrow$ $X \in sets\ (PiM\ J\ M) \Longrightarrow CI\ J\ X = C\ 0\ (up\text{-}to\ J)\ (\lambda\text{-}.\ undefined)\ (PF.emb\ \{0..<up\text{-}to\ J\}\ J\ X)$
unfolding *CI-def* **using** *up-to*[*of J*] **by** (*rule emeasure-distr-restrict*) (*auto simp*: *sets-C*)

lemma *emeasure-CI*:
$J \subseteq \{0..<n\} \Longrightarrow X \in sets\ (PiM\ J\ M) \Longrightarrow CI\ J\ X = C\ 0\ n\ (\lambda\text{-}.\ undefined)\ (PF.emb\ \{0..<n\}\ J\ X)$
apply (*subst emeasure-CI'*, *simp-all add*: *finite-subset*)
apply (*subst emeasure-C*[*symmetric*, *of up-to J n*])
apply (*auto simp*: *finite-subset up-to-iff-Ico up-to-less*)
apply (*subst prod-emb-trans*)
apply (*auto simp*: *up-to-less finite-subset up-to-iff-Ico*)
done

lemma *lim*:
assumes *J*: *finite J* **and** *X*: $X \in sets\ (PiM\ J\ M)$
shows $emeasure\ PF.lim\ (PF.emb\ UNIV\ J\ X) = emeasure\ (CI\ J)\ X$
proof (*rule PF.emeasure-lim*[*OF J subset-UNIV X*])
fix $J\ X'$ **assume** *J*[*simp*]: $\bigwedge i.\ finite\ (J\ i)$ **and** X'[*measurable*]: $\bigwedge i.\ X'\ i \in sets\ (Pi_M\ (J\ i)\ M)$
and *dec*: $decseq\ (\lambda i.\ PF.emb\ UNIV\ (J\ i)\ (X'\ i))$
define X **where** $X\ n =$
$(\bigcap i \in \{i.\ J\ i \subseteq \{0..<n\}\}.\ PF.emb\ \{0..<n\}\ (J\ i)\ (X'\ i)) \cap space\ (PiM\ \{0..<n\}\ M)$ **for** n

have *sets-X*[*measurable*]: $X\ n \in sets\ (PiM\ \{0..<n\}\ M)$ **for** n
by (*cases* $\{i.\ J\ i \subseteq \{0..<n\}\} = \{\}$)
(*simp-all add*: *X-def*, *auto intro*!: *sets.countable-INT' sets.Int*)

have *dec-X*: $n \leq m \Longrightarrow X\ m \subseteq PF.emb\ \{0..<m\}\ \{0..<n\}\ (X\ n)$ **for** $n\ m$
unfolding *X-def* **using** *ivl-subset*[*of 0 n 0 m*]
by (*cases* $\{i.\ J\ i \subseteq \{0..<n\}\} = \{\}$)
(*auto simp add*: *prod-emb-Int prod-emb-PiE space-PiM simp del*: *ivl-subset*)

```
  have dec-X': PF.emb {0..<n} (J j) (X' j) ⊆ PF.emb {0..<n} (J i) (X' i)
    if [simp]: J i ⊆ {0..<n} J j ⊆ {0..<n} i ≤ j for n i j
     by (rule PF.emb-preserve-mono[of {0..<n} UNIV]) (auto del: subsetI intro:
dec[THEN antimonoD])

  assume 0 < (INF i. CI (J i) (X' i))
  also have ... ≤ (INF i. C 0 i (λx. undefined) (X i))
  proof (intro INF-greatest)
    fix n
    interpret C: prob-space C 0 n (λx. undefined)
      by (rule prob-space-C) simp
    show (INF i. CI (J i) (X' i)) ≤ C 0 n (λx. undefined) (X n)
    proof cases
      assume {i. J i ⊆ {0..< n}} = {} with C.emeasure-space-1 show ?thesis
      by (auto simp add: X-def space-C intro!: INF-lower2[of 0] prob-space.measure-le-1
PF.prob-space-P)
    next
      assume ∗: {i. J i ⊆ {0..< n}} ≠ {}
      have (INF i. CI (J i) (X' i)) ≤
          (INF i∈{i. J i ⊆ {0..<n}}. C 0 n (λ-. undefined) (PF.emb {0..<n} (J i)
(X' i)))
        by (intro INF-superset-mono) (auto simp: emeasure-CI)
      also have ... = C 0 n (λ-. undefined) (⋂i∈{i. J i ⊆ {0..<n}}. (PF.emb
{0..<n} (J i) (X' i)))
          using ∗ by (intro emeasure-INT-decseq-subset[symmetric]) (auto intro!:
dec-X' del: subsetI simp: sets-C)
      also have ... = C 0 n (λ-. undefined) (X n)
        using ∗ by (auto simp add: X-def INT-extend-simps)
      finally show (INF i. CI (J i) (X' i)) ≤ C 0 n (λ-. undefined) (X n) .
    qed
  qed
  finally have pos: 0 < (INF i. C 0 i (λx. undefined) (X i)) .
  from less-INF-D[OF this, of 0] have X 0 ≠ {}
    by auto

  { fix ω n assume ω: ω ∈ space (PiM {0..<n} M)
    let ?C = λi. emeasure (C n i ω) (X (n + i))
    let ?C' = λi x. emeasure (C (Suc n) i (ω(n:=x))) (X (Suc n + i))
    have M: ⋀i. ?C' i ∈ borel-measurable (P n ω)
      using ω[measurable, simp] measurable-fun-upd[where J={0..<n}] by mea-
surable auto

    assume 0 < (INF i. ?C i)
    also have ... ≤ (INF i. emeasure (C n (1 + i) ω) (X (n + (1 + i))))
      by (intro INF-greatest INF-lower) auto
    also have ... = (INF i. ∫+x. ?C' i x ∂P n ω)
      using ω measurable-C[of Suc n]
      apply (intro INF-cong refl)
      apply (subst split-C[symmetric, OF ω])
```

```
    apply (subst emeasure-bind[OF - - sets-X])
    apply (simp-all del: C.simps add: space-C)
    apply measurable
    apply simp
    apply (simp add: bind-return[OF measurable-eP] nn-integral-eP)
    done
  also have ... = (∫⁺x. (INF i. ?C' i x) ∂P n ω)
  proof (rule nn-integral-monotone-convergence-INF-AE[symmetric])
    have (∫⁺x. ?C' 0 x ∂P n ω) ≤ (∫⁺x. 1 ∂P n ω)
      by (intro nn-integral-mono) (auto split: split-indicator)
    also have ... < ∞
      using prob-space-P[OF ω, THEN prob-space.emeasure-space-1] by simp
    finally show (∫⁺x. ?C' 0 x ∂P n ω) < ∞ .
  next
    show AE x in P n ω. ?C' (Suc i) x ≤ ?C' i x for i
    proof (rule AE-I2)
      fix x assume x ∈ space (P n ω)
      with ω have ω': ω(n := x) ∈ space (PiM {0..<Suc n} M)
        by (auto simp: space-P[OF ω] space-PiM PiE-iff extensional-def)
      with ω show ?C' (Suc i) x ≤ ?C' i x
        apply (subst emeasure-C[symmetric, of i Suc i])
        apply (auto intro!: emeasure-mono[OF dec-X] del: subsetI
                    simp: sets-C space-P)
        apply (subst sets-bind[OF sets-eP])
        apply (simp-all add: space-C space-P)
        done
    qed
  qed fact
  finally have (∫⁺x. (INF i. ?C' i x) ∂P n ω) ≠ 0
    by simp
  with M have ∃ᶠ x in ae-filter (P n ω). 0 < (INF i. ?C' i x)
    by (subst (asm) nn-integral-0-iff-AE)
       (auto intro!: borel-measurable-INF simp: Filter.not-eventually not-le
zero-less-iff-neq-zero)
  then have ∃ᶠ x in ae-filter (P n ω). x ∈ space (M n) ∧ 0 < (INF i. ?C' i x)
    by (rule frequently-mp[rotated]) (auto simp: space-P ω)
  then obtain x where x ∈ space (M n) 0 < (INF i. ?C' i x)
    by (auto dest: frequently-ex)
  from this(2)[THEN less-INF-D, of 0] this(2)
  have ∃x. ω(n := x) ∈ X (Suc n) ∧ 0 < (INF i. ?C' i x)
    by (intro exI[of - x]) (simp split: split-indicator-asm) }
note step = this

let ?ω = λω n x. (restrict ω {0..<n})(n := x)
let ?L = λω n r. INF i. emeasure (C (Suc n) i (?ω ω n r)) (X (Suc n + i))
have *: (⋀i. i < n ⟹ ?ω ω i (ω i) ∈ X (Suc i)) ⟹
  restrict ω {0..<n} ∈ space (Pi_M {0..<n} M) for ω n
  using sets.sets-into-space[OF sets-X, of n]
  by (cases n) (auto simp: atLeastLessThanSuc restrict-def[of - {}])
```

```
  have ∃ω. ∀ n. ?ω ω n (ω n) ∈ X (Suc n) ∧ 0 < ?L ω n (ω n)
  proof (rule dependent-wellorder-choice)
    fix n ω assume IH: ⋀i. i < n ⟹ ?ω ω i (ω i) ∈ X (Suc i) ∧ 0 < ?L ω i (ω
i)
    show ∃ r. ?ω ω n r ∈ X (Suc n) ∧ 0 < ?L ω n r
    proof (rule step)
      show restrict ω {0..<n} ∈ space (Pi_M {0..<n} M)
        using IH[THEN conjunct1] by (rule ∗)
      show 0 < (INF i. emeasure (C n i (restrict ω {0..<n})) (X (n + i)))
      proof (cases n)
        case 0 with pos show ?thesis
          by (simp add: CI-def restrict-def)
      next
        case (Suc i) then show ?thesis
          using IH[of i, THEN conjunct2] by (simp add: atLeastLessThanSuc)
      qed
    qed
  qed (simp cong: restrict-cong)
  then obtain ω where ω: ⋀n. ?ω ω n (ω n) ∈ X (Suc n)
    by auto
  from this[THEN ∗] have ω-space: ω ∈ space (PiM UNIV M)
    by (auto simp: space-PiM PiE-iff Ball-def)
  have ∗: ω ∈ PF.emb UNIV {0..<n} (X n) for n
  proof (cases n)
    case 0 with ω-space ‹X 0 ≠ {}› sets.sets-into-space[OF sets-X, of 0] show
?thesis
      by (auto simp add: space-PiM prod-emb-def restrict-def PiE-iff)
  next
    case (Suc i) then show ?thesis
      using ω[of i] ω-space by (auto simp: prod-emb-def space-PiM PiE-iff atLeast-
LessThanSuc)
  qed
  have ∗∗: {i. J i ⊆ {0..<up-to (J n)}} ≠ {} for n
    by (auto intro!: exI[of - n] up-to J)
  have ω ∈ PF.emb UNIV (J n) (X′ n) for n
   using ∗[of up-to (J n)] up-to[of J n] by (simp add: X-def prod-emb-Int prod-emb-INT[OF
∗∗])
  then show (⋂ i. PF.emb UNIV (J i) (X′ i)) ≠ {}
    by auto
qed

lemma distr-lim: assumes J[simp]: finite J shows distr PF.lim (PiM J M) (λx.
restrict x J) = CI J
  apply (rule measure-eqI)
  apply (simp add: CI-def)
 apply (simp add: emeasure-distr measurable-cong-sets[OF PF.sets-lim] lim[symmetric]
prod-emb-def space-PiM)
  done
```

end

lemma (**in** *product-prob-space*) *emeasure-lim-emb*:
assumes ∗: *finite J* $J \subseteq I$ *X* ∈ *sets* (*PiM J M*)
shows *emeasure lim* (*emb I J X*) = *emeasure* (*Pi*$_M$ *J M*) *X*
proof (*rule emeasure-lim*[*OF* ∗], *goal-cases*)
case (*1 J X*)

have ∃ *Q*. (∀ *i*. *sets Q* = *PiM* (⋃ *i*. *J i*) *M* ∧ *distr Q* (*PiM* (*J i*) *M*) (λ*x*. *restrict x* (*J i*)) = *Pi*$_M$ (*J i*) *M*)
proof *cases*
assume *finite* (⋃ *i*. *J i*)
then have *distr* (*PiM* (⋃ *i*. *J i*) *M*) (*Pi*$_M$ (*J i*) *M*) (λ*x*. *restrict x* (*J i*)) = *Pi*$_M$ (*J i*) *M* **for** *i*
by (*intro distr-restrict*[*symmetric*]) *auto*
then show *?thesis*
by *auto*
next
assume *inf*: *infinite* (⋃ *i*. *J i*)
moreover have *count*: *countable* (⋃ *i*. *J i*)
using *1*(*3*) **by** (*auto intro*: *countable-finite*)
define *f* **where** *f* = *from-nat-into* (⋃ *i*. *J i*)
define *t* **where** *t* = *to-nat-on* (⋃ *i*. *J i*)
have *ft*[*simp*]: *x* ∈ *J i* ⟹ *f* (*t x*) = *x* **for** *x i*
unfolding *f-def t-def* **using** *inf count* **by** (*intro from-nat-into-to-nat-on*) *auto*
have *tf*[*simp*]: *t* (*f i*) = *i* **for** *i*
unfolding *t-def f-def* **by** (*intro to-nat-on-from-nat-into-infinite inf count*)
have *inj-t*: *inj-on t* (⋃ *i*. *J i*)
using *count* **by** (*auto simp*: *t-def*)
then have *inj-t-J*: *inj-on t* (*J i*) **for** *i*
by (*rule subset-inj-on*) *auto*
interpret *IT*: *Ionescu-Tulcea* λ*i* ω. *M* (*f i*) λ*i*. *M* (*f i*)
by *standard auto*
interpret *Mf*: *product-prob-space* λ*x*. *M* (*f x*) *UNIV*
by *standard*
have *C-eq-PiM*: *IT.C 0 n* (λ-. *undefined*) = *PiM* {*0*..<*n*} (λ*x*. *M* (*f x*)) **for** *n*
proof (*induction n*)
case *0* **then show** *?case*
by (*auto simp*: *PiM-empty intro*!: *measure-eqI dest*!: *subset-singletonD*)
next
case (*Suc n*) **then show** *?case*
apply (*auto intro*!: *measure-eqI simp*: *sets-bind*[*OF IT.sets-eP*] *emeasure-bind*[*OF - IT.measurable-eP*])
apply (*auto simp*: *Mf.product-nn-integral-insert nn-integral-indicator*[*symmetric*] *atLeastLessThanSuc IT.emeasure-eP space-PiM*
split: *split-indicator simp del*: *nn-integral-indicator intro*!: *nn-integral-cong*)
done
qed

```
  have CI-eq-PiM: IT.CI X = PiM X (λx. M (f x)) if X: finite X for X
  by (auto simp: IT.up-to-less X IT.CI-def C-eq-PiM intro!: Mf.distr-restrict[symmetric])

  let ?Q = distr IT.PF.lim (PiM (⋃ i. J i) M) (λω. λx∈⋃ i. J i. ω (t x))
  { fix i
    have distr ?Q (Pi_M (J i) M) (λx. restrict x (J i)) =
      distr IT.PF.lim (Pi_M (J i) M) ((λω. λn∈J i. ω (t n)) ∘ (λω. restrict ω (t‘J
i)))
     proof (subst distr-distr)
      have (λω. ω (t x)) ∈ measurable (Pi_M UNIV (λx. M (f x))) (M x) if x: x
∈ J i for x i
       using measurable-component-singleton[of t x UNIV λx. M (f x)] unfolding
ft[OF x] by simp
       then show (λω. λx∈⋃ i. J i. ω (t x)) ∈ measurable IT.PF.lim (Pi_M (⋃ (J
‘ UNIV)) M)
      by (auto intro!: measurable-restrict simp: measurable-cong-sets[OF IT.PF.sets-lim
refl])
   qed (auto intro!: distr-cong measurable-restrict measurable-component-singleton)
     also have ... = distr (distr IT.PF.lim (PiM (t‘J i) (λx. M (f x))) (λω.
restrict ω (t‘J i))) (Pi_M (J i) M) (λω. λn∈J i. ω (t n))
     proof (intro distr-distr[symmetric])
      have (λω. ω (t x)) ∈ measurable (Pi_M (t‘J i) (λx. M (f x))) (M x) if x: x
∈ J i for x
       using measurable-component-singleton[of t x t‘J i λx. M (f x)] x unfolding
ft[OF x] by auto
       then show (λω. λn∈J i. ω (t n)) ∈ measurable (Pi_M (t ‘ J i) (λx. M (f
x))) (Pi_M (J i) M)
        by (auto intro!: measurable-restrict)
   qed (auto intro!: measurable-restrict simp: measurable-cong-sets[OF IT.PF.sets-lim
refl])
     also have ... = distr (PiM (t‘J i) (λx. M (f x))) (Pi_M (J i) M) (λω. λn∈J
i. ω (t n))
       using ‹finite (J i)› by (subst IT.distr-lim) (auto simp: CI-eq-PiM)
     also have ... = Pi_M (J i) M
       using Mf.distr-reorder[of t J i] by (simp add: 1 inj-t-J cong: PiM-cong)
    finally have distr ?Q (Pi_M (J i) M) (λx. restrict x (J i)) = Pi_M (J i) M . }
  then show ∃ Q. ∀ i. sets Q = PiM (⋃ i. J i) M ∧ distr Q (Pi_M (J i) M) (λx.
restrict x (J i)) = Pi_M (J i) M
    by (intro exI[of - ?Q]) auto
 qed
 then obtain Q where sets-Q: sets Q = PiM (⋃ i. J i) M
   and Q: ⋀i. distr Q (PiM (J i) M) (λx. restrict x (J i)) = Pi_M (J i) M by
blast

 from 1 interpret Q: prob-space Q
   by (intro prob-space-distrD[of λx. restrict x (J 0) Q PiM (J 0) M])
     (auto simp: Q measurable-cong-sets[OF sets-Q]
          intro!: prob-space-P measurable-restrict measurable-component-singleton)
```

have *0* < (*INF i. emeasure* (*Pi*$_M$ (*J i*) *M*) (*X i*)) **by** *fact*
also have ... = (*INF i. emeasure Q* (*emb* (⋃ *i. J i*) (*J i*) (*X i*)))
by (*simp add*: *emeasure-distr-restrict*[*OF - sets-Q 1*(*4*), *symmetric*] *SUP-upper Q*)
also have ... = *emeasure Q* (⋂ *i. emb* (⋃ *i. J i*) (*J i*) (*X i*))
proof (*rule INF-emeasure-decseq*)
from *1* **show** *decseq* (λ*n. emb* (⋃ *i. J i*) (*J n*) (*X n*))
by (*intro antimonoI emb-preserve-mono*[**where** *X*=*emb* (⋃ *i. J i*) (*J n*) (*X n*) **and** *L*=*I* **and** *J*=⋃ *i. J i* **for** *n*]
measurable-prod-emb)
(*auto simp*: *SUP-least SUP-upper antimono-def*)
qed (*insert 1*, *auto simp*: *sets-Q*)
finally have (⋂ *i. emb* (⋃ *i. J i*) (*J i*) (*X i*)) ≠ {}
by *auto*
moreover have (⋂ *i. emb I* (*J i*) (*X i*)) = {} ⟹ (⋂ *i. emb* (⋃ *i. J i*) (*J i*) (*X i*)) = {}
using *1* **by** (*intro emb-injective*[*of* ⋃ *i. J i I* - {}] *sets.countable-INT*) (*auto simp*: *SUP-least SUP-upper*)
ultimately show *?case* **by** *auto*
qed

end

9 Infinite Product Measure

theory *Infinite-Product-Measure*
imports *Probability-Measure Projective-Family*
begin

lemma (**in** *product-prob-space*) *distr-PiM-restrict-finite*:
assumes *finite J J* ⊆ *I*
shows *distr* (*PiM I M*) (*PiM J M*) (λ*x. restrict x J*) = *PiM J M*
proof (*rule PiM-eqI*)
fix *X* **assume** *X*: ⋀*i. i* ∈ *J* ⟹ *X i* ∈ *sets* (*M i*)
{ **fix** *J X* **assume** *J*: *J* ≠ {} ∨ *I* = {} *finite J J* ⊆ *I* **and** *X*: ⋀*i. i* ∈ *J* ⟹ *X i* ∈ *sets* (*M i*)
have *emeasure* (*PiM I M*) (*emb I J* (*Pi*$_E$ *J X*)) = (∏ *i*∈*J. M i* (*X i*))
proof (*subst emeasure-extend-measure-Pair*[*OF PiM-def*, **where** μ′=*lim*], *goal-cases*)
case *1* **then show** *?case*
by (*simp add*: *M.emeasure-space-1 emeasure-PiM Pi-iff sets-PiM-I-finite emeasure-lim-emb*)
next
case (*2 J X*)
then have *emb I J* (*Pi*$_E$ *J X*) ∈ *sets* (*PiM I M*)
by (*intro measurable-prod-emb sets-PiM-I-finite*) *auto*
from *this*[*THEN sets.sets-into-space*] **show** *?case*
by (*simp add*: *space-PiM*)
qed (*insert assms J X*, *simp-all del*: *sets-lim*
add: *M.emeasure-space-1 sets-lim*[*symmetric*] *emeasure-countably-additive*

emeasure-positive) **}**
note $* = this$

have *emeasure* (*PiM I M*) (*emb I J* (*Pi*$_E$ *J X*)) = ($\prod i{\in}J.$ *M i* (*X i*))
proof (*cases* $J \neq \{\} \vee I = \{\}$)
case *False*
then obtain *i* **where** *i*: $J = \{\}$ $i \in I$ **by** *auto*
then have *emb I* {} {$\lambda x.$ *undefined*} = *emb I* {*i*} (Π_E $i{\in}\{i\}.$ *space* (*M i*))
by (*auto simp*: *space-PiM prod-emb-def*)
with *i* **show** *?thesis*
by (*simp add*: $*$ *M.emeasure-space-1*)
next
case *True*
then show *?thesis*
by (*simp add*: $*$[*OF - assms X*])
qed
with *assms* **show** *emeasure* (*distr* (*Pi*$_M$ *I M*) (*Pi*$_M$ *J M*) ($\lambda x.$ *restrict x J*)) (*Pi*$_E$ *J X*) = ($\prod i{\in}J.$ *emeasure* (*M i*) (*X i*))
by (*subst emeasure-distr-restrict*[*OF - refl*]) (*auto intro*!: *sets-PiM-I-finite X*)
qed (*insert assms*, *auto*)

lemma (**in** *product-prob-space*) *emeasure-PiM-emb'*:
$J \subseteq I \Longrightarrow$ *finite J* $\Longrightarrow$ *X* $\in$ *sets* (*PiM J M*) $\Longrightarrow$ *emeasure* (*Pi*$_M$ *I M*) (*emb I J X*) = *PiM J M X*
by (*subst distr-PiM-restrict-finite*[*symmetric*, *of J*])
(*auto intro*!: *emeasure-distr-restrict*[*symmetric*])

lemma (**in** *product-prob-space*) *emeasure-PiM-emb*:
$J \subseteq I \Longrightarrow$ *finite J* $\Longrightarrow$ ($\bigwedge i.$ $i \in J \Longrightarrow X\ i \in$ *sets* (*M i*)) $\Longrightarrow$
emeasure (*Pi*$_M$ *I M*) (*emb I J* (*Pi*$_E$ *J X*)) = ($\prod$ $i{\in}J.$ *emeasure* (*M i*) (*X i*))
by (*subst emeasure-PiM-emb'*) (*auto intro*!: *emeasure-PiM*)

sublocale *product-prob-space* $\subseteq$ *P?*: *prob-space Pi*$_M$ *I M*
proof
have $*$: *emb I* {} {$\lambda x.$ *undefined*} = *space* (*PiM I M*)
by (*auto simp*: *prod-emb-def space-PiM*)
show *emeasure* (*Pi*$_M$ *I M*) (*space* (*Pi*$_M$ *I M*)) = *1*
using *emeasure-PiM-emb*[*of* {} λ-. {}] **by** (*simp add*: $*$)
qed

lemma *prob-space-PiM*:
assumes *M*: $\bigwedge i.$ $i \in I \Longrightarrow$ *prob-space* (*M i*) **shows** *prob-space* (*PiM I M*)
proof −
let *?M* = $\lambda i.$ *if* $i \in I$ *then M i else count-space* {*undefined*}
interpret *M'*: *prob-space ?M i* **for** *i*
using *M* **by** (*cases* $i \in I$) (*auto intro*!: *prob-spaceI*)
interpret *product-prob-space ?M I*
by *unfold-locales*
have *prob-space* (Π_M $i{\in}I.$ *?M i*)

by *unfold-locales*
also have $(\Pi_M\ i{\in}I.\ ?M\ i) = (\Pi_M\ i{\in}I.\ M\ i)$
by (*intro PiM-cong*) *auto*
finally show *?thesis* **.**
qed

lemma (**in** *product-prob-space*) *emeasure-PiM-Collect*:
assumes *X*: $J \subseteq I$ *finite* J $\bigwedge i.\ i \in J \Longrightarrow X\ i \in sets\ (M\ i)$
shows *emeasure* $(Pi_M\ I\ M)$ $\{x{\in}space\ (Pi_M\ I\ M).\ \forall\, i{\in}J.\ x\ i \in X\ i\} = (\prod\ i{\in}J.$ *emeasure* $(M\ i)\ (X\ i))$
proof −
have $\{x{\in}space\ (Pi_M\ I\ M).\ \forall\, i{\in}J.\ x\ i \in X\ i\} = emb\ I\ J\ (Pi_E\ J\ X)$
unfolding *prod-emb-def* **using** *assms* **by** (*auto simp*: *space-PiM Pi-iff*)
with *emeasure-PiM-emb*[*OF assms*] **show** *?thesis* **by** *simp*
qed

lemma (**in** *product-prob-space*) *emeasure-PiM-Collect-single*:
assumes *X*: $i \in I$ $A \in sets\ (M\ i)$
shows *emeasure* $(Pi_M\ I\ M)$ $\{x{\in}space\ (Pi_M\ I\ M).\ x\ i \in A\} = emeasure\ (M\ i)\ A$
using *emeasure-PiM-Collect*[*of* $\{i\}$ $\lambda i.\ A$] *assms*
by *simp*

lemma (**in** *product-prob-space*) *measure-PiM-emb*:
assumes $J \subseteq I$ *finite* J $\bigwedge i.\ i \in J \Longrightarrow X\ i \in sets\ (M\ i)$
shows *measure* $(PiM\ I\ M)$ $(emb\ I\ J\ (Pi_E\ J\ X)) = (\prod\ i{\in}J.\ measure\ (M\ i)\ (X\ i))$
using *emeasure-PiM-emb*[*OF assms*]
unfolding *emeasure-eq-measure M.emeasure-eq-measure*
by (*simp add*: *prod-ennreal measure-nonneg prod-nonneg*)

lemma *sets-Collect-single'*:
$i \in I \Longrightarrow \{x{\in}space\ (M\ i).\ P\ x\} \in sets\ (M\ i) \Longrightarrow \{x{\in}space\ (PiM\ I\ M).\ P\ (x\ i)\} \in sets\ (PiM\ I\ M)$
using *sets-Collect-single*[*of i I* $\{x{\in}space\ (M\ i).\ P\ x\}$ *M*]
by (*simp add*: *space-PiM PiE-iff cong*: *conj-cong*)

lemma (**in** *finite-product-prob-space*) *finite-measure-PiM-emb*:
$(\bigwedge i.\ i \in I \Longrightarrow A\ i \in sets\ (M\ i)) \Longrightarrow measure\ (PiM\ I\ M)\ (Pi_E\ I\ A) = (\prod i{\in}I.\ measure\ (M\ i)\ (A\ i))$
using *measure-PiM-emb*[*of I A*] *finite-index prod-emb-PiE-same-index*[*OF sets.sets-into-space*, *of I A M*]
by *auto*

lemma (**in** *product-prob-space*) *PiM-component*:
assumes $i \in I$
shows *distr* $(PiM\ I\ M)$ $(M\ i)$ $(\lambda\omega.\ \omega\ i) = M\ i$
proof (*rule measure-eqI*[*symmetric*])
fix *A* **assume** $A \in sets\ (M\ i)$
moreover have $((\lambda\omega.\ \omega\ i) -` A \cap space\ (PiM\ I\ M)) = \{x{\in}space\ (PiM\ I\ M).\ x$

$i \in A\}$
by *auto*
ultimately show *emeasure* $(M\ i)\ A =$ *emeasure* $(distr\ (PiM\ I\ M)\ (M\ i)\ (\lambda\omega.\ \omega\ i))\ A$
by (*auto simp*: ‹$i \in I$› *emeasure-distr measurable-component-singleton emeasure-PiM-Collect-single*)
qed *simp*

lemma (**in** *product-prob-space*) *PiM-eq*:
assumes M': $sets\ M' = sets\ (PiM\ I\ M)$
assumes *eq*: $\bigwedge J\ F.\ finite\ J \Longrightarrow J \subseteq I \Longrightarrow (\bigwedge j.\ j \in J \Longrightarrow F\ j \in sets\ (M\ j)) \Longrightarrow$
emeasure M' $(prod\text{-}emb\ I\ M\ J\ (\Pi_E\ j{\in}J.\ F\ j)) = (\prod j{\in}J.\ emeasure\ (M\ j)\ (F\ j))$
shows $M' = (PiM\ I\ M)$
proof (*rule measure-eqI-PiM-infinite*[*symmetric*, *OF refl M'*])
show *finite-measure* $(Pi_M\ I\ M)$
by *standard* (*simp add*: *P.emeasure-space-1*)
qed (*simp add*: *eq emeasure-PiM-emb*)

lemma (**in** *product-prob-space*) *AE-component*: $i \in I \Longrightarrow AE\ x\ in\ M\ i.\ P\ x \Longrightarrow AE\ x\ in\ PiM\ I\ M.\ P\ (x\ i)$
apply (*rule AE-distrD*[*of* $\lambda\omega.\ \omega\ i\ PiM\ I\ M\ M\ i\ P$])
apply *simp*
apply (*subst PiM-component*)
apply *simp-all*
done

lemma *emeasure-PiM-emb*:
assumes M: $\bigwedge i.\ i \in I \Longrightarrow prob\text{-}space\ (M\ i)$
assumes J: $J \subseteq I\ finite\ J$ **and** A: $\bigwedge i.\ i \in J \Longrightarrow A\ i \in sets\ (M\ i)$
shows *emeasure* $(Pi_M\ I\ M)\ (prod\text{-}emb\ I\ M\ J\ (Pi_E\ J\ A)) = (\prod i{\in}J.\ emeasure\ (M\ i)\ (A\ i))$
proof −
let *?M* $= \lambda i.$ *if* $i \in I$ *then* $M\ i$ *else count-space* $\{undefined\}$
interpret M': *prob-space ?M i* **for** i
using M **by** (*cases* $i \in I$) (*auto intro*!: *prob-spaceI*)
interpret P: *product-prob-space ?M I*
by *unfold-locales*
have *emeasure* $(Pi_M\ I\ M)\ (prod\text{-}emb\ I\ M\ J\ (Pi_E\ J\ A)) = emeasure\ (Pi_M\ I\ ?M)\ (P.emb\ I\ J\ (Pi_E\ J\ A))$
by (*auto simp*: *prod-emb-def PiE-iff intro*!: *arg-cong2*[**where** *f*=*emeasure*] *PiM-cong*)
also have $\ldots = (\prod i{\in}J.\ emeasure\ (M\ i)\ (A\ i))$
using $J\ A$ **by** (*subst P.emeasure-PiM-emb*[*OF J*]) (*auto intro*!: *prod.cong*)
finally show *?thesis* **.**
qed

lemma *distr-pair-PiM-eq-PiM*:
fixes i' :: $'i$ **and** I :: $'i\ set$ **and** M :: $'i \Rightarrow 'a\ measure$

assumes M: $\bigwedge i.\ i \in I \Longrightarrow$ *prob-space* $(M\ i)$ *prob-space* $(M\ i')$
shows *distr* $(M\ i' \bigotimes_M (\Pi_M\ i{\in}I.\ M\ i))\ (\Pi_M\ i{\in}\mathit{insert}\ i'\ I.\ M\ i)\ (\lambda(x,\ X).\ X(i' := x)) =$
$(\Pi_M\ i{\in}\mathit{insert}\ i'\ I.\ M\ i)$ (**is** *?L* = -)
proof (*rule measure-eqI-PiM-infinite*[*symmetric*, *OF refl*])
interpret M': *prob-space* $M\ i'$ **by** *fact*
interpret I: *prob-space* $(\Pi_M\ i{\in}I.\ M\ i)$
using M **by** (*intro prob-space-PiM*) *auto*
interpret I': *prob-space* $(\Pi_M\ i{\in}\mathit{insert}\ i'\ I.\ M\ i)$
using M **by** (*intro prob-space-PiM*) *auto*
show *finite-measure* $(\Pi_M\ i{\in}\mathit{insert}\ i'\ I.\ M\ i)$
by *unfold-locales*
fix $J\ A$ **assume** J: *finite* $J\ J \subseteq \mathit{insert}\ i'\ I$ **and** A: $\bigwedge i.\ i \in J \Longrightarrow A\ i \in \mathit{sets}\ (M\ i)$
let *?X* = *prod-emb* $(\mathit{insert}\ i'\ I)\ M\ J\ (Pi_E\ J\ A)$
have $Pi_M\ (\mathit{insert}\ i'\ I)\ M$ *?X* = $(\prod i{\in}J.\ M\ i\ (A\ i))$
using $M\ J\ A$ **by** (*intro emeasure-PiM-emb*) *auto*
also have ... = $M\ i'\ (\mathit{if}\ i' \in J\ \mathit{then}\ (A\ i')\ \mathit{else}\ \mathit{space}\ (M\ i')) * (\prod i{\in}J{-}\{i'\}.\ M\ i\ (A\ i))$
using *prod.insert-remove*[*of* $J\ \lambda i.\ M\ i\ (A\ i)\ i'$] $J\ M'$.*emeasure-space-1*
by (*cases* $i' \in J$) (*auto simp*: *insert-absorb*)
also have $(\prod i{\in}J{-}\{i'\}.\ M\ i\ (A\ i)) = Pi_M\ I\ M\ (\mathit{prod\text{-}emb}\ I\ M\ (J - \{i'\})\ (Pi_E\ (J - \{i'\})\ A))$
using $M\ J\ A$ **by** (*intro emeasure-PiM-emb*[*symmetric*]) *auto*
also have $M\ i'\ (\mathit{if}\ i' \in J\ \mathit{then}\ (A\ i')\ \mathit{else}\ \mathit{space}\ (M\ i')) * \ldots =$
$(M\ i' \bigotimes_M Pi_M\ I\ M)\ ((\mathit{if}\ i' \in J\ \mathit{then}\ (A\ i')\ \mathit{else}\ \mathit{space}\ (M\ i')) \times \mathit{prod\text{-}emb}\ I\ M\ (J - \{i'\})\ (Pi_E\ (J - \{i'\})\ A))$
using $J\ A$ **by** (*intro* I.*emeasure-pair-measure-Times*[*symmetric*] *sets-PiM-I*) *auto*
also have $((\mathit{if}\ i' \in J\ \mathit{then}\ (A\ i')\ \mathit{else}\ \mathit{space}\ (M\ i')) \times \mathit{prod\text{-}emb}\ I\ M\ (J - \{i'\})\ (Pi_E\ (J - \{i'\})\ A)) =$
$(\lambda(x,\ X).\ X(i' := x)) -`$ *?X* $\cap$ *space* $(M\ i' \bigotimes_M Pi_M\ I\ M)$
using A[*of* i', *THEN sets.sets-into-space*] **unfolding** *set-eq-iff*
by (*simp add*: *prod-emb-def space-pair-measure space-PiM PiE-fun-upd ac-simps cong*: *conj-cong*)
(*auto simp add*: *Pi-iff Ball-def all-conj-distrib*)
finally show $Pi_M\ (\mathit{insert}\ i'\ I)\ M$ *?X* = *?L ?X*
using $J\ A$ **by** (*simp add*: *emeasure-distr*)
qed *simp*

lemma *distr-PiM-reindex*:
assumes M: $\bigwedge i.\ i \in K \Longrightarrow$ *prob-space* $(M\ i)$
assumes f: *inj-on* $f\ I\ f \in I \to K$
shows *distr* $(Pi_M\ K\ M)\ (\Pi_M\ i{\in}I.\ M\ (f\ i))\ (\lambda\omega.\ \lambda n{\in}I.\ \omega\ (f\ n)) = (\Pi_M\ i{\in}I.\ M\ (f\ i))$
(**is** *distr ?K ?I ?t* = *?I*)
proof (*rule measure-eqI-PiM-infinite*[*symmetric*, *OF refl*])
interpret *prob-space ?I*
using $f\ M$ **by** (*intro prob-space-PiM*) *auto*

show *finite-measure ?I*
by *unfold-locales*
fix *A J* **assume** *J*: *finite J J* ⊆ *I* **and** *A*: ⋀*i*. *i* ∈ *J* ⟹ *A i* ∈ *sets* (*M* (*f i*))
have [*simp*]: *i* ∈ *J* ⟹ *the-inv-into I f* (*f i*) = *i* **for** *i*
using *J f* **by** (*intro the-inv-into-f-f*) *auto*
have *?I* (*prod-emb I* (λ*i*. *M* (*f i*)) *J* (*Pi*$_E$ *J A*)) = (∏ *j*∈*J*. *M* (*f j*) (*A j*))
using *f J A* **by** (*intro emeasure-PiM-emb M*) *auto*
also have ... = (∏ *j*∈*f*‘*J*. *M j* (*A* (*the-inv-into I f j*)))
using *f J* **by** (*subst prod.reindex*) (*auto intro*!: *prod.cong intro*: *inj-on-subset simp*: *the-inv-into-f-f*)
also have ... = *?K* (*prod-emb K M* (*f*‘*J*) (Π$_E$ *j*∈*f*‘*J*. *A* (*the-inv-into I f j*)))
using *f J A* **by** (*intro emeasure-PiM-emb*[*symmetric*] *M*) (*auto simp*: *the-inv-into-f-f*)
also have *prod-emb K M* (*f*‘*J*) (Π$_E$ *j*∈*f*‘*J*. *A* (*the-inv-into I f j*)) = *?t* −‘ *prod-emb I* (λ*i*. *M* (*f i*)) *J* (*Pi*$_E$ *J A*) ∩ *space ?K*
using *f J A* **by** (*auto simp*: *prod-emb-def space-PiM Pi-iff PiE-iff Int-absorb1*)
also have *?K* ... = *distr ?K ?I ?t* (*prod-emb I* (λ*i*. *M* (*f i*)) *J* (*Pi*$_E$ *J A*))
using *f J A* **by** (*intro emeasure-distr*[*symmetric*] *sets-PiM-I*) (*auto simp*: *Pi-iff*)
finally show *?I* (*prod-emb I* (λ*i*. *M* (*f i*)) *J* (*Pi*$_E$ *J A*)) = *distr ?K ?I ?t* (*prod-emb I* (λ*i*. *M* (*f i*)) *J* (*Pi*$_E$ *J A*)) **.**
qed *simp*

lemma *distr-PiM-component*:
assumes *M*: ⋀*i*. *i* ∈ *I* ⟹ *prob-space* (*M i*)
assumes *i* ∈ *I*
shows *distr* (*Pi*$_M$ *I M*) (*M i*) (λω. ω *i*) = *M i*
proof −
have ∗: (λω. ω *i*) −‘ *A* ∩ *space* (*Pi*$_M$ *I M*) = *prod-emb I M* {*i*} (Π$_E$ *i*′∈{*i*}. *A*) **for** *A*
by (*auto simp*: *prod-emb-def space-PiM*)
show *?thesis*
apply (*intro measure-eqI*)
apply (*auto simp add*: *emeasure-distr* ‹*i*∈*I*› ∗ *emeasure-PiM-emb M*)
apply (*subst emeasure-PiM-emb*)
apply (*simp-all add*: *M* ‹*i*∈*I*›)
done
qed

lemma *AE-PiM-component*:
(⋀*i*. *i* ∈ *I* ⟹ *prob-space* (*M i*)) ⟹ *i* ∈ *I* ⟹ *AE x in M i*. *P x* ⟹ *AE x in PiM I M*. *P* (*x i*)
using *AE-distrD*[*of* λ*x*. *x i PiM I M M i*]
by (*subst* (*asm*) *distr-PiM-component*[*of I - i*]) (*auto intro*: *AE-distrD*[*of* λ*x*. *x i - - P*])

lemma *decseq-emb-PiE*:
incseq J ⟹ *decseq* (λ*i*. *prod-emb I M* (*J i*) (Π$_E$ *j*∈*J i*. *X j*))
by (*fastforce simp*: *decseq-def prod-emb-def incseq-def Pi-iff*)

9.1 Sequence space

definition *comb-seq* :: *nat* ⇒ (*nat* ⇒ *'a*) ⇒ (*nat* ⇒ *'a*) ⇒ (*nat* ⇒ *'a*) **where**
comb-seq i ω ω' j = (*if j* < *i then ω j else ω'* (*j* − *i*))

lemma *split-comb-seq*: *P* (*comb-seq i ω ω' j*) ⟷ (*j* < *i* ⟶ *P* (*ω j*)) ∧ (∀ *k*. *j* = *i* + *k* ⟶ *P* (*ω' k*))
by (*auto simp*: *comb-seq-def not-less*)

lemma *split-comb-seq-asm*: *P* (*comb-seq i ω ω' j*) ⟷ ¬ ((*j* < *i* ∧ ¬ *P* (*ω j*)) ∨ (∃ *k*. *j* = *i* + *k* ∧ ¬ *P* (*ω' k*)))
by (*auto simp*: *comb-seq-def*)

lemma *measurable-comb-seq*:
(λ(*ω*, *ω'*). *comb-seq i ω ω'*) ∈ *measurable* ((Π$_M$ *i*∈*UNIV*. *M*) ⨂$_M$ (Π$_M$ *i*∈*UNIV*. *M*)) (Π$_M$ *i*∈*UNIV*. *M*)
proof (*rule measurable-PiM-single*)
show (λ(*ω*, *ω'*). *comb-seq i ω ω'*) ∈ *space* ((Π$_M$ *i*∈*UNIV*. *M*) ⨂$_M$ (Π$_M$ *i*∈*UNIV*. *M*)) → (*UNIV* →$_E$ *space M*)
by (*auto simp*: *space-pair-measure space-PiM PiE-iff split*: *split-comb-seq*)
fix *j* :: *nat* **and** *A* **assume** *A*: *A* ∈ *sets M*
then have ∗: {*ω* ∈ *space* ((Π$_M$ *i*∈*UNIV*. *M*) ⨂$_M$ (Π$_M$ *i*∈*UNIV*. *M*)). *case-prod* (*comb-seq i*) *ω j* ∈ *A*} =
(*if j* < *i then* {*ω* ∈ *space* (Π$_M$ *i*∈*UNIV*. *M*). *ω j* ∈ *A*} × *space* (Π$_M$ *i*∈*UNIV*. *M*)
else space (Π$_M$ *i*∈*UNIV*. *M*) × {*ω* ∈ *space* (Π$_M$ *i*∈*UNIV*. *M*). *ω* (*j* − *i*) ∈ *A*})
by (*auto simp*: *space-PiM space-pair-measure comb-seq-def dest*: *sets.sets-into-space*)
show {*ω* ∈ *space* ((Π$_M$ *i*∈*UNIV*. *M*) ⨂$_M$ (Π$_M$ *i*∈*UNIV*. *M*)). *case-prod* (*comb-seq i*) *ω j* ∈ *A*} ∈ *sets* ((Π$_M$ *i*∈*UNIV*. *M*) ⨂$_M$ (Π$_M$ *i*∈*UNIV*. *M*))
unfolding ∗ **by** (*auto simp*: *A intro*!: *sets-Collect-single*)
qed

lemma *measurable-comb-seq'*[*measurable* (*raw*)]:
assumes *f*: *f* ∈ *measurable N* (Π$_M$ *i*∈*UNIV*. *M*) **and** *g*: *g* ∈ *measurable N* (Π$_M$ *i*∈*UNIV*. *M*)
shows (λ*x*. *comb-seq i* (*f x*) (*g x*)) ∈ *measurable N* (Π$_M$ *i*∈*UNIV*. *M*)
using *measurable-compose*[*OF measurable-Pair*[*OF f g*] *measurable-comb-seq*] **by** *simp*

lemma *comb-seq-0*: *comb-seq 0 ω ω' = ω'*
by (*auto simp add*: *comb-seq-def*)

lemma *comb-seq-Suc*: *comb-seq* (*Suc n*) *ω ω' = comb-seq n ω* (*case-nat* (*ω n*) *ω'*)
by (*auto simp add*: *comb-seq-def not-less less-Suc-eq le-imp-diff-is-add intro*!: *ext split*: *nat.split*)

lemma *comb-seq-Suc-0*[*simp*]: *comb-seq* (*Suc 0*) *ω* = *case-nat* (*ω 0*)
by (*intro ext*) (*simp add*: *comb-seq-Suc comb-seq-0*)

lemma *comb-seq-less*: $i < n \Longrightarrow$ *comb-seq* $n\ \omega\ \omega'\ i = \omega\ i$
by (*auto split*: *split-comb-seq*)

lemma *comb-seq-add*: *comb-seq* $n\ \omega\ \omega'\ (i + n) = \omega'\ i$
by (*auto split*: *nat.split split-comb-seq*)

lemma *case-nat-comb-seq*: *case-nat* s' (*comb-seq* $n\ \omega\ \omega'$) $(i + n)$ = *case-nat* (*case-nat* $s'\ \omega\ n$) $\omega'\ i$
by (*auto split*: *nat.split split-comb-seq*)

lemma *case-nat-comb-seq'*:
case-nat s (*comb-seq* $i\ \omega\ \omega'$) = *comb-seq* (*Suc* i) (*case-nat* $s\ \omega$) ω'
by (*auto split*: *split-comb-seq nat.split*)

locale *sequence-space* = *product-prob-space* $\lambda i.\ M$ *UNIV* :: *nat set* **for** M
begin

abbreviation $S \equiv \Pi_M\ i \in UNIV :: nat\ set.\ M$

lemma *infprod-in-sets*[*intro*]:
fixes E :: *nat* $\Rightarrow$ *'a set* **assumes** E: $\bigwedge i.\ E\ i \in sets\ M$
shows *Pi UNIV E* $\in$ *sets S*
proof −
have *Pi UNIV E* = $(\bigcap i.\ emb\ UNIV\ \{..i\}\ (\Pi_E\ j \in \{..i\}.\ E\ j))$
using *E E*[*THEN sets.sets-into-space*]
by (*auto simp*: *prod-emb-def Pi-iff extensional-def*)
with *E* **show** *?thesis* **by** *auto*
qed

lemma *measure-PiM-countable*:
fixes E :: *nat* $\Rightarrow$ *'a set* **assumes** E: $\bigwedge i.\ E\ i \in sets\ M$
shows $(\lambda n.\ \prod i \leq n.\ measure\ M\ (E\ i)) \longrightarrow measure\ S\ (Pi\ UNIV\ E)$
proof −
let *?E* = $\lambda n.\ emb\ UNIV\ \{..n\}\ (Pi_E\ \{..\ n\}\ E)$
have $\bigwedge n.\ (\prod i \leq n.\ measure\ M\ (E\ i)) = measure\ S\ (?E\ n)$
using *E* **by** (*simp add*: *measure-PiM-emb*)
moreover have *Pi UNIV E* = $(\bigcap n.\ ?E\ n)$
using *E E*[*THEN sets.sets-into-space*]
by (*auto simp*: *prod-emb-def extensional-def Pi-iff*)
moreover have *range ?E* $\subseteq$ *sets S*
using *E* **by** *auto*
moreover have *decseq ?E*
by (*auto simp*: *prod-emb-def Pi-iff decseq-def*)
ultimately show *?thesis*
by (*simp add*: *finite-Lim-measure-decseq*)
qed

lemma *nat-eq-diff-eq*:
fixes $a\ b\ c$ :: *nat*

shows $c \leq b \Longrightarrow a = b - c \longleftrightarrow a + c = b$
by *auto*

lemma *PiM-comb-seq*:
$distr\ (S \bigotimes_M S)\ S\ (\lambda(\omega, \omega').\ comb\text{-}seq\ i\ \omega\ \omega') = S$ (**is** *?D* = -)
proof (*rule PiM-eq*)
let *?I* = *UNIV*::*nat set* **and** *?M* = $\lambda n.\ M$
let *distr - - ?f* = *?D*

fix *J E* **assume** *J*: *finite J J* $\subseteq$ *?I* $\bigwedge j.\ j \in J \Longrightarrow E\ j \in sets\ M$
let *?X* = *prod-emb ?I ?M J* $(\Pi_E\ j \in J.\ E\ j)$
have $\bigwedge j\ x.\ j \in J \Longrightarrow x \in E\ j \Longrightarrow x \in space\ M$
using *J*(*3*)[*THEN sets.sets-into-space*] **by** (*auto simp*: *space-PiM Pi-iff subset-eq*)
with *J* **have** *?f* $-`$ *?X* $\cap$ *space* $(S \bigotimes_M S) =$
(*prod-emb ?I ?M* $(J \cap \{..<i\})$ $(\Pi_E\ j \in J \cap \{..<i\}.\ E\ j)) \times$
(*prod-emb ?I ?M* $(((+)\ i) -` J)$ $(\Pi_E\ j \in ((+)\ i) -` J.\ E\ (i + j)))$ (**is** - = *?E* $\times$ *?F*)
by (*auto simp*: *space-pair-measure space-PiM prod-emb-def all-conj-distrib PiE-iff split*: *split-comb-seq split-comb-seq-asm*)
then have *emeasure ?D ?X* = *emeasure* $(S \bigotimes_M S)$ (*?E* $\times$ *?F*)
by (*subst emeasure-distr*[*OF measurable-comb-seq*])
(*auto intro*!: *sets-PiM-I simp*: *split-beta' J*)
also have ... = *emeasure S ?E* $*$ *emeasure S ?F*
using *J* **by** (*intro P.emeasure-pair-measure-Times*) (*auto intro*!: *sets-PiM-I finite-vimageI simp*: *inj-on-def*)
also have *emeasure S ?F* = $(\prod j \in ((+)\ i) -` J.\ emeasure\ M\ (E\ (i + j)))$
using *J* **by** (*intro emeasure-PiM-emb*) (*simp-all add*: *finite-vimageI inj-on-def*)
also have ... = $(\prod j \in J - (J \cap \{..<i\}).\ emeasure\ M\ (E\ j))$
by (*rule prod.reindex-cong* [*of* $\lambda x.\ x - i$])
(*auto simp*: *image-iff ac-simps nat-eq-diff-eq cong*: *conj-cong intro*!: *inj-onI*)
also have *emeasure S ?E* = $(\prod j \in J \cap \{..<i\}.\ emeasure\ M\ (E\ j))$
using *J* **by** (*intro emeasure-PiM-emb*) *simp-all*
also have $(\prod j \in J \cap \{..<i\}.\ emeasure\ M\ (E\ j)) * (\prod j \in J - (J \cap \{..<i\}).\ emeasure\ M\ (E\ j)) = (\prod j \in J.\ emeasure\ M\ (E\ j))$
by (*subst mult.commute*) (*auto simp*: *J prod.subset-diff*[*symmetric*])
finally show *emeasure ?D ?X* = $(\prod j \in J.\ emeasure\ M\ (E\ j))$.
qed *simp-all*

lemma *PiM-iter*:
$distr\ (M \bigotimes_M S)\ S\ (\lambda(s, \omega).\ case\text{-}nat\ s\ \omega) = S$ (**is** *?D* = -)
proof (*rule PiM-eq*)
let *?I* = *UNIV*::*nat set* **and** *?M* = $\lambda n.\ M$
let *distr - - ?f* = *?D*

fix *J E* **assume** *J*: *finite J J* $\subseteq$ *?I* $\bigwedge j.\ j \in J \Longrightarrow E\ j \in sets\ M$
let *?X* = *prod-emb ?I ?M J* $(\Pi_E\ j \in J.\ E\ j)$
have $\bigwedge j\ x.\ j \in J \Longrightarrow x \in E\ j \Longrightarrow x \in space\ M$
using *J*(*3*)[*THEN sets.sets-into-space*] **by** (*auto simp*: *space-PiM Pi-iff sub-*

set-eq)
with *J* **have** *?f* $-$‘ *?X* $\cap$ *space* (*M* $\bigotimes_M$ *S*) = (*if 0* $\in$ *J then E 0 else space M*) $\times$
(*prod-emb ?I ?M* (*Suc* $-$‘ *J*) (Π_E *j*$\in$*Suc* $-$‘ *J*. *E* (*Suc j*))) (**is** - = *?E* $\times$ *?F*)
by (*auto simp*: *space-pair-measure space-PiM PiE-iff prod-emb-def all-conj-distrib split*: *nat.split nat.split-asm*)
then have *emeasure ?D ?X* = *emeasure* (*M* $\bigotimes_M$ *S*) (*?E* $\times$ *?F*)
by (*subst emeasure-distr*)
(*auto intro*!: *sets-PiM-I simp*: *split-beta*′ *J*)
also have ... = *emeasure M ?E* $*$ *emeasure S ?F*
using *J* **by** (*intro P.emeasure-pair-measure-Times*) (*auto intro*!: *sets-PiM-I finite-vimageI*)
also have *emeasure S ?F* = ($\prod$*j*$\in$*Suc* $-$‘ *J*. *emeasure M* (*E* (*Suc j*)))
using *J* **by** (*intro emeasure-PiM-emb*) (*simp-all add*: *finite-vimageI*)
also have ... = ($\prod$*j*$\in$*J* $-$ {*0*}. *emeasure M* (*E j*))
by (*rule prod.reindex-cong* [*of* λ*x*. *x* $-$ *1*])
(*auto simp*: *image-iff nat-eq-diff-eq ac-simps cong*: *conj-cong intro*!: *inj-onI*)
also have *emeasure M ?E* $*$ ($\prod$*j*$\in$*J* $-$ {*0*}. *emeasure M* (*E j*)) = ($\prod$*j*$\in$*J*. *emeasure M* (*E j*))
by (*auto simp*: *M.emeasure-space-1 prod.remove J*)
finally show *emeasure ?D ?X* = ($\prod$*j*$\in$*J*. *emeasure M* (*E j*)) **.**
qed *simp-all*

end

lemma *PiM-return*:
assumes *finite I*
assumes [*measurable*]: $\bigwedge$*i*. *i* $\in$ *I* $\Longrightarrow$ {*a i*} $\in$ *sets* (*M i*)
shows *PiM I* (λ*i*. *return* (*M i*) (*a i*)) = *return* (*PiM I M*) (*restrict a I*)
proof $-$
have [*simp*]: *a i* $\in$ *space* (*M i*) **if** *i* $\in$ *I* **for** *i*
using *assms*(*2*)[*OF that*] **by** (*meson insert-subset sets.sets-into-space*)
interpret *prob-space PiM I* (λ*i*. *return* (*M i*) (*a i*))
by (*intro prob-space-PiM prob-space-return*) *auto*
have *AE x in PiM I* (λ*i*. *return* (*M i*) (*a i*)). $\forall$ *i*$\in$*I*. *x i* = *restrict a I i*
by (*intro eventually-ball-finite ballI AE-PiM-component prob-space-return assms*)
(*auto simp*: *AE-return*)
moreover have *AE x in PiM I* (λ*i*. *return* (*M i*) (*a i*)). *x* $\in$ *space* (*PiM I* (λ*i*. *return* (*M i*) (*a i*)))
by *simp*
ultimately have *AE x in PiM I* (λ*i*. *return* (*M i*) (*a i*)). *x* = *restrict a I*
by *eventually-elim* (*auto simp*: *fun-eq-iff space-PiM*)
hence Pi_M *I* (λ*i*. *return* (*M i*) (*a i*)) = *return* (Pi_M *I* (λ*i*. *return* (*M i*) (*a i*))) (*restrict a I*)
by (*rule AE-eq-constD*)
also have ... = *return* (*PiM I M*) (*restrict a I*)
by (*intro return-cong sets-PiM-cong*) *auto*
finally show *?thesis* **.**
qed

lemma *distr-PiM-finite-prob-space'*:
assumes *fin*: *finite I*
assumes $\bigwedge i.\ i \in I \Longrightarrow$ *prob-space* (*M i*)
assumes $\bigwedge i.\ i \in I \Longrightarrow$ *prob-space* (*M' i*)
assumes [*measurable*]: $\bigwedge i.\ i \in I \Longrightarrow f \in$ *measurable* (*M i*) (*M' i*)
shows *distr* (*PiM I M*) (*PiM I M'*) (*compose I f*) = *PiM I* (λ*i*. *distr* (*M i*) (*M' i*) *f*)
proof −
define *N* **where** *N* = (λ*i*. *if i* ∈ *I then M i else return* (*count-space UNIV*) *undefined*)
define *N'* **where** *N'* = (λ*i*. *if i* ∈ *I then M' i else return* (*count-space UNIV*) *undefined*)
have [*simp*]: *PiM I N* = *PiM I M PiM I N'* = *PiM I M'*
by (*intro PiM-cong*; *simp add*: *N-def N'-def*)+

have *distr* (*PiM I N*) (*PiM I N'*) (*compose I f*) = *PiM I* (λ*i*. *distr* (*N i*) (*N' i*) *f*)
proof (*rule distr-PiM-finite-prob-space*)
show *product-prob-space N*
by (*rule product-prob-spaceI*) (*auto simp*: *N-def intro*!: *prob-space-return assms*)
show *product-prob-space N'*
by (*rule product-prob-spaceI*) (*auto simp*: *N'-def intro*!: *prob-space-return assms*)
qed (*auto simp*: *N-def N'-def fin*)
also have Pi_M *I* (λ*i*. *distr* (*N i*) (*N' i*) *f*) = Pi_M *I* (λ*i*. *distr* (*M i*) (*M' i*) *f*)
by (*intro PiM-cong*) (*simp-all add*: *N-def N'-def*)
finally show *?thesis* **by** *simp*
qed

end

10 Independent families of events, event sets, and random variables

theory *Independent-Family*
imports *Infinite-Product-Measure*
begin

definition (**in** *prob-space*)
indep-sets F I ⟷ (∀ *i*∈*I*. *F i* ⊆ *events*) ∧
(∀ *J*⊆*I*. *J* ≠ {} ⟶ *finite J* ⟶ (∀ *A*∈*Pi J F*. *prob* (⋂*j*∈*J*. *A j*) = (∏*j*∈*J*. *prob* (*A j*))))

definition (**in** *prob-space*)
indep-set A B ⟷ *indep-sets* (*case-bool A B*) *UNIV*

definition (**in** *prob-space*)

indep-events-def-alt: *indep-events A I* $\longleftrightarrow$ *indep-sets* ($\lambda i.\ \{A\ i\}$) *I*

lemma (**in** *prob-space*) *indep-events-def*:
indep-events A I $\longleftrightarrow$ (*A*'*I* $\subseteq$ *events*) $\wedge$
($\forall J{\subseteq}I.\ J \neq \{\} \longrightarrow$ *finite J* $\longrightarrow$ *prob* ($\bigcap j{\in}J.\ A\ j$) = ($\prod j{\in}J.$ *prob* (*A j*)))
unfolding *indep-events-def-alt indep-sets-def*
apply (*simp add*: *Ball-def Pi-iff image-subset-iff-funcset*)
apply (*intro conj-cong refl arg-cong*[**where** *f*=*All*] *ext imp-cong*)
apply *auto*
done

lemma (**in** *prob-space*) *indep-eventsI*:
($\bigwedge i.\ i \in I \Longrightarrow F\ i \in$ *sets M*) $\Longrightarrow$ ($\bigwedge J.\ J \subseteq I \Longrightarrow$ *finite J* $\Longrightarrow$ *J* $\neq \{\} \Longrightarrow$ *prob* ($\bigcap i{\in}J.\ F\ i$) = ($\prod i{\in}J.$ *prob* (*F i*))) $\Longrightarrow$ *indep-events F I*
by (*auto simp*: *indep-events-def*)

definition (**in** *prob-space*)
indep-event A B $\longleftrightarrow$ *indep-events* (*case-bool A B*) *UNIV*

lemma (**in** *prob-space*) *indep-sets-cong*:
I = *J* $\Longrightarrow$ ($\bigwedge i.\ i \in I \Longrightarrow F\ i = G\ i$) $\Longrightarrow$ *indep-sets F I* $\longleftrightarrow$ *indep-sets G J*
by (*simp add*: *indep-sets-def*, *intro conj-cong all-cong imp-cong ball-cong*) *blast*+

lemma (**in** *prob-space*) *indep-events-finite-index-events*:
indep-events F I $\longleftrightarrow$ ($\forall J{\subseteq}I.\ J \neq \{\} \longrightarrow$ *finite J* $\longrightarrow$ *indep-events F J*)
by (*auto simp*: *indep-events-def*)

lemma (**in** *prob-space*) *indep-sets-finite-index-sets*:
indep-sets F I $\longleftrightarrow$ ($\forall J{\subseteq}I.\ J \neq \{\} \longrightarrow$ *finite J* $\longrightarrow$ *indep-sets F J*)
proof (*intro iffI allI impI*)
assume $*$: $\forall J{\subseteq}I.\ J \neq \{\} \longrightarrow$ *finite J* $\longrightarrow$ *indep-sets F J*
show *indep-sets F I* **unfolding** *indep-sets-def*
proof (*intro conjI ballI allI impI*)
fix *i* **assume** *i* $\in$ *I*
with $*$[*THEN spec*, *of* {*i*}] **show** *F i* $\subseteq$ *events*
by (*auto simp*: *indep-sets-def*)
qed (*insert* $*$, *auto simp*: *indep-sets-def*)
qed (*auto simp*: *indep-sets-def*)

lemma (**in** *prob-space*) *indep-sets-mono-index*:
J $\subseteq$ *I* $\Longrightarrow$ *indep-sets F I* $\Longrightarrow$ *indep-sets F J*
unfolding *indep-sets-def* **by** *auto*

lemma (**in** *prob-space*) *indep-sets-mono-sets*:
assumes *indep*: *indep-sets F I*
assumes *mono*: $\bigwedge i.\ i{\in}I \Longrightarrow G\ i \subseteq F\ i$
shows *indep-sets G I*
proof −
have ($\forall i{\in}I.\ F\ i \subseteq$ *events*) $\Longrightarrow$ ($\forall i{\in}I.\ G\ i \subseteq$ *events*)

using *mono* **by** *auto*
moreover have $\bigwedge A\ J.\ J \subseteq I \Longrightarrow A \in (\Pi\ j{\in}J.\ G\ j) \Longrightarrow A \in (\Pi\ j{\in}J.\ F\ j)$
using *mono* **by** (*auto simp*: *Pi-iff*)
ultimately show *?thesis*
using *indep* **by** (*auto simp*: *indep-sets-def*)
qed

lemma (**in** *prob-space*) *indep-sets-mono*:
assumes *indep*: *indep-sets F I*
assumes *mono*: $J \subseteq I\ \bigwedge i.\ i{\in}J \Longrightarrow G\ i \subseteq F\ i$
shows *indep-sets G J*
apply (*rule indep-sets-mono-sets*)
apply (*rule indep-sets-mono-index*)
apply (*fact +*)
done

lemma (**in** *prob-space*) *indep-setsI*:
assumes $\bigwedge i.\ i \in I \Longrightarrow F\ i \subseteq events$
and $\bigwedge A\ J.\ J \neq \{\} \Longrightarrow J \subseteq I \Longrightarrow$ *finite* $J \Longrightarrow (\forall j{\in}J.\ A\ j \in F\ j) \Longrightarrow$ *prob* $(\bigcap j{\in}J.\ A\ j) = (\prod j{\in}J.\ prob\ (A\ j))$
shows *indep-sets F I*
using *assms* **unfolding** *indep-sets-def* **by** (*auto simp*: *Pi-iff*)

lemma (**in** *prob-space*) *indep-setsD*:
assumes *indep-sets F I* **and** $J \subseteq I\ J \neq \{\}$ *finite* $J\ \forall j{\in}J.\ A\ j \in F\ j$
shows *prob* $(\bigcap j{\in}J.\ A\ j) = (\prod j{\in}J.\ prob\ (A\ j))$
using *assms* **unfolding** *indep-sets-def* **by** *auto*

lemma (**in** *prob-space*) *indep-setI*:
assumes *ev*: $A \subseteq events\ B \subseteq events$
and *indep*: $\bigwedge a\ b.\ a \in A \Longrightarrow b \in B \Longrightarrow prob\ (a \cap b) = prob\ a * prob\ b$
shows *indep-set A B*
unfolding *indep-set-def*
proof (*rule indep-setsI*)
fix *F J* **assume** $J \neq \{\}\ J \subseteq UNIV$
and *F*: $\forall j{\in}J.\ F\ j \in (case\ j\ of\ True \Rightarrow A \mid False \Rightarrow B)$
have $J \in Pow\ UNIV$ **by** *auto*
with *F* ‹$J \neq \{\}$› *indep*[*of F True F False*]
show *prob* $(\bigcap j{\in}J.\ F\ j) = (\prod j{\in}J.\ prob\ (F\ j))$
unfolding *UNIV-bool Pow-insert* **by** (*auto simp*: *ac-simps*)
qed (*auto split*: *bool.split simp*: *ev*)

lemma (**in** *prob-space*) *indep-setD*:
assumes *indep*: *indep-set A B* **and** *ev*: $a \in A\ b \in B$
shows *prob* $(a \cap b) = prob\ a * prob\ b$
using *indep*[*unfolded indep-set-def*, *THEN indep-setsD*, *of UNIV case-bool a b*] *ev*
by (*simp add*: *ac-simps UNIV-bool*)

lemma (**in** *prob-space*)
assumes *indep*: *indep-set A B*
shows *indep-setD-ev1*: $A \subseteq$ *events*
and *indep-setD-ev2*: $B \subseteq$ *events*
using *indep* **unfolding** *indep-set-def indep-sets-def UNIV-bool* **by** *auto*

lemma (**in** *prob-space*) *indep-sets-Dynkin*:
assumes *indep*: *indep-sets F I*
shows *indep-sets* (λi. *Dynkin* (*space M*) (*F i*)) *I*
(**is** *indep-sets ?F I*)
proof (*subst indep-sets-finite-index-sets*, *intro allI impI ballI*)
fix *J* **assume** *finite J J* $\subseteq$ *I J* $\neq \{\}$
with *indep* **have** *indep-sets F J*
by (*subst* (*asm*) *indep-sets-finite-index-sets*) *auto*
{ fix *J K* **assume** *indep-sets F K*
let *?G* = $\lambda S\ i$. *if i* $\in$ *S then ?F i else F i*
assume *finite J J* $\subseteq$ *K*
then have *indep-sets* (*?G J*) *K*
proof *induct*
case (*insert j J*)
moreover define *G* **where** *G* = *?G J*
ultimately have *G*: *indep-sets G K* $\bigwedge i.\ i \in K \Longrightarrow G\ i \subseteq$ *events* **and** $j \in K$
by (*auto simp*: *indep-sets-def*)
let *?D* = {*E*$\in$*events*. *indep-sets* (*G*(*j* := {*E*})) *K* }
{ fix *X* **assume** *X*: *X* $\in$ *events*
assume *indep*: $\bigwedge J\ A.\ J \neq \{\} \Longrightarrow J \subseteq K \Longrightarrow$ *finite J* $\Longrightarrow j \notin J \Longrightarrow (\forall i \in J.$ *A i* $\in$ *G i*)
$\Longrightarrow$ *prob* (($\bigcap i \in J.$ *A i*) $\cap$ *X*) = *prob X* $*$ ($\prod i \in J.$ *prob* (*A i*))
have *indep-sets* (*G*(*j* := {*X*})) *K*
proof (*rule indep-setsI*)
fix *i* **assume** *i* $\in$ *K* **then show** (*G*(*j*:={*X*})) *i* $\subseteq$ *events*
using *G X* **by** *auto*
next
fix *A J* **assume** *J*: *J* $\neq \{\}$ *J* $\subseteq$ *K finite J* $\forall i \in J.$ *A i* $\in$ (*G*(*j* := {*X*})) *i*
show *prob* ($\bigcap j \in J.$ *A j*) = ($\prod j \in J.$ *prob* (*A j*))
proof *cases*
assume *j* $\in$ *J*
with *J* **have** *A j* = *X* **by** *auto*
show *?thesis*
proof *cases*
assume *J* = {*j*} **then show** *?thesis* **by** *simp*
next
assume *J* $\neq$ {*j*}
have *prob* ($\bigcap i \in J.$ *A i*) = *prob* (($\bigcap i \in J - \{j\}.$ *A i*) $\cap$ *X*)
using ‹*j* $\in$ *J*› ‹*A j* = *X*› **by** (*auto intro*!: *arg-cong*[**where** *f*=*prob*] *split*: *if-split-asm*)
also have ... = *prob X* $*$ ($\prod i \in J - \{j\}.$ *prob* (*A i*))
proof (*rule indep*)
show *J* $-$ {*j*} $\neq \{\}$ *J* $-$ {*j*} $\subseteq$ *K finite* (*J* $-$ {*j*}) *j* $\notin$ *J* $-$ {*j*}

```
            using J ‹J ≠ {j}› ‹j ∈ J› by auto
          show ∀ i∈J − {j}. A i ∈ G i
            using J by auto
        qed
        also have ... = prob (A j) * (∏ i∈J−{j}. prob (A i))
          using ‹A j = X› by simp
        also have ... = (∏ i∈J. prob (A i))
          unfolding prod.insert-remove[OF ‹finite J›, symmetric, of λi. prob
(A i)]
          using ‹j ∈ J› by (simp add: insert-absorb)
        finally show ?thesis .
      qed
    next
      assume j ∉ J
      with J have ∀ i∈J. A i ∈ G i by (auto split: if-split-asm)
      with J show ?thesis
        by (intro indep-setsD[OF G(1)]) auto
    qed
  qed }
  note indep-sets-insert = this
  have Dynkin-system (space M) ?D
  proof (rule Dynkin-systemI', simp-all cong del: indep-sets-cong, safe)
    show indep-sets (G(j := {{}})) K
      by (rule indep-sets-insert) auto
  next
    fix X assume X: X ∈ events and G': indep-sets (G(j := {X})) K
    show indep-sets (G(j := {space M − X})) K
    proof (rule indep-sets-insert)
      fix J A assume J: J ≠ {} J ⊆ K finite J j ∉ J and A: ∀ i∈J. A i ∈ G i
      then have A-sets: ⋀i. i∈J ⟹ A i ∈ events
        using G by auto
      have prob ((⋂ j∈J. A j) ∩ (space M − X)) =
          prob ((⋂ j∈J. A j) − (⋂ i∈insert j J. (A(j := X)) i))
        using A-sets sets.sets-into-space[of - M] X ‹J ≠ {}›
        by (auto intro!: arg-cong[where f=prob] split: if-split-asm)
      also have ... = prob (⋂ j∈J. A j) − prob (⋂ i∈insert j J. (A(j := X)) i)
        using J ‹J ≠ {}› ‹j ∉ J› A-sets X sets.sets-into-space
        by (auto intro!: finite-measure-Diff sets.finite-INT split: if-split-asm)
      finally have prob ((⋂ j∈J. A j) ∩ (space M − X)) =
          prob (⋂ j∈J. A j) − prob (⋂ i∈insert j J. (A(j := X)) i) .
      moreover {
        have prob (⋂ j∈J. A j) = (∏ j∈J. prob (A j))
          using J A ‹finite J› by (intro indep-setsD[OF G(1)]) auto
        then have prob (⋂ j∈J. A j) = prob (space M) * (∏ i∈J. prob (A i))
          using prob-space by simp }
      moreover {
        have prob (⋂ i∈insert j J. (A(j := X)) i) = (∏ i∈insert j J. prob ((A(j
:= X)) i))
          using J A ‹j ∈ K› by (intro indep-setsD[OF G'] ) auto
```

then have *prob* $(\bigcap i{\in}insert\ j\ J.\ (A(j := X))\ i) = prob\ X * (\prod i{\in}J.\ prob\ (A\ i))$
using ‹*finite J*› ‹$j \notin J$› **by** (*auto intro*!: *prod.cong*) **}**
ultimately have *prob* $((\bigcap j{\in}J.\ A\ j) \cap (space\ M - X)) = (prob\ (space\ M) - prob\ X) * (\prod i{\in}J.\ prob\ (A\ i))$
by (*simp add*: *field-simps*)
also have $\ldots = prob\ (space\ M - X) * (\prod i{\in}J.\ prob\ (A\ i))$
using *X A* **by** (*simp add*: *finite-measure-compl*)
finally show *prob* $((\bigcap j{\in}J.\ A\ j) \cap (space\ M - X)) = prob\ (space\ M - X) * (\prod i{\in}J.\ prob\ (A\ i))$ **.**
qed (*insert X*, *auto*)
next
fix *F* :: *nat* ⇒ *'a set* **assume** *disj*: *disjoint-family F* **and** *range F* ⊆ *?D*
then have *F*: $\bigwedge i.\ F\ i \in events$ $\bigwedge i.\ indep\text{-}sets\ (G(j{:=}\{F\ i\}))\ K$ **by** *auto*
show *indep-sets* $(G(j := \{\bigcup k.\ F\ k\}))\ K$
proof (*rule indep-sets-insert*)
fix *J A* **assume** *J*: $j \notin J\ J \neq \{\}\ J \subseteq K$ *finite J* **and** *A*: $\forall i{\in}J.\ A\ i \in G\ i$
then have *A-sets*: $\bigwedge i.\ i{\in}J \Longrightarrow A\ i \in events$
using *G* **by** *auto*
have *prob* $((\bigcap j{\in}J.\ A\ j) \cap (\bigcup k.\ F\ k)) = prob\ (\bigcup k.\ (\bigcap i{\in}insert\ j\ J.\ (A(j := F\ k))\ i))$
using ‹$J \neq \{\}$› ‹$j \notin J$› ‹$j \in K$› **by** (*auto intro*!: *arg-cong*[**where** *f*=*prob*] *split*: *if-split-asm*)
moreover have $(\lambda k.\ prob\ (\bigcap i{\in}insert\ j\ J.\ (A(j := F\ k))\ i))\ sums\ prob\ (\bigcup k.\ (\bigcap i{\in}insert\ j\ J.\ (A(j := F\ k))\ i))$
proof (*rule finite-measure-UNION*)
show *disjoint-family* $(\lambda k.\ \bigcap i{\in}insert\ j\ J.\ (A(j := F\ k))\ i)$
using *disj* **by** (*rule disjoint-family-on-bisimulation*) *auto*
show *range* $(\lambda k.\ \bigcap i{\in}insert\ j\ J.\ (A(j := F\ k))\ i) \subseteq events$
using *A-sets F* ‹*finite J*› ‹$J \neq \{\}$› ‹$j \notin J$› **by** (*auto intro*!: *sets.Int*)
qed
moreover { fix *k*
from *J A* ‹$j \in K$› **have** *prob* $(\bigcap i{\in}insert\ j\ J.\ (A(j := F\ k))\ i) = prob\ (F\ k) * (\prod i{\in}J.\ prob\ (A\ i))$
by (*subst indep-setsD*[*OF F*(*2*)]) (*auto intro*!: *prod.cong split*: *if-split-asm*)
also have $\ldots = prob\ (F\ k) * prob\ (\bigcap i{\in}J.\ A\ i)$
using *J A* ‹$j \in K$› **by** (*subst indep-setsD*[*OF G*(*1*)]) *auto*
finally have *prob* $(\bigcap i{\in}insert\ j\ J.\ (A(j := F\ k))\ i) = prob\ (F\ k) * prob\ (\bigcap i{\in}J.\ A\ i)$ **. }**
ultimately have $(\lambda k.\ prob\ (F\ k) * prob\ (\bigcap i{\in}J.\ A\ i))\ sums\ (prob\ ((\bigcap j{\in}J.\ A\ j) \cap (\bigcup k.\ F\ k)))$
by *simp*
moreover
have $(\lambda k.\ prob\ (F\ k) * prob\ (\bigcap i{\in}J.\ A\ i))\ sums\ (prob\ (\bigcup k.\ F\ k) * prob\ (\bigcap i{\in}J.\ A\ i))$
using *disj F*(*1*) **by** (*intro finite-measure-UNION sums-mult2*) *auto*
then have $(\lambda k.\ prob\ (F\ k) * prob\ (\bigcap i{\in}J.\ A\ i))\ sums\ (prob\ (\bigcup k.\ F\ k) * (\prod i{\in}J.\ prob\ (A\ i)))$
using *J A* ‹$j \in K$› **by** (*subst indep-setsD*[*OF G*(*1*), *symmetric*]) *auto*

```
        ultimately
        show prob ((⋂j∈J. A j) ∩ (⋃k. F k)) = prob (⋃k. F k) * (∏j∈J. prob
(A j))
          by (auto dest!: sums-unique)
      qed (insert F, auto)
    qed (insert sets.sets-into-space, auto)
     then have mono: Dynkin (space M) (G j) ⊆ {E ∈ events. indep-sets (G(j
:= {E})) K}
    proof (rule Dynkin-system.Dynkin-subset, safe)
      fix X assume X ∈ G j
      then show X ∈ events using G ‹j ∈ K› by auto
      from ‹indep-sets G K›
      show indep-sets (G(j := {X})) K
        by (rule indep-sets-mono-sets) (insert ‹X ∈ G j›, auto)
    qed
    have indep-sets (G(j:=?D)) K
    proof (rule indep-setsI)
      fix i assume i ∈ K then show (G(j := ?D)) i ⊆ events
        using G(2) by auto
    next
     fix A J assume J: J≠{} J ⊆ K finite J and A: ∀i∈J. A i ∈ (G(j := ?D))
i
      show prob (⋂j∈J. A j) = (∏j∈J. prob (A j))
      proof cases
        assume j ∈ J
        with A have indep: indep-sets (G(j := {A j})) K by auto
        from J A show ?thesis
          by (intro indep-setsD[OF indep]) auto
      next
        assume j ∉ J
        with J A have ∀i∈J. A i ∈ G i by (auto split: if-split-asm)
        with J show ?thesis
          by (intro indep-setsD[OF G(1)]) auto
      qed
    qed
    then have indep-sets (G(j := Dynkin (space M) (G j))) K
      by (rule indep-sets-mono-sets) (insert mono, auto)
    then show ?case
      by (rule indep-sets-mono-sets) (insert ‹j ∈ K› ‹j ∉ J›, auto simp: G-def)
  qed (insert ‹indep-sets F K›, simp) }
 from this[OF ‹indep-sets F J› ‹finite J› subset-refl]
 show indep-sets ?F J
  by (rule indep-sets-mono-sets) auto
qed

lemma (in prob-space) indep-sets-sigma:
 assumes indep: indep-sets F I
 assumes stable: ⋀i. i ∈ I ⟹ Int-stable (F i)
 shows indep-sets (λi. sigma-sets (space M) (F i)) I
```

proof −
from *indep-sets-Dynkin*[*OF indep*]
show *?thesis*
proof (*rule indep-sets-mono-sets*, *subst sigma-eq-Dynkin*, *simp-all add*: *stable*)
fix *i* **assume** $i \in I$
with *indep* **have** $F\ i \subseteq$ *events* **by** (*auto simp*: *indep-sets-def*)
with *sets.sets-into-space* **show** $F\ i \subseteq Pow\ (space\ M)$ **by** *auto*
qed
qed

lemma (**in** *prob-space*) *indep-sets-sigma-sets-iff*:
assumes $\bigwedge i.\ i \in I \Longrightarrow$ *Int-stable* $(F\ i)$
shows *indep-sets* $(\lambda i.\ sigma\text{-}sets\ (space\ M)\ (F\ i))\ I \longleftrightarrow$ *indep-sets F I*
proof
assume *indep-sets F I* **then show** *indep-sets* $(\lambda i.\ sigma\text{-}sets\ (space\ M)\ (F\ i))\ I$
by (*rule indep-sets-sigma*) *fact*
next
assume *indep-sets* $(\lambda i.\ sigma\text{-}sets\ (space\ M)\ (F\ i))\ I$ **then show** *indep-sets F I*
by (*rule indep-sets-mono-sets*) (*intro subsetI sigma-sets.Basic*)
qed

definition (**in** *prob-space*)
indep-vars-def2: *indep-vars* $M'\ X\ I \longleftrightarrow$
$(\forall i \in I.\ random\text{-}variable\ (M'\ i)\ (X\ i)) \wedge$
indep-sets $(\lambda i.\ \{\ X\ i -` A \cap space\ M \mid A.\ A \in sets\ (M'\ i)\})\ I$

definition (**in** *prob-space*)
indep-var Ma A Mb B $\longleftrightarrow$ *indep-vars* (*case-bool Ma Mb*) (*case-bool A B*) *UNIV*

lemma (**in** *prob-space*) *indep-vars-def*:
indep-vars $M'\ X\ I \longleftrightarrow$
$(\forall i \in I.\ random\text{-}variable\ (M'\ i)\ (X\ i)) \wedge$
indep-sets $(\lambda i.\ sigma\text{-}sets\ (space\ M)\ \{\ X\ i -` A \cap space\ M \mid A.\ A \in sets\ (M'\ i)\})\ I$
unfolding *indep-vars-def2*
apply (*rule conj-cong*[*OF refl*])
apply (*rule indep-sets-sigma-sets-iff*[*symmetric*])
apply (*auto simp*: *Int-stable-def*)
apply (*rule-tac x*=$A \cap Aa$ **in** *exI*)
apply *auto*
done

lemma (**in** *prob-space*) *indep-var-eq*:
indep-var S X T Y $\longleftrightarrow$
$(random\text{-}variable\ S\ X \wedge random\text{-}variable\ T\ Y) \wedge$
indep-set
$(sigma\text{-}sets\ (space\ M)\ \{\ X -` A \cap space\ M \mid A.\ A \in sets\ S\})$
$(sigma\text{-}sets\ (space\ M)\ \{\ Y -` A \cap space\ M \mid A.\ A \in sets\ T\})$
unfolding *indep-var-def indep-vars-def indep-set-def UNIV-bool*

by (*intro arg-cong2*[**where** *f*=(∧)] *arg-cong2*[**where** *f*=*indep-sets*] *ext*)
(*auto split*: *bool.split*)

lemma (**in** *prob-space*) *indep-sets2-eq*:
indep-set A B ⟷ *A* ⊆ *events* ∧ *B* ⊆ *events* ∧ (∀ *a*∈*A*. ∀ *b*∈*B*. *prob* (*a* ∩ *b*) = *prob a* ∗ *prob b*)
unfolding *indep-set-def*
proof (*intro iffI ballI conjI*)
assume *indep*: *indep-sets* (*case-bool A B*) *UNIV*
{ fix *a b* **assume** *a* ∈ *A b* ∈ *B*
with *indep-setsD*[*OF indep*, *of UNIV case-bool a b*]
show *prob* (*a* ∩ *b*) = *prob a* ∗ *prob b*
unfolding *UNIV-bool* **by** (*simp add*: *ac-simps*) **}**
from *indep* **show** *A* ⊆ *events B* ⊆ *events*
unfolding *indep-sets-def UNIV-bool* **by** *auto*
next
assume ∗: *A* ⊆ *events* ∧ *B* ⊆ *events* ∧ (∀ *a*∈*A*. ∀ *b*∈*B*. *prob* (*a* ∩ *b*) = *prob a* ∗ *prob b*)
show *indep-sets* (*case-bool A B*) *UNIV*
proof (*rule indep-setsI*)
fix *i* **show** (*case i of True* ⇒ *A* | *False* ⇒ *B*) ⊆ *events*
using ∗ **by** (*auto split*: *bool.split*)
next
fix *J X* **assume** *J* ≠ {} *J* ⊆ *UNIV* **and** *X*: ∀ *j*∈*J*. *X j* ∈ (*case j of True* ⇒ *A* | *False* ⇒ *B*)
then have *J* = {*True*} ∨ *J* = {*False*} ∨ *J* = {*True*,*False*}
by (*auto simp*: *UNIV-bool*)
then show *prob* (⋂ *j*∈*J*. *X j*) = (∏ *j*∈*J*. *prob* (*X j*))
using *X* ∗ **by** *auto*
qed
qed

lemma (**in** *prob-space*) *indep-set-sigma-sets*:
assumes *indep-set A B*
assumes *A*: *Int-stable A* **and** *B*: *Int-stable B*
shows *indep-set* (*sigma-sets* (*space M*) *A*) (*sigma-sets* (*space M*) *B*)
proof −
have *indep-sets* (λ*i*. *sigma-sets* (*space M*) (*case i of True* ⇒ *A* | *False* ⇒ *B*)) *UNIV*
proof (*rule indep-sets-sigma*)
show *indep-sets* (*case-bool A B*) *UNIV*
by (*rule* ‹*indep-set A B*›[*unfolded indep-set-def*])
fix *i* **show** *Int-stable* (*case i of True* ⇒ *A* | *False* ⇒ *B*)
using *A B* **by** (*cases i*) *auto*
qed
then show *?thesis*
unfolding *indep-set-def*
by (*rule indep-sets-mono-sets*) (*auto split*: *bool.split*)
qed

lemma (**in** *prob-space*) *indep-eventsI-indep-vars*:
assumes *indep*: *indep-vars N X I*
assumes *P*: $\bigwedge i.\ i \in I \Longrightarrow \{x \in space\ (N\ i).\ P\ i\ x\} \in sets\ (N\ i)$
shows *indep-events* $(\lambda i.\ \{x \in space\ M.\ P\ i\ (X\ i\ x)\})\ I$
proof −
have *indep-sets* $(\lambda i.\ \{X\ i -` A \cap space\ M\ |A.\ A \in sets\ (N\ i)\})\ I$
using *indep* **unfolding** *indep-vars-def2* **by** *auto*
then show *?thesis*
unfolding *indep-events-def-alt*
proof (*rule indep-sets-mono-sets*)
fix *i* **assume** $i \in I$
then have $\{\{x \in space\ M.\ P\ i\ (X\ i\ x)\}\} = \{X\ i -` \{x \in space\ (N\ i).\ P\ i\ x\} \cap space\ M\}$
using *indep* **by** (*auto simp*: *indep-vars-def dest*: *measurable-space*)
also have $\ldots \subseteq \{X\ i -` A \cap space\ M\ |A.\ A \in sets\ (N\ i)\}$
using *P*[*OF* ‹*i* ∈ *I*›] **by** *blast*
finally show $\{\{x \in space\ M.\ P\ i\ (X\ i\ x)\}\} \subseteq \{X\ i -` A \cap space\ M\ |A.\ A \in sets\ (N\ i)\}$ **.**
qed
qed

lemma (**in** *prob-space*) *indep-sets-collect-sigma*:
fixes $I :: 'j \Rightarrow 'i\ set$ **and** $J :: 'j\ set$ **and** $E :: 'i \Rightarrow 'a\ set\ set$
assumes *indep*: *indep-sets E* $(\bigcup j \in J.\ I\ j)$
assumes *Int-stable*: $\bigwedge i\ j.\ j \in J \Longrightarrow i \in I\ j \Longrightarrow Int\text{-}stable\ (E\ i)$
assumes *disjoint*: *disjoint-family-on I J*
shows *indep-sets* $(\lambda j.\ sigma\text{-}sets\ (space\ M)\ (\bigcup i \in I\ j.\ E\ i))\ J$
proof −
let *?E* = $\lambda j.\ \{\bigcap k \in K.\ E'\ k |\ E'\ K.\ finite\ K \wedge K \neq \{\} \wedge K \subseteq I\ j \wedge (\forall k \in K.\ E'\ k \in E\ k)\ \}$

from *indep* **have** *E*: $\bigwedge j\ i.\ j \in J \Longrightarrow i \in I\ j \Longrightarrow E\ i \subseteq events$
unfolding *indep-sets-def* **by** *auto*
{ fix *j*
let *?S* = *sigma-sets* (*space M*) $(\bigcup i \in I\ j.\ E\ i)$
assume $j \in J$
from *E*[*OF this*] **interpret** *S*: *sigma-algebra space M ?S*
using *sets.sets-into-space*[*of - M*] **by** (*intro sigma-algebra-sigma-sets*) *auto*

have *sigma-sets* (*space M*) $(\bigcup i \in I\ j.\ E\ i) = sigma\text{-}sets\ (space\ M)\ (?E\ j)$
proof (*rule sigma-sets-eqI*)
fix *A* **assume** $A \in (\bigcup i \in I\ j.\ E\ i)$
then obtain *i* **where** $i \in I\ j\ A \in E\ i$ **..**
then show $A \in sigma\text{-}sets\ (space\ M)\ (?E\ j)$
by (*auto intro*!: *sigma-sets.intros*(2−) *exI*[*of - {i}*] *exI*[*of - λi. A*])
next
fix *A* **assume** $A \in ?E\ j$
then obtain $E'\ K$ **where** *finite K* $K \neq \{\}$ $K \subseteq I\ j$ $\bigwedge k.\ k \in K \Longrightarrow E'\ k \in$

```
E k
      and A: A = (⋂k∈K. E' k)
      by auto
    then have A ∈ ?S unfolding A
      by (safe intro!: S.finite-INT) auto
    then show A ∈ sigma-sets (space M) (⋃i∈I j. E i)
      by simp
  qed }
 moreover have indep-sets (λj. sigma-sets (space M) (?E j)) J
 proof (rule indep-sets-sigma)
  show indep-sets ?E J
  proof (intro indep-setsI)
  fix j assume j ∈ J with E show ?E j ⊆ events by (force intro!: sets.finite-INT)
  next
   fix K A assume K: K ≠ {} K ⊆ J finite K
    and ∀j∈K. A j ∈ ?E j
   then have ∀j∈K. ∃E' L. A j = (⋂l∈L. E' l) ∧ finite L ∧ L ≠ {} ∧ L ⊆ I
j ∧ (∀l∈L. E' l ∈ E l)
    by simp
   from bchoice[OF this] obtain E'
    where ∀x∈K. ∃L. A x = ⋂ (E' x ' L) ∧ finite L ∧ L ≠ {} ∧ L ⊆ I x ∧
(∀l∈L. E' x l ∈ E l)
    ..
   from bchoice[OF this] obtain L
    where A: ⋀j. j∈K ⟹ A j = (⋂l∈L j. E' j l)
     and L: ⋀j. j∈K ⟹ finite (L j) ⋀j. j∈K ⟹ L j ≠ {} ⋀j. j∈K ⟹ L j
⊆ I j
    and E': ⋀j l. j∈K ⟹ l ∈ L j ⟹ E' j l ∈ E l
    by auto
   { fix k l j assume k ∈ K j ∈ K l ∈ L j l ∈ L k
    have k = j
    proof (rule ccontr)
     assume k ≠ j
     with disjoint ‹K ⊆ J› ‹k ∈ K› ‹j ∈ K› have I k ∩ I j = {}
      unfolding disjoint-family-on-def by auto
     with L(2,3)[OF ‹j ∈ K›] L(2,3)[OF ‹k ∈ K›]
     show False using ‹l ∈ L k› ‹l ∈ L j› by auto
    qed }
   note L-inj = this

   define k where k l = (SOME k. k ∈ K ∧ l ∈ L k) for l
   { fix x j l assume *: j ∈ K l ∈ L j
    have k l = j unfolding k-def
    proof (rule some-equality)
     fix k assume k ∈ K ∧ l ∈ L k
     with * L-inj show k = j by auto
    qed (insert *, simp) }
   note k-simp[simp] = this
   let ?E' = λl. E' (k l) l
```

have *prob* $(\bigcap j{\in}K.\ A\ j) = prob\ (\bigcap l{\in}(\bigcup k{\in}K.\ L\ k).\ ?E'\ l)$
by (*auto simp*: *A intro*!: *arg-cong*[**where** *f*=*prob*])
also have $\ldots = (\prod l{\in}(\bigcup k{\in}K.\ L\ k).\ prob\ (?E'\ l))$
using *L K E'* **by** (*intro indep-setsD*[*OF indep*]) (*simp-all add*: *UN-mono*)
also have $\ldots = (\prod j{\in}K.\ \prod l{\in}L\ j.\ prob\ (E'\ j\ l))$
using *K L L-inj* **by** (*subst prod.UNION-disjoint*) *auto*
also have $\ldots = (\prod j{\in}K.\ prob\ (A\ j))$
using *K L E'* **by** (*auto simp add*: *A intro*!: *prod.cong indep-setsD*[*OF indep*, *symmetric*]) *blast*
finally show $prob\ (\bigcap j{\in}K.\ A\ j) = (\prod j{\in}K.\ prob\ (A\ j))$ **.**
qed
next
fix *j* **assume** $j \in J$
show *Int-stable* (*?E j*)
proof (*rule Int-stableI*)
fix *a* **assume** $a \in ?E\ j$ **then obtain** *Ka Ea*
where *a*: $a = (\bigcap k{\in}Ka.\ Ea\ k)$ *finite Ka* $Ka \neq \{\}$ $Ka \subseteq I\ j$ $\bigwedge k.\ k{\in}Ka \Longrightarrow Ea\ k \in E\ k$ **by** *auto*
fix *b* **assume** $b \in ?E\ j$ **then obtain** *Kb Eb*
where *b*: $b = (\bigcap k{\in}Kb.\ Eb\ k)$ *finite Kb* $Kb \neq \{\}$ $Kb \subseteq I\ j$ $\bigwedge k.\ k{\in}Kb \Longrightarrow Eb\ k \in E\ k$ **by** *auto*
let $?f = \lambda k.\ (\textit{if } k \in Ka \cap Kb \textit{ then } Ea\ k \cap Eb\ k \textit{ else if } k \in Kb \textit{ then } Eb\ k \textit{ else if } k \in Ka \textit{ then } Ea\ k \textit{ else } \{\})$
have $Ka \cup Kb = (Ka \cap Kb) \cup (Kb - Ka) \cup (Ka - Kb)$
by *blast*
moreover have $(\bigcap x{\in}Ka \cap Kb.\ Ea\ x \cap Eb\ x) \cap (\bigcap x{\in}Kb - Ka.\ Eb\ x) \cap (\bigcap x{\in}Ka - Kb.\ Ea\ x) = (\bigcap k{\in}Ka.\ Ea\ k) \cap (\bigcap k{\in}Kb.\ Eb\ k)$
by *auto*
ultimately have $(\bigcap k{\in}Ka \cup Kb.\ ?f\ k) = (\bigcap k{\in}Ka.\ Ea\ k) \cap (\bigcap k{\in}Kb.\ Eb\ k)$ (**is** *?lhs* = *?rhs*)
by (*simp only*: *image-Un Inter-Un-distrib*) *simp*
then have $a \cap b = (\bigcap k{\in}Ka \cup Kb.\ ?f\ k)$
by (*simp only*: *a*(*1*) *b*(*1*))
with *a b* ‹$j \in J$› *Int-stableD*[*OF Int-stable*] **show** $a \cap b \in ?E\ j$
by (*intro CollectI exI*[*of - Ka* ∪ *Kb*] *exI*[*of - ?f*]) *auto*
qed
qed
ultimately show *?thesis*
by (*simp cong*: *indep-sets-cong*)
qed

lemma (**in** *prob-space*) *indep-vars-restrict*:
assumes *ind*: *indep-vars M' X I* **and** *K*: $\bigwedge j.\ j \in L \Longrightarrow K\ j \subseteq I$ **and** *J*: *disjoint-family-on K L*
shows *indep-vars* $(\lambda j.\ PiM\ (K\ j)\ M')\ (\lambda j\ \omega.\ restrict\ (\lambda i.\ X\ i\ \omega)\ (K\ j))\ L$
unfolding *indep-vars-def*
proof *safe*
fix *j* **assume** $j \in L$ **then show** *random-variable* $(Pi_M\ (K\ j)\ M')\ (\lambda\omega.\ \lambda i{\in}K\ j.$

```
X i ω)
    using K ind by (auto simp: indep-vars-def intro!: measurable-restrict)
next
  have X: ⋀i. i ∈ I ⟹ X i ∈ measurable M (M' i)
    using ind by (auto simp: indep-vars-def)
  let ?proj = λj S. {(λω. λi∈K j. X i ω) -` A ∩ space M |A. A ∈ S}
  let ?UN = λj. sigma-sets (space M) (⋃i∈K j. { X i -` A ∩ space M| A. A ∈
sets (M' i) })
  show indep-sets (λi. sigma-sets (space M) (?proj i (sets (Pi_M (K i) M')))) L
  proof (rule indep-sets-mono-sets)
    fix j assume j: j ∈ L
    have sigma-sets (space M) (?proj j (sets (Pi_M (K j) M'))) =
     sigma-sets (space M) (sigma-sets (space M) (?proj j (prod-algebra (K j) M')))
      using j K X[THEN measurable-space] unfolding sets-PiM
      by (subst sigma-sets-vimage-commute) (auto simp add: Pi-iff)
    also have ... = sigma-sets (space M) (?proj j (prod-algebra (K j) M'))
      by (rule sigma-sets-sigma-sets-eq) auto
    also have ... ⊆ ?UN j
    proof (rule sigma-sets-mono, safe del: disjE elim!: prod-algebraE)
      fix J E assume J: finite J J ≠ {} ∨ K j = {}  J ⊆ K j and E: ∀i. i ∈ J
⟶ E i ∈ sets (M' i)
      show (λω. λi∈K j. X i ω) -` prod-emb (K j) M' J (Pi_E J E) ∩ space M ∈
?UN j
      proof cases
        assume K j = {} with J show ?thesis
          by (auto simp add: sigma-sets-empty-eq prod-emb-def)
      next
        assume K j ≠ {} with J have J ≠ {}
          by auto
        { interpret sigma-algebra space M ?UN j
            by (rule sigma-algebra-sigma-sets) auto
          have ⋀A. (⋀i. i ∈ J ⟹ A i ∈ ?UN j) ⟹ ⋂(A ` J) ∈ ?UN j
            using ‹finite J› ‹J ≠ {}› by (rule finite-INT) blast }
        note INT = this

        from ‹J ≠ {}› J K E[rule-format, THEN sets.sets-into-space] j
        have (λω. λi∈K j. X i ω) -` prod-emb (K j) M' J (Pi_E J E) ∩ space M
          = (⋂i∈J. X i -` E i ∩ space M)
          apply (subst prod-emb-PiE[OF - ])
          apply auto []
          apply auto []
          apply (auto simp add: Pi-iff intro!: X[THEN measurable-space])
          apply (erule-tac x=i in ballE)
          apply auto
          done
        also have ... ∈ ?UN j
          apply (rule INT)
          apply (rule sigma-sets.Basic)
          using ‹J ⊆ K j› E
```

```
        apply auto
        done
      finally show ?thesis .
    qed
  qed
  finally show sigma-sets (space M) (?proj j (sets (Pi_M (K j) M'))) ⊆ ?UN j .
 next
  show indep-sets ?UN L
  proof (rule indep-sets-collect-sigma)
   show indep-sets (λi. {X i -‘ A ∩ space M |A. A ∈ sets (M' i)}) (⋃j∈L. K
j)
   proof (rule indep-sets-mono-index)
    show indep-sets (λi. {X i -‘ A ∩ space M |A. A ∈ sets (M' i)}) I
     using ind unfolding indep-vars-def2 by auto
    show (⋃l∈L. K l) ⊆ I
     using K by auto
   qed
  next
   fix l i assume l ∈ L i ∈ K l
   show Int-stable {X i -‘ A ∩ space M |A. A ∈ sets (M' i)}
    apply (auto simp: Int-stable-def)
    apply (rule-tac x=A ∩ Aa in exI)
    apply auto
    done
  qed fact
 qed
qed

lemma (in prob-space) indep-var-restrict:
 assumes ind: indep-vars M' X I and AB: A ∩ B = {} A ⊆ I B ⊆ I
 shows indep-var (PiM A M') (λω. restrict (λi. X i ω) A) (PiM B M') (λω.
restrict (λi. X i ω) B)
proof -
 have ∗:
   case-bool (Pi_M A M') (Pi_M B M') = (λb. PiM (case-bool A B b) M')
   case-bool (λω. λi∈A. X i ω) (λω. λi∈B. X i ω) = (λb ω. λi∈case-bool A B b.
X i ω)
   by (simp-all add: fun-eq-iff split: bool.split)
 show ?thesis
   unfolding indep-var-def ∗ using AB
   by (intro indep-vars-restrict[OF ind]) (auto simp: disjoint-family-on-def split:
bool.split)
qed

lemma (in prob-space) indep-vars-subset:
 assumes indep-vars M' X I J ⊆ I
 shows indep-vars M' X J
 using assms unfolding indep-vars-def indep-sets-def
 by auto
```

lemma (**in** *prob-space*) *indep-vars-cong*:
  $I = J \Longrightarrow (\bigwedge i.\ i \in I \Longrightarrow X\ i = Y\ i) \Longrightarrow (\bigwedge i.\ i \in I \Longrightarrow M'\ i = N'\ i) \Longrightarrow$ *indep-vars* $M'\ X\ I \longleftrightarrow$ *indep-vars* $N'\ Y\ J$
  unfolding *indep-vars-def2* **by** (*intro conj-cong indep-sets-cong*) *auto*

definition (**in** *prob-space*) *tail-events* **where**
  tail-events $A = (\bigcap n.$ *sigma-sets* (*space* M) $(\bigcup\ (A\ `\ \{n..\})))$

lemma (**in** *prob-space*) *tail-events-sets*:
  assumes A: $\bigwedge i$::*nat*. $A\ i \subseteq$ *events*
  shows *tail-events* $A \subseteq$ *events*
proof
  fix X **assume** X: $X \in$ *tail-events* A
  let *?A* $= (\bigcap n.$ *sigma-sets* (*space* M) $(\bigcup\ (A\ `\ \{n..\})))$
  from X **have** $\bigwedge n$::*nat*. $X \in$ *sigma-sets* (*space* M) $(\bigcup\ (A\ `\ \{n..\}))$ **by** (*auto simp*: *tail-events-def*)
  from *this*[*of 0*] **have** $X \in$ *sigma-sets* (*space* M) $(\bigcup (A\ `\ UNIV))$ **by** *simp*
  then show $X \in$ *events*
    by *induct* (*insert A*, *auto*)
qed

lemma (**in** *prob-space*) *sigma-algebra-tail-events*:
  assumes $\bigwedge i$::*nat*. *sigma-algebra* (*space* M) $(A\ i)$
  shows *sigma-algebra* (*space* M) (*tail-events* A)
  unfolding *tail-events-def*
proof (*simp add*: *sigma-algebra-iff2*, *safe*)
  let *?A* $= (\bigcap n.$ *sigma-sets* (*space* M) $(\bigcup\ (A\ `\ \{n..\})))$
  interpret A: *sigma-algebra space* M A i **for** i **by** *fact*
  { fix X x **assume** $X \in$ *?A* $x \in X$
    then have $\bigwedge n.\ X \in$ *sigma-sets* (*space* M) $(\bigcup\ (A\ `\ \{n..\}))$ **by** *auto*
    from *this*[*of 0*] **have** $X \in$ *sigma-sets* (*space* M) $(\bigcup (A\ `\ UNIV))$ **by** *simp*
    then have $X \subseteq$ *space* M
      by *induct* (*insert A.sets-into-space*, *auto*)
    with ‹$x \in X$› **show** $x \in$ *space* M **by** *auto* **}**
  { fix F :: *nat* $\Rightarrow$ $'a$ *set* **and** n **assume** *range* $F \subseteq$ *?A*
    then show $(\bigcup (F\ `\ UNIV)) \in$ *sigma-sets* (*space* $M$) $(\bigcup\ (A\ `\ \{n..\}))$
      by (*intro sigma-sets.Union*) *auto* **}**
qed (*auto intro*!: *sigma-sets.Compl sigma-sets.Empty*)

lemma (**in** *prob-space*) *kolmogorov-0-1-law*:
  fixes A :: *nat* $\Rightarrow$ $'a$ *set set*
  assumes $\bigwedge i$::*nat*. *sigma-algebra* (*space* M) $(A\ i)$
  assumes *indep*: *indep-sets A UNIV*
  and X: $X \in$ *tail-events A*
  shows *prob* $X = 0 \vee$ *prob* $X = 1$
proof −
  have A: $\bigwedge i.\ A\ i \subseteq$ *events*
    using *indep* **unfolding** *indep-sets-def* **by** *simp*

let *?D* = {*D* ∈ *events*. *prob* (*X* ∩ *D*) = *prob X* ∗ *prob D*}
interpret *A*: *sigma-algebra space M A i* **for** *i* **by** *fact*
interpret *T*: *sigma-algebra space M tail-events A*
by (*rule sigma-algebra-tail-events*) *fact*
have *X* ⊆ *space M* **using** *T.space-closed X* **by** *auto*

have *X-in*: *X* ∈ *events*
using *tail-events-sets A X* **by** *auto*

interpret *D*: *Dynkin-system space M ?D*
proof (*rule Dynkin-systemI*)
fix *D* **assume** *D* ∈ *?D* **then show** *D* ⊆ *space M*
using *sets.sets-into-space* **by** *auto*
next
show *space M* ∈ *?D*
using *prob-space* ‹*X* ⊆ *space M*› **by** (*simp add*: *Int-absorb2*)
next
fix *A* **assume** *A*: *A* ∈ *?D*
have *prob* (*X* ∩ (*space M* − *A*)) = *prob* (*X* − (*X* ∩ *A*))
using ‹*X* ⊆ *space M*› **by** (*auto intro*!: *arg-cong*[**where** *f*=*prob*])
also have ... = *prob X* − *prob* (*X* ∩ *A*)
using *X-in A* **by** (*intro finite-measure-Diff*) *auto*
also have ... = *prob X* ∗ *prob* (*space M*) − *prob X* ∗ *prob A*
using *A prob-space* **by** *auto*
also have ... = *prob X* ∗ *prob* (*space M* − *A*)
using *X-in A sets.sets-into-space*
by (*subst finite-measure-Diff*) (*auto simp*: *field-simps*)
finally show *space M* − *A* ∈ *?D*
using *A* ‹*X* ⊆ *space M*› **by** *auto*
next
fix *F* :: *nat* ⇒ *'a set* **assume** *dis*: *disjoint-family F* **and** *range F* ⊆ *?D*
then have *F*: *range F* ⊆ *events* ⋀*i*. *prob* (*X* ∩ *F i*) = *prob X* ∗ *prob* (*F i*)
by *auto*
have (λ*i*. *prob* (*X* ∩ *F i*)) *sums prob* (⋃ *i*. *X* ∩ *F i*)
proof (*rule finite-measure-UNION*)
show *range* (λ*i*. *X* ∩ *F i*) ⊆ *events*
using *F X-in* **by** *auto*
show *disjoint-family* (λ*i*. *X* ∩ *F i*)
using *dis* **by** (*rule disjoint-family-on-bisimulation*) *auto*
qed
with *F* **have** (λ*i*. *prob X* ∗ *prob* (*F i*)) *sums prob* (*X* ∩ (⋃ *i*. *F i*))
by *simp*
moreover have (λ*i*. *prob X* ∗ *prob* (*F i*)) *sums* (*prob X* ∗ *prob* (⋃ *i*. *F i*))
by (*intro sums-mult finite-measure-UNION F dis*)
ultimately have *prob* (*X* ∩ (⋃ *i*. *F i*)) = *prob X* ∗ *prob* (⋃ *i*. *F i*)
by (*auto dest*!: *sums-unique*)
with *F* **show** (⋃ *i*. *F i*) ∈ *?D*
by *auto*

qed

{ fix *n*
have *indep-sets* (*λb. sigma-sets* (*space M*) (⋃*m*∈*case-bool* {..*n*} {*Suc n*..} *b. A m*)) *UNIV*
proof (*rule indep-sets-collect-sigma*)
have ∗: (⋃*b. case b of True* ⇒ {..*n*} | *False* ⇒ {*Suc n*..}) = *UNIV* (**is** *?U* = -)
by (*simp split*: *bool.split add*: *set-eq-iff*) (*metis not-less-eq-eq*)
with *indep* **show** *indep-sets A ?U* **by** *simp*
show *disjoint-family* (*case-bool* {..*n*} {*Suc n*..})
unfolding *disjoint-family-on-def* **by** (*auto split*: *bool.split*)
fix *m*
show *Int-stable* (*A m*)
unfolding *Int-stable-def* **using** *A.Int* **by** *auto*
qed
also have (*λb. sigma-sets* (*space M*) (⋃*m*∈*case-bool* {..*n*} {*Suc n*..} *b. A m*)) =
case-bool (*sigma-sets* (*space M*) (⋃*m*∈{..*n*}. *A m*)) (*sigma-sets* (*space M*) (⋃*m*∈{*Suc n*..}. *A m*))
by (*auto intro*!: *ext split*: *bool.split*)
finally have *indep*: *indep-set* (*sigma-sets* (*space M*) (⋃*m*∈{..*n*}. *A m*)) (*sigma-sets* (*space M*) (⋃*m*∈{*Suc n*..}. *A m*))
unfolding *indep-set-def* **by** *simp*

have *sigma-sets* (*space M*) (⋃*m*∈{..*n*}. *A m*) ⊆ *?D*
proof (*simp add*: *subset-eq*, *rule*)
fix *D* **assume** *D*: *D* ∈ *sigma-sets* (*space M*) (⋃*m*∈{..*n*}. *A m*)
have *X* ∈ *sigma-sets* (*space M*) (⋃*m*∈{*Suc n*..}. *A m*)
using *X* **unfolding** *tail-events-def* **by** *simp*
from *indep-setD*[*OF indep D this*] *indep-setD-ev1*[*OF indep*] *D*
show *D* ∈ *events* ∧ *prob* (*X* ∩ *D*) = *prob X* ∗ *prob D*
by (*auto simp add*: *ac-simps*)
qed }
then have (⋃*n. sigma-sets* (*space M*) (⋃*m*∈{..*n*}. *A m*)) ⊆ *?D* (**is** *?A* ⊆ -)
by *auto*

note ‹*X* ∈ *tail-events A*›
also {
have ⋀*n. sigma-sets* (*space M*) (⋃*i*∈{*n*..}. *A i*) ⊆ *sigma-sets* (*space M*) *?A*
by (*intro sigma-sets-subseteq UN-mono*) *auto*
then have *tail-events A* ⊆ *sigma-sets* (*space M*) *?A*
unfolding *tail-events-def* **by** *auto* **}**
also have *sigma-sets* (*space M*) *?A* = *Dynkin* (*space M*) *?A*
proof (*rule sigma-eq-Dynkin*)
{ fix *B n* **assume** *B* ∈ *sigma-sets* (*space M*) (⋃*m*∈{..*n*}. *A m*)
then have *B* ⊆ *space M*
by *induct* (*insert A sets.sets-into-space*[*of* - *M*], *auto*) **}**
then show *?A* ⊆ *Pow* (*space M*) **by** *auto*

```
    show Int-stable ?A
    proof (rule Int-stableI)
      fix a b assume a ∈ ?A b ∈ ?A then obtain n m
        where a: n ∈ UNIV a ∈ sigma-sets (space M) (⋃ (A ` {..n}))
          and b: m ∈ UNIV b ∈ sigma-sets (space M) (⋃ (A ` {..m})) by auto
      interpret Amn: sigma-algebra space M sigma-sets (space M) (⋃ i∈{..max m
n}. A i)
        using A sets.sets-into-space[of - M] by (intro sigma-algebra-sigma-sets) auto
      have sigma-sets (space M) (⋃ i∈{..n}. A i) ⊆ sigma-sets (space M) (⋃ i∈{..max
m n}. A i)
        by (intro sigma-sets-subseteq UN-mono) auto
      with a have a ∈ sigma-sets (space M) (⋃ i∈{..max m n}. A i) by auto
      moreover
      have sigma-sets (space M) (⋃ i∈{..m}. A i) ⊆ sigma-sets (space M) (⋃ i∈{..max
m n}. A i)
        by (intro sigma-sets-subseteq UN-mono) auto
      with b have b ∈ sigma-sets (space M) (⋃ i∈{..max m n}. A i) by auto
      ultimately have a ∩ b ∈ sigma-sets (space M) (⋃ i∈{..max m n}. A i)
        using Amn.Int[of a b] by simp
      then show a ∩ b ∈ (⋃ n. sigma-sets (space M) (⋃ i∈{..n}. A i)) by auto
    qed
  qed
  also have Dynkin (space M) ?A ⊆ ?D
    using ‹?A ⊆ ?D› by (auto intro!: D.Dynkin-subset)
  finally show ?thesis by auto
qed

lemma (in prob-space) borel-0-1-law:
  fixes F :: nat ⇒ 'a set
  assumes F2: indep-events F UNIV
  shows prob (⋂ n. ⋃ m∈{n..}. F m) = 0 ∨ prob (⋂ n. ⋃ m∈{n..}. F m) = 1
proof (rule kolmogorov-0-1-law[of λi. sigma-sets (space M) { F i }])
  have F1: range F ⊆ events
    using F2 by (simp add: indep-events-def subset-eq)
  { fix i show sigma-algebra (space M) (sigma-sets (space M) {F i})
      using sigma-algebra-sigma-sets[of {F i} space M] F1 sets.sets-into-space
      by auto }
  show indep-sets (λi. sigma-sets (space M) {F i}) UNIV
  proof (rule indep-sets-sigma)
    show indep-sets (λi. {F i}) UNIV
      unfolding indep-events-def-alt[symmetric] by fact
    fix i show Int-stable {F i}
      unfolding Int-stable-def by simp
  qed
  let ?Q = λn. ⋃ i∈{n..}. F i
  show (⋂ n. ⋃ m∈{n..}. F m) ∈ tail-events (λi. sigma-sets (space M) {F i})
    unfolding tail-events-def
  proof
    fix j
```

interpret *S*: *sigma-algebra space M sigma-sets* (*space M*) (⋃ *i*∈{*j*..}. *sigma-sets* (*space M*) {*F i*})
using *order-trans*[*OF F1 sets.space-closed*]
by (*intro sigma-algebra-sigma-sets*) (*simp add*: *sigma-sets-singleton subset-eq*)
have (⋂ *n*. *?Q n*) = (⋂ *n*∈{*j*..}. *?Q n*)
by (*intro decseq-SucI INT-decseq-offset UN-mono*) *auto*
also have ... ∈ *sigma-sets* (*space M*) (⋃ *i*∈{*j*..}. *sigma-sets* (*space M*) {*F i*})
using *order-trans*[*OF F1 sets.space-closed*]
by (*safe intro*!: *S.countable-INT S.countable-UN*)
(*auto simp*: *sigma-sets-singleton intro*!: *sigma-sets.Basic bexI*)
finally show (⋂ *n*. *?Q n*) ∈ *sigma-sets* (*space M*) (⋃ *i*∈{*j*..}. *sigma-sets* (*space M*) {*F i*})
by *simp*
qed
qed

lemma (**in** *prob-space*) *borel-0-1-law-AE*:
fixes *P* :: *nat* ⇒ *'a* ⇒ *bool*
assumes *indep-events* (λ*m*. {*x*∈*space M*. *P m x*}) *UNIV* (**is** *indep-events ?P* -)
shows (*AE x in M*. *infinite* {*m*. *P m x*}) ∨ (*AE x in M*. *finite* {*m*. *P m x*})
proof −
have [*measurable*]: ⋀*m*. {*x*∈*space M*. *P m x*} ∈ *sets M*
using *assms* **by** (*auto simp*: *indep-events-def*)
have ∗: (⋂ *n*. ⋃ *m*∈{*n*..}. {*x* ∈ *space M*. *P m x*}) ∈ *events*
by *simp*
from *assms* **have** *prob* (⋂ *n*. ⋃ *m*∈{*n*..}. *?P m*) = *0* ∨ *prob* (⋂ *n*. ⋃ *m*∈{*n*..}. *?P m*) = *1*
by (*rule borel-0-1-law*)
also have *prob* (⋂ *n*. ⋃ *m*∈{*n*..}. *?P m*) = *1* ⟷ (*AE x in M*. *infinite* {*m*. *P m x*})
using ∗ **by** (*simp add*: *prob-eq-1*)
(*simp add*: *Bex-def infinite-nat-iff-unbounded-le*)
also have *prob* (⋂ *n*. ⋃ *m*∈{*n*..}. *?P m*) = *0* ⟷ (*AE x in M*. *finite* {*m*. *P m x*})
using ∗ **by** (*simp add*: *prob-eq-0*)
(*auto simp add*: *Ball-def finite-nat-iff-bounded not-less* [*symmetric*])
finally show *?thesis*
by *blast*
qed

lemma (**in** *prob-space*) *indep-sets-finite*:
assumes *I*: *I* ≠ {} *finite I*
and *F*: ⋀*i*. *i* ∈ *I* ⟹ *F i* ⊆ *events* ⋀*i*. *i* ∈ *I* ⟹ *space M* ∈ *F i*
shows *indep-sets F I* ⟷ (∀ *A*∈*Pi I F*. *prob* (⋂ *j*∈*I*. *A j*) = (∏ *j*∈*I*. *prob* (*A j*)))
proof
assume ∗: *indep-sets F I*
from *I* **show** ∀ *A*∈*Pi I F*. *prob* (⋂ *j*∈*I*. *A j*) = (∏ *j*∈*I*. *prob* (*A j*))
by (*intro indep-setsD*[*OF* ∗] *ballI*) *auto*

next
assume *indep*: $\forall A{\in}Pi\ I\ F.\ prob\ (\bigcap j{\in}I.\ A\ j) = (\prod j{\in}I.\ prob\ (A\ j))$
show *indep-sets F I*
proof (*rule indep-setsI*[*OF F*(*1*)])
fix *A J* **assume** *J*: $J \neq \{\}$ $J \subseteq I$ *finite J*
assume *A*: $\forall j{\in}J.\ A\ j \in F\ j$
let *?A* = $\lambda j.$ *if j* $\in$ *J then A j else space M*
have *prob* $(\bigcap j{\in}I.$ *?A j*$) =$ *prob* $(\bigcap j{\in}J.$ *A j*$)$
using *subset-trans*[*OF F*(*1*) *sets.space-closed*] *J A*
by (*auto intro*!: *arg-cong*[**where** *f*=*prob*] *split*: *if-split-asm*) *blast*
also
from *A F* **have** $(\lambda j.$ *if j* $\in$ *J then A j else space M*$) \in$ *Pi I F* (**is** *?A* $\in$ -)
by (*auto split*: *if-split-asm*)
with *indep* **have** *prob* $(\bigcap j{\in}I.$ *?A j*$) = (\prod j{\in}I.$ *prob* (*?A j*))
by *auto*
also have $\ldots = (\prod j{\in}J.$ *prob* (*A j*))
unfolding *if-distrib prod.If-cases*[*OF* ‹*finite I*›]
using *prob-space* ‹*J* $\subseteq$ *I*› **by** (*simp add*: *Int-absorb1 prod.neutral-const*)
finally show *prob* $(\bigcap j{\in}J.$ *A j*$) = (\prod j{\in}J.$ *prob* (*A j*)) **..**
qed
qed

lemma (**in** *prob-space*) *indep-vars-finite*:
fixes *I* :: *'i set*
assumes *I*: $I \neq \{\}$ *finite I*
and *M'*: $\bigwedge i.\ i \in I \Longrightarrow$ *sets* (*M' i*) = *sigma-sets* (*space* (*M' i*)) (*E i*)
and *rv*: $\bigwedge i.\ i \in I \Longrightarrow$ *random-variable* (*M' i*) (*X i*)
and *Int-stable*: $\bigwedge i.\ i \in I \Longrightarrow$ *Int-stable* (*E i*)
and *space*: $\bigwedge i.\ i \in I \Longrightarrow$ *space* (*M' i*) $\in$ *E i* **and** *closed*: $\bigwedge i.\ i \in I \Longrightarrow E\ i \subseteq$ *Pow* (*space* (*M' i*))
shows *indep-vars M' X I* $\longleftrightarrow$
$(\forall A{\in}(\Pi\ i{\in}I.\ E\ i).$ *prob* $(\bigcap j{\in}I.\ X\ j\ -`\ A\ j \cap$ *space M*$) = (\prod j{\in}I.$ *prob* (*X j* $-`$ *A j* $\cap$ *space M*)))
proof −
from *rv* **have** *X*: $\bigwedge i.\ i \in I \Longrightarrow X\ i \in$ *space M* $\to$ *space* (*M' i*)
unfolding *measurable-def* **by** *simp*

{ fix *i* **assume** *i*$\in$*I*
from *closed*[*OF* ‹*i* $\in$ *I*›]
have *sigma-sets* (*space M*) $\{X\ i\ -`\ A \cap$ *space M* $|A.\ A \in$ *sets* (*M' i*)$\}$
$=$ *sigma-sets* (*space M*) $\{X\ i\ -`\ A \cap$ *space M* $|A.\ A \in E\ i\}$
unfolding *sigma-sets-vimage-commute*[*OF X*, *OF* ‹*i* $\in$ *I*›, *symmetric*] *M'*[*OF* ‹*i* $\in$ *I*›]
by (*subst sigma-sets-sigma-sets-eq*) *auto* **}**
note *sigma-sets-X* = *this*

{ fix *i* **assume** *i*$\in$*I*
have *Int-stable* $\{X\ i\ -`\ A \cap$ *space M* $|A.\ A \in E\ i\}$
proof (*rule Int-stableI*)

```
    fix a assume a ∈ {X i −‘ A ∩ space M |A. A ∈ E i}
    then obtain A where a = X i −‘ A ∩ space M A ∈ E i by auto
    moreover
    fix b assume b ∈ {X i −‘ A ∩ space M |A. A ∈ E i}
    then obtain B where b = X i −‘ B ∩ space M B ∈ E i by auto
    moreover
    have (X i −‘ A ∩ space M) ∩ (X i −‘ B ∩ space M) = X i −‘ (A ∩ B) ∩
space M by auto
    moreover note Int-stable[OF ‹i ∈ I›]
    ultimately
    show a ∩ b ∈ {X i −‘ A ∩ space M |A. A ∈ E i}
      by (auto simp del: vimage-Int intro!: exI[of - A ∩ B] dest: Int-stableD)
  qed }
 note indep-sets-X = indep-sets-sigma-sets-iff[OF this]

 { fix i assume i ∈ I
  { fix A assume A ∈ E i
    with M'[OF ‹i ∈ I›] have A ∈ sets (M' i) by auto
    moreover
    from rv[OF ‹i∈I›] have X i ∈ measurable M (M' i) by auto
    ultimately
    have X i −‘ A ∩ space M ∈ sets M by (auto intro: measurable-sets) }
  with X[OF ‹i∈I›] space[OF ‹i∈I›]
  have {X i −‘ A ∩ space M |A. A ∈ E i} ⊆ events
    space M ∈ {X i −‘ A ∩ space M |A. A ∈ E i}
    by (auto intro!: exI[of - space (M' i)]) }
 note indep-sets-finite-X = indep-sets-finite[OF I this]

 have (∀ A∈Π i∈I. {X i −‘ A ∩ space M |A. A ∈ E i}. prob (⋂(A ‘ I)) = (∏j∈I.
prob (A j))) =
   (∀ A∈Π i∈I. E i. prob ((⋂j∈I. X j −‘ A j) ∩ space M) = (∏ x∈I. prob (X x
−‘ A x ∩ space M)))
   (is ?L = ?R)
 proof safe
   fix A assume ?L and A: A ∈ (Π i∈I. E i)
   from ‹?L›[THEN bspec, of λi. X i −‘ A i ∩ space M] A ‹I ≠ {}›
   show prob ((⋂j∈I. X j −‘ A j) ∩ space M) = (∏ x∈I. prob (X x −‘ A x ∩
space M))
     by (auto simp add: Pi-iff)
 next
   fix A assume ?R and A: A ∈ (Π i∈I. {X i −‘ A ∩ space M |A. A ∈ E i})
   from A have ∀ i∈I. ∃ B. A i = X i −‘ B ∩ space M ∧ B ∈ E i by auto
   from bchoice[OF this] obtain B where B: ∀ i∈I. A i = X i −‘ B i ∩ space M
     B ∈ (Π i∈I. E i) by auto
   from ‹?R›[THEN bspec, OF B(2)] B(1) ‹I ≠ {}›
   show prob (⋂(A ‘ I)) = (∏j∈I. prob (A j))
     by simp
 qed
 then show ?thesis using ‹I ≠ {}›
```

by (*simp add*: *rv indep-vars-def indep-sets-X sigma-sets-X indep-sets-finite-X cong*: *indep-sets-cong*)
qed

lemma (**in** *prob-space*) *indep-vars-compose*:
assumes *indep-vars M' X I*
assumes *rv*: $\bigwedge i.\ i \in I \Longrightarrow Y\ i \in measurable\ (M'\ i)\ (N\ i)$
shows *indep-vars N* $(\lambda i.\ Y\ i \circ X\ i)$ *I*
unfolding *indep-vars-def*
proof
from *rv* ‹*indep-vars M' X I*›
show $\forall i \in I.$ *random-variable* $(N\ i)$ $(Y\ i \circ X\ i)$
by (*auto simp*: *indep-vars-def*)

have *indep-sets* $(\lambda i.$ *sigma-sets* $(space\ M)$ $\{X\ i -`\ A \cap space\ M\ |A.\ A \in sets\ (M'\ i)\})$ *I*
using ‹*indep-vars M' X I*› **by** (*simp add*: *indep-vars-def*)
then show *indep-sets* $(\lambda i.$ *sigma-sets* $(space\ M)$ $\{(Y\ i \circ X\ i) -`\ A \cap space\ M\ |A.\ A \in sets\ (N\ i)\})$ *I*
proof (*rule indep-sets-mono-sets*)
fix *i* **assume** $i \in I$
with ‹*indep-vars M' X I*› **have** *X*: $X\ i \in space\ M \to space\ (M'\ i)$
unfolding *indep-vars-def measurable-def* **by** *auto*
{ fix *A* **assume** $A \in sets\ (N\ i)$
then have $\exists B.\ (Y\ i \circ X\ i) -`\ A \cap space\ M = X\ i -`\ B \cap space\ M \wedge B \in sets\ (M'\ i)$
by (*intro exI*[*of* - $Y\ i -`\ A \cap space\ (M'\ i)$])
(*auto simp*: *vimage-comp intro*!: *measurable-sets rv* ‹$i \in I$› *funcset-mem*[*OF X*]) **}**
then show *sigma-sets* $(space\ M)$ $\{(Y\ i \circ X\ i) -`\ A \cap space\ M\ |A.\ A \in sets\ (N\ i)\} \subseteq$
sigma-sets $(space\ M)$ $\{X\ i -`\ A \cap space\ M\ |A.\ A \in sets\ (M'\ i)\}$
by (*intro sigma-sets-subseteq*) (*auto simp*: *vimage-comp*)
qed
qed

lemma (**in** *prob-space*) *indep-vars-compose2*:
assumes *indep-vars M' X I*
assumes *rv*: $\bigwedge i.\ i \in I \Longrightarrow Y\ i \in measurable\ (M'\ i)\ (N\ i)$
shows *indep-vars N* $(\lambda i\ x.\ Y\ i\ (X\ i\ x))$ *I*
using *indep-vars-compose* [*OF assms*] **by** (*simp add*: *comp-def*)

lemma (**in** *prob-space*) *indep-var-compose*:
assumes *indep-var M1 X1 M2 X2 Y1* $\in$ *measurable M1 N1 Y2* $\in$ *measurable M2 N2*
shows *indep-var N1* (*Y1* $\circ$ *X1*) *N2* (*Y2* $\circ$ *X2*)
proof −
have *indep-vars* (*case-bool N1 N2*) $(\lambda b.$ *case-bool Y1 Y2 b* $\circ$ *case-bool X1 X2 b*) *UNIV*

using *assms*
by (*intro indep-vars-compose*[**where** *M'=case-bool M1 M2*])
(*auto simp*: *indep-var-def split*: *bool.split*)
also have (λ*b*. *case-bool Y1 Y2 b* ∘ *case-bool X1 X2 b*) = *case-bool* (*Y1* ∘ *X1*) (*Y2* ∘ *X2*)
by (*simp add*: *fun-eq-iff split*: *bool.split*)
finally show *?thesis*
unfolding *indep-var-def* .
qed

lemma (**in** *prob-space*) *indep-vars-Min*:
fixes *X* :: *'i* ⇒ *'a* ⇒ *real*
assumes *I*: *finite I i* ∉ *I* **and** *indep*: *indep-vars* (λ-. *borel*) *X* (*insert i I*)
shows *indep-var borel* (*X i*) *borel* (λω. *Min* ((λ*i*. *X i* ω)‘*I*))
proof −
have *indep-var*
borel ((λ*f*. *f i*) ∘ (λω. *restrict* (λ*i*. *X i* ω) {*i*}))
borel ((λ*f*. *Min* (*f*‘*I*)) ∘ (λω. *restrict* (λ*i*. *X i* ω) *I*))
using *I* **by** (*intro indep-var-compose*[*OF indep-var-restrict*[*OF indep*]] *borel-measurable-Min*) *auto*
also have ((λ*f*. *f i*) ∘ (λω. *restrict* (λ*i*. *X i* ω) {*i*})) = *X i*
by *auto*
also have ((λ*f*. *Min* (*f*‘*I*)) ∘ (λω. *restrict* (λ*i*. *X i* ω) *I*)) = (λω. *Min* ((λ*i*. *X i* ω)‘*I*))
by (*auto cong*: *rev-conj-cong*)
finally show *?thesis*
unfolding *indep-var-def* .
qed

lemma (**in** *prob-space*) *indep-vars-sum*:
fixes *X* :: *'i* ⇒ *'a* ⇒ *real*
assumes *I*: *finite I i* ∉ *I* **and** *indep*: *indep-vars* (λ-. *borel*) *X* (*insert i I*)
shows *indep-var borel* (*X i*) *borel* (λω. ∑*i*∈*I*. *X i* ω)
proof −
have *indep-var*
borel ((λ*f*. *f i*) ∘ (λω. *restrict* (λ*i*. *X i* ω) {*i*}))
borel ((λ*f*. ∑*i*∈*I*. *f i*) ∘ (λω. *restrict* (λ*i*. *X i* ω) *I*))
using *I* **by** (*intro indep-var-compose*[*OF indep-var-restrict*[*OF indep*]]) *auto*
also have ((λ*f*. *f i*) ∘ (λω. *restrict* (λ*i*. *X i* ω) {*i*})) = *X i*
by *auto*
also have ((λ*f*. ∑*i*∈*I*. *f i*) ∘ (λω. *restrict* (λ*i*. *X i* ω) *I*)) = (λω. ∑*i*∈*I*. *X i* ω)
by (*auto cong*: *rev-conj-cong*)
finally show *?thesis* .
qed

lemma (**in** *prob-space*) *indep-vars-prod*:
fixes *X* :: *'i* ⇒ *'a* ⇒ *real*
assumes *I*: *finite I i* ∉ *I* **and** *indep*: *indep-vars* (λ-. *borel*) *X* (*insert i I*)
shows *indep-var borel* (*X i*) *borel* (λω. ∏*i*∈*I*. *X i* ω)

proof −
have *indep-var*
borel ((λ*f*. *f i*) ∘ (λω. *restrict* (λ*i*. *X i* ω) {*i*}))
borel ((λ*f*. ∏ *i*∈*I*. *f i*) ∘ (λω. *restrict* (λ*i*. *X i* ω) *I*))
using *I* **by** (*intro indep-var-compose*[*OF indep-var-restrict*[*OF indep*]]) *auto*
also have ((λ*f*. *f i*) ∘ (λω. *restrict* (λ*i*. *X i* ω) {*i*})) = *X i*
by *auto*
also have ((λ*f*. ∏ *i*∈*I*. *f i*) ∘ (λω. *restrict* (λ*i*. *X i* ω) *I*)) = (λω. ∏ *i*∈*I*. *X i* ω)
by (*auto cong*: *rev-conj-cong*)
finally show *?thesis* **.**
qed

lemma (**in** *prob-space*) *indep-varsD-finite*:
assumes *X*: *indep-vars M' X I*
assumes *I*: *I* ≠ {} *finite I* ⋀*i*. *i* ∈ *I* ⟹ *A i* ∈ *sets* (*M' i*)
shows *prob* (⋂ *i*∈*I*. *X i* −‘ *A i* ∩ *space M*) = (∏ *i*∈*I*. *prob* (*X i* −‘ *A i* ∩ *space M*))
proof (*rule indep-setsD*)
show *indep-sets* (λ*i*. *sigma-sets* (*space M*) {*X i* −‘ *A* ∩ *space M* |*A*. *A* ∈ *sets* (*M' i*)}) *I*
using *X* **by** (*auto simp*: *indep-vars-def*)
show *I* ⊆ *I I* ≠ {} *finite I* **using** *I* **by** *auto*
show ∀ *i*∈*I*. *X i* −‘ *A i* ∩ *space M* ∈ *sigma-sets* (*space M*) {*X i* −‘ *A* ∩ *space M* |*A*. *A* ∈ *sets* (*M' i*)}
using *I* **by** *auto*
qed

lemma (**in** *prob-space*) *indep-varsD*:
assumes *X*: *indep-vars M' X I*
assumes *I*: *J* ≠ {} *finite J J* ⊆ *I* ⋀*i*. *i* ∈ *J* ⟹ *A i* ∈ *sets* (*M' i*)
shows *prob* (⋂ *i*∈*J*. *X i* −‘ *A i* ∩ *space M*) = (∏ *i*∈*J*. *prob* (*X i* −‘ *A i* ∩ *space M*))
proof (*rule indep-setsD*)
show *indep-sets* (λ*i*. *sigma-sets* (*space M*) {*X i* −‘ *A* ∩ *space M* |*A*. *A* ∈ *sets* (*M' i*)}) *I*
using *X* **by** (*auto simp*: *indep-vars-def*)
show ∀ *i*∈*J*. *X i* −‘ *A i* ∩ *space M* ∈ *sigma-sets* (*space M*) {*X i* −‘ *A* ∩ *space M* |*A*. *A* ∈ *sets* (*M' i*)}
using *I* **by** *auto*
qed *fact+*

lemma (**in** *prob-space*) *indep-vars-iff-distr-eq-PiM*:
fixes *I* :: *'i set* **and** *X* :: *'i* ⇒ *'a* ⇒ *'b*
assumes *I* ≠ {}
assumes *rv*: ⋀*i*. *random-variable* (*M' i*) (*X i*)
shows *indep-vars M' X I* ⟷
distr M (Π*M* *i*∈*I*. *M' i*) (λ*x*. λ*i*∈*I*. *X i x*) = (Π*M* *i*∈*I*. *distr M* (*M' i*) (*X i*))
proof −
let *?P* = Π*M* *i*∈*I*. *M' i*

let *?X* = $\lambda x.\ \lambda i{\in}I.\ X\ i\ x$
let *?D* = *distr M ?P ?X*
have *X*: *random-variable ?P ?X* **by** (*intro measurable-restrict rv*)
interpret *D*: *prob-space ?D* **by** (*intro prob-space-distr X*)

let *?D′* = $\lambda i.$ *distr M* (*M′ i*) (*X i*)
let *?P′* = $\Pi_M\ i{\in}I.$ *distr M* (*M′ i*) (*X i*)
interpret *D′*: *prob-space ?D′ i* **for** *i* **by** (*intro prob-space-distr rv*)
interpret *P*: *product-prob-space ?D′ I* **..**

show *?thesis*
proof
assume *indep-vars M′ X I*
show *?D* = *?P′*
proof (*rule measure-eqI-generator-eq*)
show *Int-stable* (*prod-algebra I M′*)
by (*rule Int-stable-prod-algebra*)
show *prod-algebra I M′* ⊆ *Pow* (*space ?P*)
using *prod-algebra-sets-into-space* **by** (*simp add*: *space-PiM*)
show *sets ?D* = *sigma-sets* (*space ?P*) (*prod-algebra I M′*)
by (*simp add*: *sets-PiM space-PiM*)
show *sets ?P′* = *sigma-sets* (*space ?P*) (*prod-algebra I M′*)
by (*simp add*: *sets-PiM space-PiM cong*: *prod-algebra-cong*)
let *?A* = $\lambda i.\ \Pi_E\ i{\in}I.$ *space* (*M′ i*)
show *range ?A* ⊆ *prod-algebra I M′* ($\bigcup i.$ *?A i*) = *space* (Pi_M *I M′*)
by (*auto simp*: *space-PiM intro*!: *space-in-prod-algebra cong*: *prod-algebra-cong*)
{ fix *i* **show** *emeasure ?D* ($\Pi_E\ i{\in}I.$ *space* (*M′ i*)) ≠ ∞ **by** *auto* **}**
next
fix *E* **assume** *E*: *E* ∈ *prod-algebra I M′*
from *prod-algebraE*[*OF E*] **obtain** *J Y*
where *J*:
E = *prod-emb I M′ J* (Pi_E *J Y*)
finite J
J ≠ {} ∨ *I* = {}
J ⊆ *I*
$\bigwedge i.\ i \in J \Longrightarrow Y\ i \in$ *sets* (*M′ i*)
by *auto*
from *E* **have** *E* ∈ *sets ?P* **by** (*auto simp*: *sets-PiM*)
then have *emeasure ?D E* = *emeasure M* (*?X* −‘ *E* ∩ *space M*)
by (*simp add*: *emeasure-distr X*)
also have *?X* −‘ *E* ∩ *space M* = ($\bigcap i{\in}J.$ *X i* −‘ *Y i* ∩ *space M*)
using *J* ‹*I* ≠ {}› *measurable-space*[*OF rv*] **by** (*auto simp*: *prod-emb-def PiE-iff split*: *if-split-asm*)
also have *emeasure M* ($\bigcap i{\in}J.$ *X i* −‘ *Y i* ∩ *space M*) = ($\prod\ i{\in}J.$ *emeasure M* (*X i* −‘ *Y i* ∩ *space M*))
using ‹*indep-vars M′ X I*› *J* ‹*I* ≠ {}› **using** *indep-varsD*[*of M′ X I J*]
by (*auto simp*: *emeasure-eq-measure prod-ennreal measure-nonneg prod-nonneg*)
also have ... = ($\prod\ i{\in}J.$ *emeasure* (*?D′ i*) (*Y i*))
using *rv J* **by** (*simp add*: *emeasure-distr*)

```
    also have ... = emeasure ?P' E
      using P.emeasure-PiM-emb[of J Y] J by (simp add: prod-emb-def)
    finally show emeasure ?D E = emeasure ?P' E .
  qed
 next
  assume ?D = ?P'
  show indep-vars M' X I unfolding indep-vars-def
  proof (intro conjI indep-setsI ballI rv)
    fix i show sigma-sets (space M) {X i -` A ∩ space M |A. A ∈ sets (M' i)}
⊆ events
      by (auto intro!: sets.sigma-sets-subset measurable-sets rv)
  next
    fix J Y' assume J: J ≠ {} J ⊆ I finite J
    assume Y': ∀j∈J. Y' j ∈ sigma-sets (space M) {X j -` A ∩ space M |A. A
∈ sets (M' j)}
    have ∀j∈J. ∃ Y. Y' j = X j -` Y ∩ space M ∧ Y ∈ sets (M' j)
    proof
      fix j assume j ∈ J
      from Y'[rule-format, OF this] rv[of j]
      show ∃ Y. Y' j = X j -` Y ∩ space M ∧ Y ∈ sets (M' j)
        by (subst (asm) sigma-sets-vimage-commute[symmetric, of - - space (M'
j)])
           (auto dest: measurable-space simp: sets.sigma-sets-eq)
    qed
    from bchoice[OF this] obtain Y where
      Y: ⋀j. j ∈ J ⟹ Y' j = X j -` Y j ∩ space M ⋀j. j ∈ J ⟹ Y j ∈ sets
(M' j) by auto
    let ?E = prod-emb I M' J (Pi_E J Y)
    from Y have (⋂j∈J. Y' j) = ?X -` ?E ∩ space M
      using J ‹I ≠ {}› measurable-space[OF rv] by (auto simp: prod-emb-def
PiE-iff split: if-split-asm)
    then have emeasure M (⋂j∈J. Y' j) = emeasure M (?X -` ?E ∩ space M)
      by simp
    also have ... = emeasure ?D ?E
      using Y J by (intro emeasure-distr[symmetric] X sets-PiM-I) auto
    also have ... = emeasure ?P' ?E
      using ‹?D = ?P'› by simp
    also have ... = (∏ i∈J. emeasure (?D' i) (Y i))
      using P.emeasure-PiM-emb[of J Y] J Y by (simp add: prod-emb-def)
    also have ... = (∏ i∈J. emeasure M (Y' i))
      using rv J Y by (simp add: emeasure-distr)
    finally have emeasure M (⋂j∈J. Y' j) = (∏ i∈J. emeasure M (Y' i)) .
    then show prob (⋂j∈J. Y' j) = (∏ i∈J. prob (Y' i))
    by (auto simp: emeasure-eq-measure prod-ennreal measure-nonneg prod-nonneg)
  qed
 qed
qed

lemma (in prob-space) indep-vars-iff-distr-eq-PiM':
```

fixes $I :: 'i\ set$ **and** $X :: 'i \Rightarrow 'a \Rightarrow 'b$
assumes $I \neq \{\}$
assumes *rv*: $\bigwedge i.\ i \in I \Longrightarrow$ *random-variable* $(M'\ i)\ (X\ i)$
shows *indep-vars* $M'\ X\ I \longleftrightarrow$
distr $M\ (\Pi_M\ i{\in}I.\ M'\ i)\ (\lambda x.\ \lambda i{\in}I.\ X\ i\ x) = (\Pi_M\ i{\in}I.\ distr\ M\ (M'\ i)\ (X\ i))$
proof −
from *assms* **obtain** j **where** *j*: $j \in I$
by *auto*
define N' **where** $N' = (\lambda i.\ if\ i \in I\ then\ M'\ i\ else\ M'\ j)$
define Y **where** $Y = (\lambda i.\ if\ i \in I\ then\ X\ i\ else\ X\ j)$
have *rv*: *random-variable* $(N'\ i)\ (Y\ i)$ **for** i
using j **by** (*auto simp*: *N'-def Y-def intro*: *assms*)

have *indep-vars* $M'\ X\ I =$ *indep-vars* $N'\ Y\ I$
by (*intro indep-vars-cong*) (*auto simp*: *N'-def Y-def*)
also have ... $\longleftrightarrow$ *distr* $M\ (\Pi_M\ i{\in}I.\ N'\ i)\ (\lambda x.\ \lambda i{\in}I.\ Y\ i\ x) = (\Pi_M\ i{\in}I.\ distr\ M\ (N'\ i)\ (Y\ i))$
by (*intro indep-vars-iff-distr-eq-PiM rv assms*)
also have $(\Pi_M\ i{\in}I.\ N'\ i) = (\Pi_M\ i{\in}I.\ M'\ i)$
by (*intro PiM-cong*) (*simp-all add*: *N'-def*)
also have $(\lambda x.\ \lambda i{\in}I.\ Y\ i\ x) = (\lambda x.\ \lambda i{\in}I.\ X\ i\ x)$
by (*simp-all add*: *Y-def fun-eq-iff*)
also have $(\Pi_M\ i{\in}I.\ distr\ M\ (N'\ i)\ (Y\ i)) = (\Pi_M\ i{\in}I.\ distr\ M\ (M'\ i)\ (X\ i))$
by (*intro PiM-cong distr-cong*) (*simp-all add*: *N'-def Y-def*)
finally show *?thesis* **.**
qed

lemma (**in** *prob-space*) *indep-varD*:
assumes *indep*: *indep-var Ma A Mb B*
assumes *sets*: $Xa \in sets\ Ma\ Xb \in sets\ Mb$
shows *prob* $((\lambda x.\ (A\ x,\ B\ x)) -`\ (Xa \times Xb) \cap space\ M) =$
prob $(A -`\ Xa \cap space\ M) * prob\ (B -`\ Xb \cap space\ M)$
proof −
have *prob* $((\lambda x.\ (A\ x,\ B\ x)) -`\ (Xa \times Xb) \cap space\ M) =$
prob $(\bigcap i{\in}UNIV.\ (case\text{-}bool\ A\ B\ i -`\ case\text{-}bool\ Xa\ Xb\ i \cap space\ M))$
by (*auto intro*!: *arg-cong*[**where** *f*=*prob*] *simp*: *UNIV-bool*)
also have ... $= (\prod i{\in}UNIV.\ prob\ (case\text{-}bool\ A\ B\ i -`\ case\text{-}bool\ Xa\ Xb\ i \cap space\ M))$
using *indep* **unfolding** *indep-var-def*
by (*rule indep-varsD*) (*auto split*: *bool.split intro*: *sets*)
also have ... $=$ *prob* $(A -`\ Xa \cap space\ M) * prob\ (B -`\ Xb \cap space\ M)$
unfolding *UNIV-bool* **by** *simp*
finally show *?thesis* **.**
qed

lemma (**in** *prob-space*) *prob-indep-random-variable*:
assumes *ind*[*simp*]: *indep-var N X N Y*
assumes [*simp*]: $A \in sets\ N\ B \in sets\ N$

shows $\mathcal{P}(x \text{ in } M.\ X\ x \in A \wedge Y\ x \in B) = \mathcal{P}(x \text{ in } M.\ X\ x \in A) * \mathcal{P}(x \text{ in } M.\ Y\ x \in B)$
proof −
have $\mathcal{P}(x \text{ in } M.\ (X\ x) \in A \wedge (Y\ x) \in B) = prob\ ((\lambda x.\ (X\ x,\ Y\ x)) -` (A \times B) \cap space\ M)$
by (*auto intro*!: *arg-cong*[**where** *f*= *prob*])
also have ...= $prob\ (X -` A \cap space\ M) * prob\ (Y -` B \cap space\ M)$
by (*auto intro*!: *indep-varD*[**where** *Ma*=*N* **and** *Mb*=*N*])
also have ... = $\mathcal{P}(x \text{ in } M.\ X\ x \in A) * \mathcal{P}(x \text{ in } M.\ Y\ x \in B)$
by (*auto intro*!: *arg-cong2*[**where** *f*= (∗)] *arg-cong*[**where** *f*= *prob*])
finally show *?thesis* **.**
qed

lemma (**in** *prob-space*)
assumes *indep-var S X T Y*
shows *indep-var-rv1*: *random-variable S X*
and *indep-var-rv2*: *random-variable T Y*
proof −
have $\forall i \in UNIV.$ *random-variable* (*case-bool S T i*) (*case-bool X Y i*)
using *assms* **unfolding** *indep-var-def indep-vars-def* **by** *auto*
then show *random-variable S X random-variable T Y*
unfolding *UNIV-bool* **by** *auto*
qed

lemma (**in** *prob-space*) *indep-var-distribution-eq*:
indep-var S X T Y ⟷ *random-variable S X* ∧ *random-variable T Y* ∧
$distr\ M\ S\ X \bigotimes_M distr\ M\ T\ Y = distr\ M\ (S \bigotimes_M T)\ (\lambda x.\ (X\ x,\ Y\ x))$ (**is** - ⟷ - ∧ - ∧ $?S \bigotimes_M ?T = ?J$)
proof *safe*
assume *indep-var S X T Y*
then show *rvs*: *random-variable S X random-variable T Y*
by (*blast dest*: *indep-var-rv1 indep-var-rv2*)+
then have *XY*: *random-variable* $(S \bigotimes_M T)\ (\lambda x.\ (X\ x,\ Y\ x))$
by (*rule measurable-Pair*)

interpret *X*: *prob-space ?S* **by** (*rule prob-space-distr*) *fact*
interpret *Y*: *prob-space ?T* **by** (*rule prob-space-distr*) *fact*
interpret *XY*: *pair-prob-space ?S ?T* **..**
show $?S \bigotimes_M ?T = ?J$
proof (*rule pair-measure-eqI*)
show *sigma-finite-measure ?S* **..**
show *sigma-finite-measure ?T* **..**

fix *A B* **assume** *A*: *A* ∈ *sets ?S* **and** *B*: *B* ∈ *sets ?T*
have *emeasure ?J* $(A \times B) = emeasure\ M\ ((\lambda x.\ (X\ x,\ Y\ x)) -` (A \times B) \cap space\ M)$
using *A B* **by** (*intro emeasure-distr*[*OF XY*]) *auto*
also have ... = $emeasure\ M\ (X -` A \cap space\ M) * emeasure\ M\ (Y -` B \cap space\ M)$

```
      using indep-varD[OF ‹indep-var S X T Y›, of A B] A B
      by (simp add: emeasure-eq-measure measure-nonneg ennreal-mult)
    also have ... = emeasure ?S A * emeasure ?T B
      using rvs A B by (simp add: emeasure-distr)
    finally show emeasure ?S A * emeasure ?T B = emeasure ?J (A × B) by
simp
  qed simp
next
  assume rvs: random-variable S X random-variable T Y
  then have XY: random-variable (S ⨂M T) (λx. (X x, Y x))
    by (rule measurable-Pair)

  let ?S = distr M S X and ?T = distr M T Y
  interpret X: prob-space ?S by (rule prob-space-distr) fact
  interpret Y: prob-space ?T by (rule prob-space-distr) fact
  interpret XY: pair-prob-space ?S ?T ..

  assume ?S ⨂M ?T = ?J

  { fix S and X
    have Int-stable {X -‘ A ∩ space M |A. A ∈ sets S}
    proof (safe intro!: Int-stableI)
      fix A B assume A ∈ sets S B ∈ sets S
      then show ∃ C. (X -‘ A ∩ space M) ∩ (X -‘ B ∩ space M) = (X -‘ C ∩
space M) ∧ C ∈ sets S
        by (intro exI[of - A ∩ B]) auto
    qed }
  note Int-stable = this

  show indep-var S X T Y unfolding indep-var-eq
  proof (intro conjI indep-set-sigma-sets Int-stable rvs)
    show indep-set {X -‘ A ∩ space M |A. A ∈ sets S} {Y -‘ A ∩ space M |A.
A ∈ sets T}
    proof (safe intro!: indep-setI)
      { fix A assume A ∈ sets S then show X -‘ A ∩ space M ∈ sets M
        using ‹X ∈ measurable M S› by (auto intro: measurable-sets) }
      { fix A assume A ∈ sets T then show Y -‘ A ∩ space M ∈ sets M
        using ‹Y ∈ measurable M T› by (auto intro: measurable-sets) }
    next
      fix A B assume ab: A ∈ sets S B ∈ sets T
      then have prob ((X -‘ A ∩ space M) ∩ (Y -‘ B ∩ space M)) = emeasure
?J (A × B)
        using XY by (auto simp add: emeasure-distr emeasure-eq-measure mea-
sure-nonneg intro!: arg-cong[where f=prob])
      also have ... = emeasure (?S ⨂M ?T) (A × B)
        unfolding ‹?S ⨂M ?T = ?J› ..
      also have ... = emeasure ?S A * emeasure ?T B
        using ab by (simp add: Y.emeasure-pair-measure-Times)
      finally show prob ((X -‘ A ∩ space M) ∩ (Y -‘ B ∩ space M)) =
```

prob (*X* −‘ *A* ∩ *space M*) ∗ *prob* (*Y* −‘ *B* ∩ *space M*)
using *rvs ab* **by** (*simp add*: *emeasure-eq-measure emeasure-distr measure-nonneg ennreal-mult*[*symmetric*])
qed
qed
qed

lemma (**in** *prob-space*) *distributed-joint-indep*:
assumes *S*: *sigma-finite-measure S* **and** *T*: *sigma-finite-measure T*
assumes *X*: *distributed M S X Px* **and** *Y*: *distributed M T Y Py*
assumes *indep*: *indep-var S X T Y*
shows *distributed M* (*S* ⨂$_M$ *T*) (λ*x*. (*X x*, *Y x*)) (λ(*x*, *y*). *Px x* ∗ *Py y*)
using *indep-var-distribution-eq*[*of S X T Y*] *indep*
by (*intro distributed-joint-indep'*[*OF S T X Y*]) *auto*

lemma (**in** *prob-space*) *indep-vars-nn-integral*:
assumes *I*: *finite I indep-vars* (λ-. *borel*) *X I* ⋀*i ω*. *i* ∈ *I* ⟹ 0 ≤ *X i ω*
shows (∫$^+ω$. (∏ *i*∈*I*. *X i ω*) ∂*M*) = (∏ *i*∈*I*. ∫$^+ω$. *X i ω* ∂*M*)
proof *cases*
assume *I* ≠ {}
define *Y* **where** [*abs-def*]: *Y i ω* = (*if i* ∈ *I then X i ω else* 0) **for** *i ω*
{ fix *i* **have** *i* ∈ *I* ⟹ *random-variable borel* (*X i*)
using *I*(2) **by** (*cases i*∈*I*) (*auto simp*: *indep-vars-def*) **}**
note *rv-X* = *this*

{ fix *i* **have** *random-variable borel* (*Y i*)
using *I*(2) **by** (*cases i*∈*I*) (*auto simp*: *Y-def rv-X*) **}**
note *rv-Y* = *this*[*measurable*]

interpret *Y*: *prob-space distr M borel* (*Y i*) **for** *i*
using *I*(2) **by** (*cases i* ∈ *I*) (*auto intro*!: *prob-space-distr simp*: *indep-vars-def prob-space-return*)
interpret *product-sigma-finite* λ*i*. *distr M borel* (*Y i*)
..

have *indep-Y*: *indep-vars* (λ*i*. *borel*) *Y I*
by (*rule indep-vars-cong*[*THEN iffD1*, *OF - - - I*(2)]) (*auto simp*: *Y-def*)

have (∫$^+ω$. (∏ *i*∈*I*. *X i ω*) ∂*M*) = (∫$^+ω$. (∏ *i*∈*I*. *Y i ω*) ∂*M*)
using *I*(3) **by** (*auto intro*!: *nn-integral-cong prod.cong simp add*: *Y-def max-def*)
also have ... = (∫$^+ω$. (∏ *i*∈*I*. *ω i*) ∂*distr M* (*Pi$_M$ I* (λ*i*. *borel*)) (λ*x*. λ*i*∈*I*. *Y i x*))
by (*subst nn-integral-distr*) *auto*
also have ... = (∫$^+ω$. (∏ *i*∈*I*. *ω i*) ∂*Pi$_M$ I* (λ*i*. *distr M borel* (*Y i*)))
unfolding *indep-vars-iff-distr-eq-PiM*[*THEN iffD1*, *OF* ‹*I* ≠ {}› *rv-Y indep-Y*]
..
also have ... = (∏ *i*∈*I*. (∫$^+ω$. *ω* ∂*distr M borel* (*Y i*)))
by (*rule product-nn-integral-prod*) (*auto intro*: ‹*finite I*›)
also have ... = (∏ *i*∈*I*. ∫$^+ω$. *X i ω* ∂*M*)

by (*intro prod.cong nn-integral-cong*) (*auto simp*: *nn-integral-distr Y-def rv-X*)
finally show *?thesis* **.**
qed (*simp add*: *emeasure-space-1*)

lemma (**in** *prob-space*)
fixes $X :: 'i \Rightarrow 'a \Rightarrow 'b$::{*real-normed-field*, *banach*, *second-countable-topology*}
assumes *I*: *finite I indep-vars* (λ-. *borel*) *X I* ⋀*i*. $i \in I \Longrightarrow$ *integrable M* (*X i*)
shows *indep-vars-lebesgue-integral*: $(\int \omega.\ (\prod i \in I.\ X\ i\ \omega)\ \partial M) = (\prod i \in I.\ \int \omega.\ X\ i\ \omega\ \partial M)$ (**is** *?eq*)
and *indep-vars-integrable*: *integrable M* $(\lambda\omega.\ (\prod i \in I.\ X\ i\ \omega))$ (**is** *?int*)
proof (*induct rule*: *case-split*)
assume $I \neq \{\}$
define *Y* **where** [*abs-def*]: *Y i* ω = (*if i* ∈ *I then X i* ω *else 0*) **for** *i* ω
{ fix *i* **have** $i \in I \Longrightarrow$ *random-variable borel* (*X i*)
using *I*(*2*) **by** (*cases i*∈*I*) (*auto simp*: *indep-vars-def*) **}**
note *rv-X* = *this*[*measurable*]

{ fix *i* **have** *random-variable borel* (*Y i*)
using *I*(*2*) **by** (*cases i*∈*I*) (*auto simp*: *Y-def rv-X*) **}**
note *rv-Y* = *this*[*measurable*]

{ fix *i* **have** *integrable M* (*Y i*)
using *I*(*3*) **by** (*cases i*∈*I*) (*auto simp*: *Y-def*) **}**
note *int-Y* = *this*

interpret *Y*: *prob-space distr M borel* (*Y i*) **for** *i*
using *I*(*2*) **by** (*cases i* ∈ *I*) (*auto intro*!: *prob-space-distr simp*: *indep-vars-def prob-space-return*)
interpret *product-sigma-finite* λ*i*. *distr M borel* (*Y i*)
..

have *indep-Y*: *indep-vars* (λ*i*. *borel*) *Y I*
by (*rule indep-vars-cong*[*THEN iffD1*, *OF - - - I*(*2*)]) (*auto simp*: *Y-def*)

have $(\int \omega.\ (\prod i \in I.\ X\ i\ \omega)\ \partial M) = (\int \omega.\ (\prod i \in I.\ Y\ i\ \omega)\ \partial M)$
using *I*(*3*) **by** (*simp add*: *Y-def*)
also have ... = $(\int \omega.\ (\prod i \in I.\ \omega\ i)\ \partial distr\ M\ (Pi_M\ I\ (\lambda i.\ borel))\ (\lambda x.\ \lambda i \in I.\ Y\ i\ x))$
by (*subst integral-distr*) *auto*
also have ... = $(\int \omega.\ (\prod i \in I.\ \omega\ i)\ \partial Pi_M\ I\ (\lambda i.\ distr\ M\ borel\ (Y\ i)))$
unfolding *indep-vars-iff-distr-eq-PiM*[*THEN iffD1*, *OF* ‹*I* ≠ {}› *rv-Y indep-Y*]
..
also have ... = $(\prod i \in I.\ (\int \omega.\ \omega\ \partial distr\ M\ borel\ (Y\ i)))$
by (*rule product-integral-prod*) (*auto intro*: ‹*finite I*› *simp*: *integrable-distr-eq int-Y*)
also have ... = $(\prod i \in I.\ \int \omega.\ X\ i\ \omega\ \partial M)$
by (*intro prod.cong integral-cong*)
(*auto simp*: *integral-distr Y-def rv-X*)
finally show *?eq* **.**

have *integrable* (*distr M* (*Pi$_M$ I* (λ*i*. *borel*)) (λ*x*. λ*i*∈*I*. *Y i x*)) (λω. (∏ *i*∈*I*. ω *i*))
unfolding *indep-vars-iff-distr-eq-PiM*[*THEN iffD1*, *OF* ‹*I* ≠ {}› *rv-Y indep-Y*]
by (*intro product-integrable-prod*[*OF* ‹*finite I*›])
(*simp add*: *integrable-distr-eq int-Y*)
then show *?int*
by (*simp add*: *integrable-distr-eq Y-def*)
qed (*simp-all add*: *prob-space*)

lemma (**in** *prob-space*)
fixes *X1 X2* :: *'a* ⇒ *'b*::{*real-normed-field*, *banach*, *second-countable-topology*}
assumes *indep-var borel X1 borel X2 integrable M X1 integrable M X2*
shows *indep-var-lebesgue-integral*: (∫ ω. *X1* ω ∗ *X2* ω ∂*M*) = (∫ ω. *X1* ω ∂*M*) ∗ (∫ ω. *X2* ω ∂*M*) (**is** *?eq*)
and *indep-var-integrable*: *integrable M* (λω. *X1* ω ∗ *X2* ω) (**is** *?int*)
unfolding *indep-var-def*
proof −
have ∗: (λω. *X1* ω ∗ *X2* ω) = (λω. ∏ *i*∈*UNIV*. (*case-bool X1 X2 i* ω))
by (*simp add*: *UNIV-bool mult.commute*)
have ∗∗: (λ -. *borel*) = *case-bool borel borel*
by (*rule ext*, *metis* (*full-types*) *bool.simps*(*3*) *bool.simps*(*4*))
show *?eq*
apply (*subst* ∗)
apply (*subst indep-vars-lebesgue-integral*)
apply (*auto*)
apply (*subst* ∗∗, *subst indep-var-def* [*symmetric*], *rule assms*)
apply (*simp split*: *bool.split add*: *assms*)
by (*simp add*: *UNIV-bool mult.commute*)
show *?int*
apply (*subst* ∗)
apply (*rule indep-vars-integrable*)
apply *auto*
apply (*subst* ∗∗, *subst indep-var-def* [*symmetric*], *rule assms*)
by (*simp split*: *bool.split add*: *assms*)
qed

end

11 Convolution Measure

theory *Convolution*
imports *Independent-Family*
begin

lemma (**in** *finite-measure*) *sigma-finite-measure*: *sigma-finite-measure M*
..

definition *convolution* :: (*'a* :: *ordered-euclidean-space*) *measure* ⇒ *'a measure* ⇒

'a measure (**infix** ⋆ *50*) **where**
convolution M N = *distr* (*M* ⨂$_M$ *N*) *borel* (λ(*x*, *y*). *x* + *y*)

lemma
shows *space-convolution*[*simp*]: *space* (*convolution M N*) = *space borel*
and *sets-convolution*[*simp*]: *sets* (*convolution M N*) = *sets borel*
and *measurable-convolution1*[*simp*]: *measurable A* (*convolution M N*) = *measurable A borel*
and *measurable-convolution2*[*simp*]: *measurable* (*convolution M N*) *B* = *measurable borel B*
by (*simp-all add*: *convolution-def*)

lemma *nn-integral-convolution*:
assumes *finite-measure M finite-measure N*
assumes [*measurable-cong*]: *sets N* = *sets borel sets M* = *sets borel*
assumes [*measurable*]: *f* ∈ *borel-measurable borel*
shows (∫$^+$*x*. *f x* ∂*convolution M N*) = (∫$^+$*x*. ∫$^+$*y*. *f* (*x* + *y*) ∂*N* ∂*M*)
proof −
interpret *M*: *finite-measure M* **by** *fact*
interpret *N*: *finite-measure N* **by** *fact*
interpret *pair-sigma-finite M N* **..**
show *?thesis*
unfolding *convolution-def*
by (*simp add*: *nn-integral-distr N*.*nn-integral-fst*[*symmetric*])
qed

lemma *convolution-emeasure*:
assumes *A* ∈ *sets borel finite-measure M finite-measure N*
assumes [*simp*]: *sets N* = *sets borel sets M* = *sets borel*
assumes [*simp*]: *space M* = *space N space N* = *space borel*
shows *emeasure* (*M* ⋆ *N*) *A* = ∫$^+$*x*. (*emeasure N* {*a*. *a* + *x* ∈ *A*}) ∂*M*
using *assms* **by** (*auto intro*!: *nn-integral-cong simp del*: *nn-integral-indicator simp*: *nn-integral-convolution*
nn-integral-indicator [*symmetric*] *ac-simps split*:*split-indicator*)

lemma *convolution-emeasure'*:
assumes [*simp*]:*A* ∈ *sets borel*
assumes [*simp*]: *finite-measure M finite-measure N*
assumes [*simp*]: *sets N* = *sets borel sets M* = *sets borel*
shows *emeasure* (*M* ⋆ *N*) *A* = ∫$^+$*x*. ∫$^+$*y*. (*indicator A* (*x* + *y*)) ∂*N* ∂*M*
by (*auto simp del*: *nn-integral-indicator simp*: *nn-integral-convolution*
nn-integral-indicator[*symmetric*] *borel-measurable-indicator*)

lemma *convolution-finite*:
assumes [*simp*]: *finite-measure M finite-measure N*
assumes [*measurable-cong*]: *sets N* = *sets borel sets M* = *sets borel*
shows *finite-measure* (*M* ⋆ *N*)
unfolding *convolution-def*
by (*intro finite-measure-pair-measure finite-measure.finite-measure-distr*) *auto*

lemma *convolution-emeasure-3*:
assumes [*simp*, *measurable*]: $A \in$ *sets borel*
assumes [*simp*]: *finite-measure M finite-measure N finite-measure L*
assumes [*simp*]: *sets N = sets borel sets M = sets borel sets L = sets borel*
shows *emeasure* $(L \star (M \star N)) A = \int^+ x. \int^+ y. \int^+ z.$ *indicator A* $(x + y + z)$ *∂N ∂M ∂L*
apply (*subst nn-integral-indicator*[*symmetric*], *simp*)
apply (*subst nn-integral-convolution*,
auto intro!: *borel-measurable-indicator borel-measurable-indicator′ convolution-finite*)+
by (*rule nn-integral-cong*)+ (*auto simp*: *semigroup-add-class.add.assoc*)

lemma *convolution-emeasure-3′*:
assumes [*simp*, *measurable*]:$A \in$ *sets borel*
assumes [*simp*]: *finite-measure M finite-measure N finite-measure L*
assumes [*measurable-cong*, *simp*]: *sets N = sets borel sets M = sets borel sets L = sets borel*
shows *emeasure* $((L \star M) \star N) A = \int^+ x. \int^+ y. \int^+ z.$ *indicator A* $(x + y + z)$ *∂N ∂M ∂L*
apply (*subst nn-integral-indicator*[*symmetric*], *simp*)+
apply (*subst nn-integral-convolution*)
apply (*simp-all add*: *convolution-finite*)
apply (*subst nn-integral-convolution*)
apply (*simp-all add*: *finite-measure.sigma-finite-measure sigma-finite-measure.borel-measurable-nn-integral*)
done

lemma *convolution-commutative*:
assumes [*simp*]: *finite-measure M finite-measure N*
assumes [*measurable-cong*, *simp*]: *sets N = sets borel sets M = sets borel*
shows $(M \star N) = (N \star M)$
proof (*rule measure-eqI*)
interpret *M*: *finite-measure M* **by** *fact*
interpret *N*: *finite-measure N* **by** *fact*
interpret *pair-sigma-finite M N* **..**

show *sets* $(M \star N) =$ *sets* $(N \star M)$ **by** *simp*

fix *A* **assume** $A \in$ *sets* $(M \star N)$
then have *1*[*measurable*]:$A \in$ *sets borel* **by** *simp*
have *emeasure* $(M \star N) A = \int^+ x. \int^+ y.$ *indicator A* $(x + y)$ *∂N ∂M* **by** (*auto intro*!: *convolution-emeasure′*)
also have ... $= \int^+ x. \int^+ y. (\lambda(x,y).$ *indicator A* $(x + y))$ (x, y) *∂N ∂M* **by** (*auto intro*!: *nn-integral-cong*)
also have ... $= \int^+ y. \int^+ x. (\lambda(x,y).$ *indicator A* $(x + y))$ (x, y) *∂M ∂N* **by** (*rule Fubini*[*symmetric*]) *simp*
also have ... = *emeasure* $(N \star M)$ *A* **by** (*auto intro*!: *nn-integral-cong simp*: *add.commute convolution-emeasure′*)
finally show *emeasure* $(M \star N)$ *A* = *emeasure* $(N \star M)$ *A* **by** *simp*

qed

lemma *convolution-associative*:
assumes [*simp*]: *finite-measure M finite-measure N finite-measure L*
assumes [*simp*]: *sets N = sets borel sets M = sets borel sets L = sets borel*
shows $(L \star (M \star N)) = ((L \star M) \star N)$
by (*auto intro*!: *measure-eqI simp*: *convolution-emeasure-3 convolution-emeasure-3'*)

lemma (**in** *prob-space*) *sum-indep-random-variable*:
assumes *ind*: *indep-var borel X borel Y*
assumes [*simp*, *measurable*]: *random-variable borel X*
assumes [*simp*, *measurable*]: *random-variable borel Y*
shows *distr M borel* $(\lambda x.\ X\ x + Y\ x)$ = *convolution* (*distr M borel X*) (*distr M borel Y*)
using *ind* **unfolding** *indep-var-distribution-eq convolution-def*
by (*auto simp*: *distr-distr intro*!:*arg-cong*[**where** *f* = *distr M borel*])

lemma (**in** *prob-space*) *sum-indep-random-variable-lborel*:
assumes *ind*: *indep-var borel X borel Y*
assumes [*simp*, *measurable*]: *random-variable lborel X*
assumes [*simp*, *measurable*]:*random-variable lborel Y*
shows *distr M lborel* $(\lambda x.\ X\ x + Y\ x)$ = *convolution* (*distr M lborel X*) (*distr M lborel Y*)
using *ind* **unfolding** *indep-var-distribution-eq convolution-def*
by (*auto simp*: *distr-distr o-def intro*!: *arg-cong*[**where** *f* = *distr M borel*] *cong*: *distr-cong*)

lemma *convolution-density*:
fixes *f g* :: *real* ⇒ *ennreal*
assumes [*measurable*]: *f* ∈ *borel-measurable borel g* ∈ *borel-measurable borel*
assumes [*simp*]:*finite-measure* (*density lborel f*) *finite-measure* (*density lborel g*)
shows *density lborel f* ⋆ *density lborel g* = *density lborel* $(\lambda x.\ \int^+ y.\ f\ (x - y) * g\ y\ \partial lborel)$
(**is** *?l* = *?r*)
proof (*intro measure-eqI*)
fix *A* **assume** *A* ∈ *sets ?l*
then have [*measurable*]: *A* ∈ *sets borel*
by *simp*

have $(\int^+ x.\ f\ x * (\int^+ y.\ g\ y * indicator\ A\ (x + y)\ \partial lborel)\ \partial lborel) =$
$(\int^+ x.\ (\int^+ y.\ g\ y * (f\ x * indicator\ A\ (x + y))\ \partial lborel)\ \partial lborel)$
proof (*intro nn-integral-cong-AE*, *eventually-elim*)
fix *x*
have $f\ x * (\int^+ y.\ g\ y * indicator\ A\ (x + y)\ \partial lborel) =$
$(\int^+ y.\ f\ x * (g\ y * indicator\ A\ (x + y))\ \partial lborel)$
by (*intro nn-integral-cmult*[*symmetric*]) *auto*
then show $f\ x * (\int^+ y.\ g\ y * indicator\ A\ (x + y)\ \partial lborel) =$
$(\int^+ y.\ g\ y * (f\ x * indicator\ A\ (x + y))\ \partial lborel)$
by (*simp add*: *ac-simps*)

qed
also have ... = $(\int^+ y.\ (\int^+ x.\ g\ y * (f\ x * \mathit{indicator}\ A\ (x + y))\ \partial \mathit{lborel})\ \partial \mathit{lborel})$
by (*intro lborel-pair.Fubini'*) *simp*
also have ... = $(\int^+ y.\ (\int^+ x.\ f\ (x - y) * g\ y * \mathit{indicator}\ A\ x\ \partial \mathit{lborel})\ \partial \mathit{lborel})$
proof (*intro nn-integral-cong-AE*, *eventually-elim*)
fix *y*
have $(\int^+ x.\ g\ y * (f\ x * \mathit{indicator}\ A\ (x + y))\ \partial \mathit{lborel}) =$
$g\ y * (\int^+ x.\ f\ x * \mathit{indicator}\ A\ (x + y)\ \partial \mathit{lborel})$
by (*intro nn-integral-cmult*) *auto*
also have ... = $g\ y * (\int^+ x.\ f\ (x - y) * \mathit{indicator}\ A\ x\ \partial \mathit{lborel})$
by (*subst nn-integral-real-affine*[**where** *c=1* **and** *t=−y*])
(*auto simp add*: *one-ennreal-def*[*symmetric*])
also have ... = $(\int^+ x.\ g\ y * (f\ (x - y) * \mathit{indicator}\ A\ x)\ \partial \mathit{lborel})$
by (*intro nn-integral-cmult*[*symmetric*]) *auto*
finally show $(\int^+ x.\ g\ y * (f\ x * \mathit{indicator}\ A\ (x + y))\ \partial \mathit{lborel}) =$
$(\int^+ x.\ f\ (x - y) * g\ y * \mathit{indicator}\ A\ x\ \partial \mathit{lborel})$
by (*simp add*: *ac-simps*)
qed
also have ... = $(\int^+ x.\ (\int^+ y.\ f\ (x - y) * g\ y * \mathit{indicator}\ A\ x\ \partial \mathit{lborel})\ \partial \mathit{lborel})$
by (*intro lborel-pair.Fubini'*) *simp*
finally show *emeasure ?l A = emeasure ?r A*
by (*auto simp*: *convolution-emeasure' nn-integral-density emeasure-density nn-integral-multc*)
qed *simp*

lemma (**in** *prob-space*) *distributed-finite-measure-density*:
distributed M N X f $\Longrightarrow$ *finite-measure* (*density N f*)
using *finite-measure-distr*[*of X N*] *distributed-distr-eq-density*[*of M N X f*] **by** *simp*

lemma (**in** *prob-space*) *distributed-convolution*:
fixes *f* :: *real* ⇒ -
fixes *g* :: *real* ⇒ -
assumes *indep*: *indep-var borel X borel Y*
assumes *X*: *distributed M lborel X f*
assumes *Y*: *distributed M lborel Y g*
shows *distributed M lborel* $(\lambda x.\ X\ x + Y\ x)$ $(\lambda x.\ \int^+ y.\ f\ (x - y) * g\ y\ \partial \mathit{lborel})$
unfolding *distributed-def*
proof *safe*
have *fg*[*measurable*]: *f* ∈ *borel-measurable borel g* ∈ *borel-measurable borel*
using *distributed-borel-measurable*[*OF X*] *distributed-borel-measurable*[*OF Y*]
by *simp-all*

show $(\lambda x.\ \int^+ xa.\ f\ (x - xa) * g\ xa\ \partial \mathit{lborel}) \in \mathit{borel\text{-}measurable\ lborel}$
by *measurable*

have *distr M borel* $(\lambda x.\ X\ x + Y\ x) = (\mathit{distr\ M\ borel\ X} \star \mathit{distr\ M\ borel\ Y})$
using *distributed-measurable*[*OF X*] *distributed-measurable*[*OF Y*]

by (*intro sum-indep-random-variable*) (*auto simp*: *indep*)
also have ... = (*density lborel f* ⋆ *density lborel g*)
using *distributed-distr-eq-density*[*OF X*] *distributed-distr-eq-density*[*OF Y*]
by (*simp cong*: *distr-cong*)
also have ... = *density lborel* (λ*x*. ∫⁺ *y*. *f* (*x* − *y*) ∗ *g y* ∂*lborel*)
proof (*rule convolution-density*)
show *finite-measure* (*density lborel f*)
using *X* **by** (*rule distributed-finite-measure-density*)
show *finite-measure* (*density lborel g*)
using *Y* **by** (*rule distributed-finite-measure-density*)
qed *fact*+
finally show *distr M lborel* (λ*x*. *X x* + *Y x*) = *density lborel* (λ*x*. ∫⁺ *y*. *f* (*x* − *y*) ∗ *g y* ∂*lborel*)
by (*simp cong*: *distr-cong*)
show *random-variable lborel* (λ*x*. *X x* + *Y x*)
using *distributed-measurable*[*OF X*] *distributed-measurable*[*OF Y*] **by** *simp*
qed

lemma *prob-space-convolution-density*:
fixes *f*:: *real* ⇒ -
fixes *g*:: *real* ⇒ -
assumes [*measurable*]: *f*∈ *borel-measurable borel*
assumes [*measurable*]: *g*∈ *borel-measurable borel*
assumes *gt-0*[*simp*]: ⋀*x*. *0* ≤ *f x* ⋀*x*. *0* ≤ *g x*
assumes *prob-space* (*density lborel f*) (**is** *prob-space ?F*)
assumes *prob-space* (*density lborel g*) (**is** *prob-space ?G*)
shows *prob-space* (*density lborel* (λ*x*.∫⁺*y*. *f* (*x* − *y*) ∗ *g y* ∂*lborel*)) (**is** *prob-space ?D*)
proof (*subst convolution-density*[*symmetric*])
interpret *F*: *prob-space ?F* **by** *fact*
show *finite-measure ?F* **by** *unfold-locales*
interpret *G*: *prob-space ?G* **by** *fact*
show *finite-measure ?G* **by** *unfold-locales*
interpret *FG*: *pair-prob-space ?F ?G* **..**

show *prob-space* (*density lborel f* ⋆ *density lborel g*)
unfolding *convolution-def* **by** (*rule FG.prob-space-distr*) *simp*
qed *simp-all*

end

12 Information theory

theory *Information*
imports
Independent-Family
begin

lemma *log-le*: *1* < *a* ⟹ *0* < *x* ⟹ *x* ≤ *y* ⟹ *log a x* ≤ *log a y*

by (*subst log-le-cancel-iff*) *auto*

lemma *log-less*: $1 < a \Longrightarrow 0 < x \Longrightarrow x < y \Longrightarrow log\ a\ x < log\ a\ y$
by (*subst log-less-cancel-iff*) *auto*

lemma *sum-cartesian-product'*:
$(\sum x{\in}A \times B.\ f\ x) = (\sum x{\in}A.\ sum\ (\lambda y.\ f\ (x,\ y))\ B)$
unfolding *sum.cartesian-product* **by** *simp*

lemma *split-pairs*:
$((A,\ B) = X) \longleftrightarrow (fst\ X = A \wedge snd\ X = B)$ **and**
$(X = (A,\ B)) \longleftrightarrow (fst\ X = A \wedge snd\ X = B)$ **by** *auto*

12.1 Information theory

locale *information-space* = *prob-space* +
fixes b :: *real* **assumes** *b-gt-1*: $1 < b$

context *information-space*
begin

Introduce some simplification rules for logarithm of base b.

lemma *log-neg-const*:
assumes $x \leq 0$
shows $log\ b\ x = log\ b\ 0$
proof −
{ fix u :: *real*
have $x \leq 0$ **by** *fact*
also have $0 < exp\ u$
using *exp-gt-zero* **.**
finally have $exp\ u \neq x$
by *auto* **}**
then show $log\ b\ x = log\ b\ 0$
by (*simp add*: *log-def ln-real-def*)
qed

lemma *log-mult-eq*:
$log\ b\ (A * B) = (if\ 0 < A * B\ then\ log\ b\ |A| + log\ b\ |B|\ else\ log\ b\ 0)$
using *log-mult*[*of b* $|A|$ $|B|$] *b-gt-1 log-neg-const*[*of* $A * B$]
by (*auto simp*: *zero-less-mult-iff mult-le-0-iff*)

lemma *log-inverse-eq*:
$log\ b\ (inverse\ B) = (if\ 0 < B\ then\ -\ log\ b\ B\ else\ log\ b\ 0)$
using *log-inverse*[*of b B*] *log-neg-const*[*of inverse B*] *b-gt-1* **by** *simp*

lemma *log-divide-eq*:
$log\ b\ (A\ /\ B) = (if\ 0 < A * B\ then\ log\ b\ |A| - log\ b\ |B|\ else\ log\ b\ 0)$
unfolding *divide-inverse log-mult-eq log-inverse-eq abs-inverse*
by (*auto simp*: *zero-less-mult-iff mult-le-0-iff*)

lemmas *log-simps* = *log-mult-eq log-inverse-eq log-divide-eq*

end

12.2 Kullback−Leibler divergence

The Kullback−Leibler divergence is also known as relative entropy or Kullback−Leibler distance.

definition
entropy-density b M N = *log b* ∘ *enn2real* ∘ *RN-deriv M N*

definition
KL-divergence b M N = *integral*^{L} *N* (*entropy-density b M N*)

lemma *measurable-entropy-density*[*measurable*]: *entropy-density b M N* ∈ *borel-measurable M*
unfolding *entropy-density-def* **by** *auto*

lemma (**in** *sigma-finite-measure*) *KL-density*:
fixes *f* :: *'a* ⇒ *real*
assumes *1* < *b*
assumes *f*[*measurable*]: *f* ∈ *borel-measurable M* **and** *nn*: *AE x in M*. *0* ≤ *f x*
shows *KL-divergence b M* (*density M f*) = (∫ *x*. *f x* ∗ *log b* (*f x*) ∂*M*)
unfolding *KL-divergence-def*
proof (*subst integral-real-density*)
show [*measurable*]: *entropy-density b M* (*density M* (λ*x*. *ennreal* (*f x*))) ∈ *borel-measurable M*
using *f*
by (*auto simp*: *comp-def entropy-density-def*)
have *density M* (*RN-deriv M* (*density M f*)) = *density M f*
using *f nn* **by** (*intro density-RN-deriv-density*) *auto*
then have *eq*: *AE x in M*. *RN-deriv M* (*density M f*) *x* = *f x*
using *f nn* **by** (*intro density-unique*) *auto*
show (∫ *x*. *f x* ∗ *entropy-density b M* (*density M* (λ*x*. *ennreal* (*f x*))) *x* ∂*M*) = (∫ *x*. *f x* ∗ *log b* (*f x*) ∂*M*)
apply (*intro integral-cong-AE*)
apply *measurable*
using *eq nn*
apply *eventually-elim*
apply (*auto simp*: *entropy-density-def*)
done
qed *fact*+

lemma (**in** *sigma-finite-measure*) *KL-density-density*:
fixes *f g* :: *'a* ⇒ *real*
assumes *1* < *b*
assumes *f*: *f* ∈ *borel-measurable M AE x in M*. *0* ≤ *f x*
assumes *g*: *g* ∈ *borel-measurable M AE x in M*. *0* ≤ *g x*

assumes *ac*: *AE x in M. f x = 0 ⟶ g x = 0*
shows *KL-divergence b (density M f) (density M g) = (∫ x. g x * log b (g x / f x) ∂M)*
proof −
interpret *Mf*: *sigma-finite-measure density M f*
using *f* **by** (*subst sigma-finite-iff-density-finite*) *auto*
have *KL-divergence b (density M f) (density M g) =*
KL-divergence b (density M f) (density (density M f) (λx. g x / f x))
using *f g ac* **by** (*subst density-density-divide*) *simp-all*
also have *... = (∫ x. (g x / f x) * log b (g x / f x) ∂density M f)*
using *f g ‹1 < b›* **by** (*intro Mf.KL-density*) (*auto simp*: *AE-density*)
also have *... = (∫ x. g x * log b (g x / f x) ∂M)*
using *ac f g ‹1 < b›* **by** (*subst integral-density*) (*auto intro*!: *integral-cong-AE*)
finally show *?thesis* **.**
qed

lemma (**in** *information-space*) *KL-gt-0*:
fixes *D* :: *'a ⇒ real*
assumes *prob-space* (*density M D*)
assumes *D*: *D ∈ borel-measurable M AE x in M. 0 ≤ D x*
assumes *int*: *integrable M* (*λx. D x * log b* (*D x*))
assumes *A*: *density M D ≠ M*
shows *0 < KL-divergence b M* (*density M D*)
proof −
interpret *N*: *prob-space density M D* **by** *fact*

obtain *A* **where** *A ∈ sets M emeasure* (*density M D*) *A ≠ emeasure M A*
using *measure-eqI*[*of density M D M*] *‹density M D ≠ M›* **by** *auto*

let *?D-set* = {*x*∈*space M. D x ≠ 0*}
have [*simp*, *intro*]: *?D-set ∈ sets M*
using *D* **by** *auto*

have *D-neg*: (∫⁺ *x. ennreal* (− *D x*) *∂M*) = *0*
using *D* **by** (*subst nn-integral-0-iff-AE*) (*auto simp*: *ennreal-neg*)

have (∫⁺ *x. ennreal* (*D x*) *∂M*) = *emeasure* (*density M D*) (*space M*)
using *D* **by** (*simp add*: *emeasure-density cong*: *nn-integral-cong*)
then have *D-pos*: (∫⁺ *x. ennreal* (*D x*) *∂M*) = *1*
using *N.emeasure-space-1* **by** *simp*

have *integrable M D*
using *D D-pos D-neg* **unfolding** *real-integrable-def real-lebesgue-integral-def* **by** *simp-all*
then have $integral^L$ *M D = 1*
using *D D-pos D-neg* **by** (*simp add*: *real-lebesgue-integral-def*)

have *0 ≤ 1 − measure M ?D-set*
using *prob-le-1* **by** (*auto simp*: *field-simps*)

```
  also have ... = (∫ x. D x − indicator ?D-set x ∂M)
    using ‹integrable M D› ‹integral^L M D = 1›
    by (simp add: emeasure-eq-measure)
  also have ... < (∫ x. D x * (ln b * log b (D x)) ∂M)
  proof (rule integral-less-AE)
    show integrable M (λx. D x − indicator ?D-set x)
      using ‹integrable M D› by (auto simp: less-top[symmetric])
  next
    from integrable-mult-left(1)[OF int, of ln b]
    show integrable M (λx. D x * (ln b * log b (D x)))
      by (simp add: ac-simps)
  next
    show emeasure M {x∈space M. D x ≠ 1 ∧ D x ≠ 0} ≠ 0
    proof
      assume eq-0: emeasure M {x∈space M. D x ≠ 1 ∧ D x ≠ 0} = 0
      then have disj: AE x in M. D x = 1 ∨ D x = 0
        using D(1) by (auto intro!: AE-I[OF subset-refl] sets.sets-Collect)

      have emeasure M {x∈space M. D x = 1} = (∫⁺ x. indicator {x∈space M.
D x = 1} x ∂M)
        using D(1) by auto
      also have ... = (∫⁺ x. ennreal (D x) ∂M)
      using disj by (auto intro!: nn-integral-cong-AE simp: indicator-def one-ennreal-def)
      finally have AE x in M. D x = 1
        using D D-pos by (intro AE-I-eq-1) auto
       then have (∫⁺x. indicator A x∂M) = (∫⁺x. ennreal (D x) * indicator A
x∂M)
        by (intro nn-integral-cong-AE) (auto simp: one-ennreal-def[symmetric])
      also have ... = density M D A
        using ‹A ∈ sets M› D by (simp add: emeasure-density)
        finally show False using ‹A ∈ sets M› ‹emeasure (density M D) A ≠
emeasure M A› by simp
    qed
    show {x∈space M. D x ≠ 1 ∧ D x ≠ 0} ∈ sets M
      using D(1) by (auto intro: sets.sets-Collect-conj)

    show AE t in M. t ∈ {x∈space M. D x ≠ 1 ∧ D x ≠ 0} ⟶
      D t − indicator ?D-set t ≠ D t * (ln b * log b (D t))
      using D(2)
    proof (eventually-elim, safe)
      fix t assume Dt: t ∈ space M D t ≠ 1 D t ≠ 0 0 ≤ D t
        and eq: D t − indicator ?D-set t = D t * (ln b * log b (D t))

      have D t − 1 = D t − indicator ?D-set t
        using Dt by simp
      also note eq
      also have D t * (ln b * log b (D t)) = − D t * ln (1 / D t)
        using b-gt-1 ‹D t ≠ 0› ‹0 ≤ D t›
        by (simp add: log-def ln-div less-le)
```

```
      finally have ln (1 / D t) = 1 / D t − 1
        using ‹D t ≠ 0› by (auto simp: field-simps)
      from ln-eq-minus-one[OF - this] ‹D t ≠ 0› ‹0 ≤ D t› ‹D t ≠ 1›
      show False by auto
    qed

    show AE t in M. D t − indicator ?D-set t ≤ D t * (ln b * log b (D t))
      using D(2) AE-space
    proof eventually-elim
      fix t assume t ∈ space M 0 ≤ D t
      show D t − indicator ?D-set t ≤ D t * (ln b * log b (D t))
      proof cases
        assume asm: D t ≠ 0
        then have 0 < D t using ‹0 ≤ D t› by auto
        then have 0 < 1 / D t by auto
        have D t − indicator ?D-set t ≤ − D t * (1 / D t − 1)
          using asm ‹t ∈ space M› by (simp add: field-simps)
        also have − D t * (1 / D t − 1) ≤ − D t * ln (1 / D t)
          using ln-le-minus-one ‹0 < 1 / D t› by (intro mult-left-mono-neg) auto
        also have ... = D t * (ln b * log b (D t))
          using ‹0 < D t› b-gt-1
          by (simp-all add: log-def ln-div)
        finally show ?thesis by simp
      qed simp
    qed
  qed
  also have ... = (∫ x. ln b * (D x * log b (D x)) ∂M)
    by (simp add: ac-simps)
  also have ... = ln b * (∫ x. D x * log b (D x) ∂M)
    using int by simp
  finally show ?thesis
    using b-gt-1 D by (subst KL-density) (auto simp: zero-less-mult-iff)
qed

lemma (in sigma-finite-measure) KL-same-eq-0: KL-divergence b M M = 0
proof −
  have AE x in M. 1 = RN-deriv M M x
  proof (rule RN-deriv-unique)
    show density M (λx. 1) = M
      apply (auto intro!: measure-eqI emeasure-density)
      apply (subst emeasure-density)
      apply auto
      done
  qed auto
  then have AE x in M. log b (enn2real (RN-deriv M M x)) = 0
    by (elim AE-mp) simp
  from integral-cong-AE[OF - - this]
  have integral^L M (entropy-density b M M) = 0
    by (simp add: entropy-density-def comp-def)
```

```
  then show KL-divergence b M M = 0
    unfolding KL-divergence-def
    by auto
qed

lemma (in information-space) KL-eq-0-iff-eq:
  fixes D :: 'a ⇒ real
  assumes prob-space (density M D)
  assumes D: D ∈ borel-measurable M AE x in M. 0 ≤ D x
  assumes int: integrable M (λx. D x * log b (D x))
  shows KL-divergence b M (density M D) = 0 ⟷ density M D = M
  using KL-same-eq-0[of b] KL-gt-0[OF assms]
  by (auto simp: less-le)

lemma (in information-space) KL-eq-0-iff-eq-ac:
  fixes D :: 'a ⇒ real
  assumes prob-space N
  assumes ac: absolutely-continuous M N sets N = sets M
  assumes int: integrable N (entropy-density b M N)
  shows KL-divergence b M N = 0 ⟷ N = M
proof −
  interpret N: prob-space N by fact
  have finite-measure N by unfold-locales
  from real-RN-deriv[OF this ac] obtain D
    where D:
      random-variable borel D
      AE x in M. RN-deriv M N x = ennreal (D x)
      AE x in N. 0 < D x
      ⋀x. 0 ≤ D x
    by this auto

  have N = density M (RN-deriv M N)
    using ac by (rule density-RN-deriv[symmetric])
  also have ... = density M D
    using D by (auto intro!: density-cong)
  finally have N: N = density M D .

 from absolutely-continuous-AE[OF ac(2,1) D(2)] D b-gt-1 ac measurable-entropy-density
  have integrable N (λx. log b (D x))
    by (intro integrable-cong-AE[THEN iffD2, OF - - - int])
       (auto simp: N entropy-density-def)
  with D b-gt-1 have integrable M (λx. D x * log b (D x))
   by (subst integrable-real-density[symmetric]) (auto simp: N[symmetric] comp-def)
  with ‹prob-space N› D show ?thesis
    unfolding N
    by (intro KL-eq-0-iff-eq) auto
qed

lemma (in information-space) KL-nonneg:
```

assumes *prob-space* (*density M D*)
assumes *D*: *D* ∈ *borel-measurable M AE x in M*. *0* ≤ *D x*
assumes *int*: *integrable M* (λ*x*. *D x* ∗ *log b* (*D x*))
shows *0* ≤ *KL-divergence b M* (*density M D*)
using *KL-gt-0*[*OF assms*] **by** (*cases density M D* = *M*) (*auto simp*: *KL-same-eq-0*)

lemma (**in** *sigma-finite-measure*) *KL-density-density-nonneg*:
fixes *f g* :: *'a* ⇒ *real*
assumes *1* < *b*
assumes *f*: *f* ∈ *borel-measurable M AE x in M*. *0* ≤ *f x prob-space* (*density M f*)
assumes *g*: *g* ∈ *borel-measurable M AE x in M*. *0* ≤ *g x prob-space* (*density M g*)
assumes *ac*: *AE x in M*. *f x* = *0* ⟶ *g x* = *0*
assumes *int*: *integrable M* (λ*x*. *g x* ∗ *log b* (*g x* / *f x*))
shows *0* ≤ *KL-divergence b* (*density M f*) (*density M g*)
proof −
interpret *Mf*: *prob-space density M f* **by** *fact*
interpret *Mf*: *information-space density M f b* **by** *standard fact*
have *eq*: *density* (*density M f*) (λ*x*. *g x* / *f x*) = *density M g* (**is** *?DD* = -)
using *f g ac* **by** (*subst density-density-divide*) *simp-all*

have *0* ≤ *KL-divergence b* (*density M f*) (*density* (*density M f*) (λ*x*. *g x* / *f x*))
proof (*rule Mf.KL-nonneg*)
show *prob-space ?DD* **unfolding** *eq* **by** *fact*
from *f g* **show** (λ*x*. *g x* / *f x*) ∈ *borel-measurable* (*density M f*)
by *auto*
show *AE x in density M f*. *0* ≤ *g x* / *f x*
using *f g* **by** (*auto simp*: *AE-density*)
show *integrable* (*density M f*) (λ*x*. *g x* / *f x* ∗ *log b* (*g x* / *f x*))
using ‹*1* < *b*› *f g ac*
by (*subst integrable-density*)
(*auto intro*!: *integrable-cong-AE*[*THEN iffD2*, *OF - - - int*] *measurable-If*)
qed
also have ... = *KL-divergence b* (*density M f*) (*density M g*)
using *f g ac* **by** (*subst density-density-divide*) *simp-all*
finally show *?thesis* **.**
qed

12.3 Finite Entropy

definition (**in** *information-space*) *finite-entropy* :: *'b measure* ⇒ (*'a* ⇒ *'b*) ⇒ (*'b* ⇒ *real*) ⇒ *bool*
where
finite-entropy S X f ⟷
distributed M S X f ∧
integrable S (λ*x*. *f x* ∗ *log b* (*f x*)) ∧
(∀ *x*∈*space S*. *0* ≤ *f x*)

lemma (**in** *information-space*) *finite-entropy-simple-function*:
assumes *X*: *simple-function M X*
shows *finite-entropy* (*count-space* (*X*‘*space M*)) *X* (λ*a*. *measure M* {*x* ∈ *space M*. *X x* = *a*})
unfolding *finite-entropy-def*
proof *safe*
have [*simp*]: *finite* (*X* ‘ *space M*)
using *X* **by** (*auto simp*: *simple-function-def*)
then show *integrable* (*count-space* (*X* ‘ *space M*))
(λ*x*. *prob* {*xa* ∈ *space M*. *X xa* = *x*} ∗ *log b* (*prob* {*xa* ∈ *space M*. *X xa* = *x*}))
by (*rule integrable-count-space*)
have *d*: *distributed M* (*count-space* (*X* ‘ *space M*)) *X* (λ*x*. *ennreal* (*if x* ∈ *X*‘*space M then prob* {*xa* ∈ *space M*. *X xa* = *x*} *else 0*))
by (*rule distributed-simple-function-superset*[*OF X*]) (*auto intro*!: *arg-cong*[**where** *f*=*prob*])
show *distributed M* (*count-space* (*X* ‘ *space M*)) *X* (λ*x*. *ennreal* (*prob* {*xa* ∈ *space M*. *X xa* = *x*}))
by (*rule distributed-cong-density*[*THEN iffD1*, *OF - - - d*]) *auto*
qed (*rule measure-nonneg*)

lemma *ac-fst*:
assumes *sigma-finite-measure T*
shows *absolutely-continuous S* (*distr* (*S* ⨂$_M$ *T*) *S fst*)
proof −
interpret *sigma-finite-measure T* **by** *fact*
{ fix *A* **assume** *A*: *A* ∈ *sets S emeasure S A* = *0*
then have *fst* −‘ *A* ∩ *space* (*S* ⨂$_M$ *T*) = *A* × *space T*
by (*auto simp*: *space-pair-measure dest*!: *sets.sets-into-space*)
with *A* **have** *emeasure* (*S* ⨂$_M$ *T*) (*fst* −‘ *A* ∩ *space* (*S* ⨂$_M$ *T*)) = *0*
by (*simp add*: *emeasure-pair-measure-Times*) **}**
then show *?thesis*
unfolding *absolutely-continuous-def*
apply (*auto simp*: *null-sets-distr-iff*)
apply (*auto simp*: *null-sets-def intro*!: *measurable-sets*)
done
qed

lemma *ac-snd*:
assumes *sigma-finite-measure T*
shows *absolutely-continuous T* (*distr* (*S* ⨂$_M$ *T*) *T snd*)
proof −
interpret *sigma-finite-measure T* **by** *fact*
{ fix *A* **assume** *A*: *A* ∈ *sets T emeasure T A* = *0*
then have *snd* −‘ *A* ∩ *space* (*S* ⨂$_M$ *T*) = *space S* × *A*
by (*auto simp*: *space-pair-measure dest*!: *sets.sets-into-space*)
with *A* **have** *emeasure* (*S* ⨂$_M$ *T*) (*snd* −‘ *A* ∩ *space* (*S* ⨂$_M$ *T*)) = *0*
by (*simp add*: *emeasure-pair-measure-Times*) **}**
then show *?thesis*
unfolding *absolutely-continuous-def*

apply (*auto simp*: *null-sets-distr-iff*)
apply (*auto simp*: *null-sets-def intro*!: *measurable-sets*)
done
qed

lemma (**in** *information-space*) *finite-entropy-integrable*:
finite-entropy S X Px ⟹ *integrable S* (λ*x*. *Px x* ∗ *log b* (*Px x*))
unfolding *finite-entropy-def* **by** *auto*

lemma (**in** *information-space*) *finite-entropy-distributed*:
finite-entropy S X Px ⟹ *distributed M S X Px*
unfolding *finite-entropy-def* **by** *auto*

lemma (**in** *information-space*) *finite-entropy-nn*:
finite-entropy S X Px ⟹ *x* ∈ *space S* ⟹ *0* ≤ *Px x*
by (*auto simp*: *finite-entropy-def*)

lemma (**in** *information-space*) *finite-entropy-measurable*:
finite-entropy S X Px ⟹ *Px* ∈ *S* →$_M$ *borel*
using *distributed-real-measurable*[*of S Px M X*]
finite-entropy-nn[*of S X Px*] *finite-entropy-distributed*[*of S X Px*] **by** *auto*

lemma (**in** *information-space*) *subdensity-finite-entropy*:
fixes *g* :: *'b* ⇒ *real* **and** *f* :: *'c* ⇒ *real*
assumes *T*: *T* ∈ *measurable P Q*
assumes *f*: *finite-entropy P X f*
assumes *g*: *finite-entropy Q Y g*
assumes *Y*: *Y* = *T* ∘ *X*
shows *AE x in P*. *g* (*T x*) = *0* ⟶ *f x* = *0*
using *subdensity*[*OF T*, *of M X* λ*x*. *ennreal* (*f x*) *Y* λ*x*. *ennreal* (*g x*)]
finite-entropy-distributed[*OF f*] *finite-entropy-distributed*[*OF g*]
finite-entropy-nn[*OF f*] *finite-entropy-nn*[*OF g*]
assms
by *auto*

lemma (**in** *information-space*) *finite-entropy-integrable-transform*:
finite-entropy S X Px ⟹ *distributed M T Y Py* ⟹ (⋀*x*. *x* ∈ *space T* ⟹ *0* ≤ *Py x*) ⟹
X = (λ*x*. *f* (*Y x*)) ⟹ *f* ∈ *measurable T S* ⟹ *integrable T* (λ*x*. *Py x* ∗ *log b* (*Px* (*f x*)))
using *distributed-transform-integrable*[*of M T Y Py S X Px f* λ*x*. *log b* (*Px x*)]
using *distributed-real-measurable*[*of S Px M X*]
by (*auto simp*: *finite-entropy-def*)

12.4 Mutual Information

definition (**in** *prob-space*)
mutual-information b S T X Y =
KL-divergence b (*distr M S X* ⨂$_M$ *distr M T Y*) (*distr M* (*S* ⨂$_M$ *T*) (λ*x*.

$(X\ x,\ Y\ x)))$

lemma (**in** *information-space*) *mutual-information-indep-vars*:
fixes $S\ T\ X\ Y$
defines $P \equiv$ *distr* $M\ S\ X \bigotimes_M$ *distr* $M\ T\ Y$
defines $Q \equiv$ *distr* $M\ (S \bigotimes_M T)\ (\lambda x.\ (X\ x,\ Y\ x))$
shows *indep-var* $S\ X\ T\ Y \longleftrightarrow$
($random\text{-}variable\ S\ X \wedge random\text{-}variable\ T\ Y \wedge$
absolutely-continuous $P\ Q \wedge$ *integrable* Q (*entropy-density* $b\ P\ Q) \wedge$
mutual-information $b\ S\ T\ X\ Y = 0$)
unfolding *indep-var-distribution-eq*
proof *safe*
assume *rv*[*measurable*]: *random-variable* $S\ X$ *random-variable* $T\ Y$

interpret X: *prob-space distr* $M\ S\ X$
by (*rule prob-space-distr*) *fact*
interpret Y: *prob-space distr* $M\ T\ Y$
by (*rule prob-space-distr*) *fact*
interpret XY: *pair-prob-space distr* $M\ S\ X$ *distr* $M\ T\ Y$ **by** *standard*
interpret P: *information-space* $P\ b$ **unfolding** *P-def* **by** *standard* (*rule b-gt-1*)

interpret Q: *prob-space* Q **unfolding** *Q-def*
by (*rule prob-space-distr*) *simp*

{ assume *distr* $M\ S\ X \bigotimes_M$ *distr* $M\ T\ Y =$ *distr* $M\ (S \bigotimes_M T)\ (\lambda x.\ (X\ x,\ Y\ x))$
then have [*simp*]: $Q = P$ **unfolding** *Q-def P-def* **by** *simp*

show *ac*: *absolutely-continuous* $P\ Q$ **by** (*simp add*: *absolutely-continuous-def*)
then have *ed*: *entropy-density* $b\ P\ Q \in$ *borel-measurable* P
by *simp*

have *AE x in P. 1 = RN-deriv P Q x*
proof (*rule P.RN-deriv-unique*)
show *density* $P\ (\lambda x.\ 1) = Q$
unfolding ‹$Q = P$› **by** (*intro measure-eqI*) (*auto simp*: *emeasure-density*)
qed *auto*
then have *ae-0*: *AE x in P. entropy-density b P Q x = 0*
by *eventually-elim* (*auto simp*: *entropy-density-def*)
then have *integrable P* (*entropy-density b P Q*) $\longleftrightarrow$ *integrable Q* ($\lambda x.\ 0$::*real*)
using *ed* **unfolding** ‹$Q = P$› **by** (*intro integrable-cong-AE*) *auto*
then show *integrable Q* (*entropy-density b P Q*) **by** *simp*

from *ae-0* **have** *mutual-information b S T X Y* $= (\int x.\ 0\ \partial P)$
unfolding *mutual-information-def KL-divergence-def P-def*[*symmetric*] *Q-def*[*symmetric*] ‹$Q = P$›
by (*intro integral-cong-AE*) *auto*
then show *mutual-information b S T X Y = 0*
by *simp* **}**

{ assume *ac*: *absolutely-continuous P Q*
assume *int*: *integrable Q* (*entropy-density b P Q*)
assume *I-eq-0*: *mutual-information b S T X Y = 0*

have *eq*: *Q = P*
proof (*rule P.KL-eq-0-iff-eq-ac*[*THEN iffD1*])
show *prob-space Q* **by** *unfold-locales*
show *absolutely-continuous P Q* **by** *fact*
show *integrable Q* (*entropy-density b P Q*) **by** *fact*
show *sets Q = sets P* **by** (*simp add*: *P-def Q-def sets-pair-measure*)
show *KL-divergence b P Q = 0*
using *I-eq-0* **unfolding** *mutual-information-def* **by** (*simp add*: *P-def Q-def*)
qed
then show *distr M S X* $\bigotimes_M$ *distr M T Y = distr M* (*S* $\bigotimes_M$ *T*) (*λx*. (*X x*, *Y x*))
unfolding *P-def Q-def* **.. }**
qed

abbreviation (**in** *information-space*)
mutual-information-Pow (*ℐ'*(- ; -')) **where**
ℐ(*X* ; *Y*) ≡ *mutual-information b* (*count-space* (*X*'*space M*)) (*count-space* (*Y*'*space M*)) *X Y*

lemma (**in** *information-space*)
fixes *Pxy* :: *'b* × *'c* ⇒ *real* **and** *Px* :: *'b* ⇒ *real* **and** *Py* :: *'c* ⇒ *real*
assumes *S*: *sigma-finite-measure S* **and** *T*: *sigma-finite-measure T*
assumes *Fx*: *finite-entropy S X Px* **and** *Fy*: *finite-entropy T Y Py*
assumes *Fxy*: *finite-entropy* (*S* $\bigotimes_M$ *T*) (*λx*. (*X x*, *Y x*)) *Pxy*
defines *f* ≡ *λx*. *Pxy x* ∗ *log b* (*Pxy x* / (*Px* (*fst x*) ∗ *Py* (*snd x*)))
shows *mutual-information-distr'*: *mutual-information b S T X Y = integral*^{L} (*S* $\bigotimes_M$ *T*) *f* (**is** *?M = ?R*)
and *mutual-information-nonneg'*: *0* ≤ *mutual-information b S T X Y*
proof −
have *Px*: *distributed M S X Px* **and** *Px-nn*: ⋀*x*. *x* ∈ *space S* ⟹ *0* ≤ *Px x*
using *Fx* **by** (*auto simp*: *finite-entropy-def*)
have *Py*: *distributed M T Y Py* **and** *Py-nn*: ⋀*x*. *x* ∈ *space T* ⟹ *0* ≤ *Py x*
using *Fy* **by** (*auto simp*: *finite-entropy-def*)
have *Pxy*: *distributed M* (*S* $\bigotimes_M$ *T*) (*λx*. (*X x*, *Y x*)) *Pxy*
and *Pxy-nn*: ⋀*x*. *x* ∈ *space* (*S* $\bigotimes_M$ *T*) ⟹ *0* ≤ *Pxy x*
⋀*x y*. *x* ∈ *space S* ⟹ *y* ∈ *space T* ⟹ *0* ≤ *Pxy* (*x*, *y*)
using *Fxy* **by** (*auto simp*: *finite-entropy-def space-pair-measure*)

have [*measurable*]: *Px* ∈ *S* →$_M$ *borel*
using *Px Px-nn* **by** (*intro distributed-real-measurable*)
have [*measurable*]: *Py* ∈ *T* →$_M$ *borel*
using *Py Py-nn* **by** (*intro distributed-real-measurable*)
have *measurable-Pxy*[*measurable*]: *Pxy* ∈ (*S* $\bigotimes_M$ *T*) →$_M$ *borel*
using *Pxy Pxy-nn* **by** (*intro distributed-real-measurable*) (*auto simp*: *space-pair-measure*)

have X[*measurable*]: *random-variable S X*
using *Px* **by** *auto*
have Y[*measurable*]: *random-variable T Y*
using *Py* **by** *auto*
interpret S: *sigma-finite-measure S* **by** *fact*
interpret T: *sigma-finite-measure T* **by** *fact*
interpret ST: *pair-sigma-finite S T* **..**
interpret X: *prob-space distr M S X* **using** X **by** (*rule prob-space-distr*)
interpret Y: *prob-space distr M T Y* **using** Y **by** (*rule prob-space-distr*)
interpret XY: *pair-prob-space distr M S X distr M T Y* **..**
let *?P* = $S \bigotimes_M T$
let *?D* = *distr M ?P* ($\lambda x.$ ($X\ x$, $Y\ x$))

{ fix A **assume** $A \in$ *sets S*
with X[*THEN measurable-space*] Y[*THEN measurable-space*]
have *emeasure* (*distr M S X*) A = *emeasure ?D* ($A \times$ *space T*)
by (*auto simp*: *emeasure-distr intro*!: *arg-cong*[**where** *f*=*emeasure M*]) **}**
note *marginal-eq1* = *this*
{ fix A **assume** $A \in$ *sets T*
with X[*THEN measurable-space*] Y[*THEN measurable-space*]
have *emeasure* (*distr M T Y*) A = *emeasure ?D* (*space S* $\times$ A)
by (*auto simp*: *emeasure-distr intro*!: *arg-cong*[**where** *f*=*emeasure M*]) **}**
note *marginal-eq2* = *this*

have *distr-eq*: *distr M S X* $\bigotimes_M$ *distr M T Y* = *density ?P* ($\lambda x.$ *ennreal* (*Px* (*fst x*) $*$ *Py* (*snd x*)))
unfolding *Px*(*1*)[*THEN distributed-distr-eq-density*] *Py*(*1*)[*THEN distributed-distr-eq-density*]
proof (*subst pair-measure-density*)
show ($\lambda x.$ *ennreal* (*Px x*)) $\in$ *borel-measurable S* ($\lambda y.$ *ennreal* (*Py y*)) $\in$ *borel-measurable T*
using *Px Py* **by** (*auto simp*: *distributed-def*)
show *sigma-finite-measure* (*density T Py*) **unfolding** *Py*(*1*)[*THEN distributed-distr-eq-density*, *symmetric*] **..**
show *density* ($S \bigotimes_M T$) ($\lambda(x, y).$ *ennreal* (*Px x*) $*$ *ennreal* (*Py y*)) =
density ($S \bigotimes_M T$) ($\lambda x.$ *ennreal* (*Px* (*fst x*) $*$ *Py* (*snd x*)))
using *Px-nn Py-nn* **by** (*auto intro*!: *density-cong simp*: *distributed-def ennreal-mult space-pair-measure*)
qed *fact*

have M: *?M* = *KL-divergence b* (*density ?P* ($\lambda x.$ *ennreal* (*Px* (*fst x*) $*$ *Py* (*snd x*)))) (*density ?P* ($\lambda x.$ *ennreal* (*Pxy x*)))
unfolding *mutual-information-def distr-eq Pxy*(*1*)[*THEN distributed-distr-eq-density*]
..

from *Px Py* **have** f: ($\lambda x.$ *Px* (*fst x*) $*$ *Py* (*snd x*)) $\in$ *borel-measurable ?P*
by (*intro borel-measurable-times*) (*auto intro*: *distributed-real-measurable measurable-fst''* *measurable-snd''*)
have *PxPy-nonneg*: *AE x in ?P*. 0 $\leq$ *Px* (*fst x*) $*$ *Py* (*snd x*)

using *Px-nn Py-nn* **by** (*auto simp*: *space-pair-measure*)

have *A*: (*AE x in ?P. Px* (*fst x*) = *0* ⟶ *Pxy x* = *0*)
by (*rule subdensity-real*[*OF measurable-fst Pxy Px*]) (*insert Px-nn Pxy-nn*, *auto simp*: *space-pair-measure*)
moreover
have *B*: (*AE x in ?P. Py* (*snd x*) = *0* ⟶ *Pxy x* = *0*)
by (*rule subdensity-real*[*OF measurable-snd Pxy Py*]) (*insert Py-nn Pxy-nn*, *auto simp*: *space-pair-measure*)
ultimately have *ac*: *AE x in ?P. Px* (*fst x*) ∗ *Py* (*snd x*) = *0* ⟶ *Pxy x* = *0*
by *eventually-elim auto*

show *?M* = *?R*
unfolding *M f-def* **using** *Pxy-nn Px-nn Py-nn*
by (*intro ST.KL-density-density b-gt-1 f PxPy-nonneg ac*) (*auto simp*: *space-pair-measure*)

have *X*: *X* = *fst* ∘ (λ*x*. (*X x*, *Y x*)) **and** *Y*: *Y* = *snd* ∘ (λ*x*. (*X x*, *Y x*))
by *auto*

have *integrable* (*S* ⨂$_M$ *T*) (λ*x*. *Pxy x* ∗ *log b* (*Pxy x*) − *Pxy x* ∗ *log b* (*Px* (*fst x*)) − *Pxy x* ∗ *log b* (*Py* (*snd x*)))
using *finite-entropy-integrable*[*OF Fxy*]
using *finite-entropy-integrable-transform*[*OF Fx Pxy*, *of fst*]
using *finite-entropy-integrable-transform*[*OF Fy Pxy*, *of snd*]
by (*simp add*: *Pxy-nn*)
moreover have *f* ∈ *borel-measurable* (*S* ⨂$_M$ *T*)
unfolding *f-def* **using** *Px Py Pxy*
by (*auto intro*: *distributed-real-measurable measurable-fst′ measurable-snd′*
intro!: *borel-measurable-times borel-measurable-log borel-measurable-divide*)
ultimately have *int*: *integrable* (*S* ⨂$_M$ *T*) *f*
apply (*rule integrable-cong-AE-imp*)
using *A B AE-space*
by *eventually-elim*
(*auto simp*: *f-def log-divide-eq log-mult-eq field-simps space-pair-measure Px-nn Py-nn Pxy-nn*
less-le)

show *0* ≤ *?M* **unfolding** *M*
proof (*intro ST.KL-density-density-nonneg*)
show *prob-space* (*density* (*S* ⨂$_M$ *T*) (λ*x*. *ennreal* (*Pxy x*)))
unfolding *distributed-distr-eq-density*[*OF Pxy*, *symmetric*]
using *distributed-measurable*[*OF Pxy*] **by** (*rule prob-space-distr*)
show *prob-space* (*density* (*S* ⨂$_M$ *T*) (λ*x*. *ennreal* (*Px* (*fst x*) ∗ *Py* (*snd x*))))
unfolding *distr-eq*[*symmetric*] **by** *unfold-locales*
show *integrable* (*S* ⨂$_M$ *T*) (λ*x*. *Pxy x* ∗ *log b* (*Pxy x* / (*Px* (*fst x*) ∗ *Py* (*snd x*))))
using *int* **unfolding** *f-def* **.**
qed (*insert ac*, *auto simp*: *b-gt-1 Px-nn Py-nn Pxy-nn space-pair-measure*)
qed

lemma (**in** *information-space*)
fixes $Pxy :: 'b \times 'c \Rightarrow real$ **and** $Px :: 'b \Rightarrow real$ **and** $Py :: 'c \Rightarrow real$
assumes *sigma-finite-measure S sigma-finite-measure T*
assumes *Px*: *distributed M S X Px* **and** *Px-nn*: $\bigwedge x.\ x \in space\ S \Longrightarrow 0 \leq Px\ x$
and *Py*: *distributed M T Y Py* **and** *Py-nn*: $\bigwedge y.\ y \in space\ T \Longrightarrow 0 \leq Py\ y$
and *Pxy*: *distributed M* $(S \bigotimes_M T)$ $(\lambda x.\ (X\ x,\ Y\ x))$ *Pxy*
and *Pxy-nn*: $\bigwedge x\ y.\ x \in space\ S \Longrightarrow y \in space\ T \Longrightarrow 0 \leq Pxy\ (x,\ y)$
defines $f \equiv \lambda x.\ Pxy\ x * log\ b\ (Pxy\ x\ /\ (Px\ (fst\ x) * Py\ (snd\ x)))$
shows *mutual-information-distr*: *mutual-information b S T X Y* = $integral^L$ $(S \bigotimes_M T)$ *f* (**is** *?M* = *?R*)
and *mutual-information-nonneg*: *integrable* $(S \bigotimes_M T)$ *f* $\Longrightarrow 0 \leq$ *mutual-information b S T X Y*
proof −
have *X*[*measurable*]: *random-variable S X*
using *Px* **by** (*auto simp*: *distributed-def*)
have *Y*[*measurable*]: *random-variable T Y*
using *Py* **by** (*auto simp*: *distributed-def*)
have [*measurable*]: $Px \in S \rightarrow_M borel$
using *Px Px-nn* **by** (*intro distributed-real-measurable*)
have [*measurable*]: $Py \in T \rightarrow_M borel$
using *Py Py-nn* **by** (*intro distributed-real-measurable*)
have *measurable-Pxy*[*measurable*]: $Pxy \in (S \bigotimes_M T) \rightarrow_M borel$
using *Pxy Pxy-nn* **by** (*intro distributed-real-measurable*) (*auto simp*: *space-pair-measure*)

interpret *S*: *sigma-finite-measure S* **by** *fact*
interpret *T*: *sigma-finite-measure T* **by** *fact*
interpret *ST*: *pair-sigma-finite S T* **..**
interpret *X*: *prob-space distr M S X* **using** *X* **by** (*rule prob-space-distr*)
interpret *Y*: *prob-space distr M T Y* **using** *Y* **by** (*rule prob-space-distr*)
interpret *XY*: *pair-prob-space distr M S X distr M T Y* **..**
let *?P* = $S \bigotimes_M T$
let *?D* = *distr M ?P* $(\lambda x.\ (X\ x,\ Y\ x))$

{ fix *A* **assume** $A \in sets\ S$
with *X*[*THEN measurable-space*] *Y*[*THEN measurable-space*]
have *emeasure* (*distr M S X*) *A* = *emeasure ?D* $(A \times space\ T)$
by (*auto simp*: *emeasure-distr intro*!: *arg-cong*[**where** *f*=*emeasure M*]) **}**
note *marginal-eq1* = *this*
{ fix *A* **assume** $A \in sets\ T$
with *X*[*THEN measurable-space*] *Y*[*THEN measurable-space*]
have *emeasure* (*distr M T Y*) *A* = *emeasure ?D* $(space\ S \times A)$
by (*auto simp*: *emeasure-distr intro*!: *arg-cong*[**where** *f*=*emeasure M*]) **}**
note *marginal-eq2* = *this*

have *distr-eq*: *distr M S X* $\bigotimes_M$ *distr M T Y* = *density ?P* $(\lambda x.\ ennreal\ (Px\ (fst\ x) * Py\ (snd\ x)))$
unfolding *Px*(*1*)[*THEN distributed-distr-eq-density*] *Py*(*1*)[*THEN distributed-distr-eq-density*]
proof (*subst pair-measure-density*)

show ($\lambda x.$ *ennreal* (*Px x*)) ∈ *borel-measurable S* ($\lambda y.$ *ennreal* (*Py y*)) ∈ *borel-measurable T*
using *Px Py* **by** (*auto simp*: *distributed-def*)
show *sigma-finite-measure* (*density T Py*) **unfolding** *Py*(*1*)[*THEN distributed-distr-eq-density*, *symmetric*] **..**
show *density* (*S* $\bigotimes_M$ *T*) ($\lambda(x, y).$ *ennreal* (*Px x*) ∗ *ennreal* (*Py y*)) =
density (*S* $\bigotimes_M$ *T*) ($\lambda x.$ *ennreal* (*Px* (*fst x*) ∗ *Py* (*snd x*)))
using *Px-nn Py-nn* **by** (*auto intro*!: *density-cong simp*: *distributed-def ennreal-mult space-pair-measure*)
qed *fact*

have *M*: *?M* = *KL-divergence b* (*density ?P* ($\lambda x.$ *ennreal* (*Px* (*fst x*) ∗ *Py* (*snd x*)))) (*density ?P* ($\lambda x.$ *ennreal* (*Pxy x*)))
unfolding *mutual-information-def distr-eq Pxy*(*1*)[*THEN distributed-distr-eq-density*]
..

from *Px Py* **have** *f*: ($\lambda x.$ *Px* (*fst x*) ∗ *Py* (*snd x*)) ∈ *borel-measurable ?P*
by (*intro borel-measurable-times*) (*auto intro*: *distributed-real-measurable measurable-fst'' measurable-snd''*)
have *PxPy-nonneg*: *AE x in ?P*. *0* ≤ *Px* (*fst x*) ∗ *Py* (*snd x*)
using *Px-nn Py-nn* **by** (*auto simp*: *space-pair-measure*)

have (*AE x in ?P*. *Px* (*fst x*) = *0* ⟶ *Pxy x* = *0*)
by (*rule subdensity-real*[*OF measurable-fst Pxy Px*]) (*insert Px-nn Pxy-nn*, *auto simp*: *space-pair-measure*)
moreover
have (*AE x in ?P*. *Py* (*snd x*) = *0* ⟶ *Pxy x* = *0*)
by (*rule subdensity-real*[*OF measurable-snd Pxy Py*]) (*insert Py-nn Pxy-nn*, *auto simp*: *space-pair-measure*)
ultimately have *ac*: *AE x in ?P*. *Px* (*fst x*) ∗ *Py* (*snd x*) = *0* ⟶ *Pxy x* = *0*
by *eventually-elim auto*

show *?M* = *?R*
unfolding *M f-def*
using *b-gt-1 f PxPy-nonneg ac Pxy-nn*
by (*intro ST.KL-density-density*) (*auto simp*: *space-pair-measure*)

assume *int*: *integrable* (*S* $\bigotimes_M$ *T*) *f*
show *0* ≤ *?M* **unfolding** *M*
proof (*intro ST.KL-density-density-nonneg*)
show *prob-space* (*density* (*S* $\bigotimes_M$ *T*) ($\lambda x.$ *ennreal* (*Pxy x*)))
unfolding *distributed-distr-eq-density*[*OF Pxy*, *symmetric*]
using *distributed-measurable*[*OF Pxy*] **by** (*rule prob-space-distr*)
show *prob-space* (*density* (*S* $\bigotimes_M$ *T*) ($\lambda x.$ *ennreal* (*Px* (*fst x*) ∗ *Py* (*snd x*))))
unfolding *distr-eq*[*symmetric*] **by** *unfold-locales*
show *integrable* (*S* $\bigotimes_M$ *T*) ($\lambda x.$ *Pxy x* ∗ *log b* (*Pxy x* / (*Px* (*fst x*) ∗ *Py* (*snd x*))))
using *int* **unfolding** *f-def* **.**
qed (*insert ac*, *auto simp*: *b-gt-1 Px-nn Py-nn Pxy-nn space-pair-measure*)

qed

lemma (**in** *information-space*)
fixes *Pxy* :: *'b* × *'c* ⇒ *real* **and** *Px* :: *'b* ⇒ *real* **and** *Py* :: *'c* ⇒ *real*
assumes *sigma-finite-measure S sigma-finite-measure T*
assumes *Px*[*measurable*]: *distributed M S X Px* **and** *Px-nn*: ⋀*x*. *x* ∈ *space S* ⟹ *0* ≤ *Px x*
and *Py*[*measurable*]: *distributed M T Y Py* **and** *Py-nn*: ⋀*x*. *x* ∈ *space T* ⟹ *0* ≤ *Py x*
and *Pxy*[*measurable*]: *distributed M* (*S* ⨂$_M$ *T*) (λ*x*. (*X x*, *Y x*)) *Pxy*
and *Pxy-nn*: ⋀*x*. *x* ∈ *space* (*S* ⨂$_M$ *T*) ⟹ *0* ≤ *Pxy x*
assumes *ae*: *AE x in S*. *AE y in T*. *Pxy* (*x*, *y*) = *Px x* ∗ *Py y*
shows *mutual-information-eq-0*: *mutual-information b S T X Y* = *0*
proof −
interpret *S*: *sigma-finite-measure S* **by** *fact*
interpret *T*: *sigma-finite-measure T* **by** *fact*
interpret *ST*: *pair-sigma-finite S T* **..**
note
distributed-real-measurable[*OF Px-nn Px*, *measurable*]
distributed-real-measurable[*OF Py-nn Py*, *measurable*]
distributed-real-measurable[*OF Pxy-nn Pxy*, *measurable*]

have *AE x in S* ⨂$_M$ *T*. *Px* (*fst x*) = *0* ⟶ *Pxy x* = *0*
by (*rule subdensity-real*[*OF measurable-fst Pxy Px*]) (*auto simp*: *Px-nn Pxy-nn space-pair-measure*)
moreover
have *AE x in S* ⨂$_M$ *T*. *Py* (*snd x*) = *0* ⟶ *Pxy x* = *0*
by (*rule subdensity-real*[*OF measurable-snd Pxy Py*]) (*auto simp*: *Py-nn Pxy-nn space-pair-measure*)
moreover
have *AE x in S* ⨂$_M$ *T*. *Pxy x* = *Px* (*fst x*) ∗ *Py* (*snd x*)
by (*intro ST.AE-pair-measure*) (*auto simp*: *ae intro*!: *measurable-snd′′ measurable-fst′′*)
ultimately have *AE x in S* ⨂$_M$ *T*. *Pxy x* ∗ *log b* (*Pxy x* / (*Px* (*fst x*) ∗ *Py* (*snd x*))) = *0*
by *eventually-elim simp*
then have (∫ *x*. *Pxy x* ∗ *log b* (*Pxy x* / (*Px* (*fst x*) ∗ *Py* (*snd x*))) ∂(*S* ⨂$_M$ *T*)) = (∫ *x*. *0* ∂(*S* ⨂$_M$ *T*))
by (*intro integral-cong-AE*) *auto*
then show *?thesis*
by (*subst mutual-information-distr*[*OF assms*(*1*−*8*)]) (*auto simp add*: *space-pair-measure*)
qed

lemma (**in** *information-space*) *mutual-information-simple-distributed*:
assumes *X*: *simple-distributed M X Px* **and** *Y*: *simple-distributed M Y Py*
assumes *XY*: *simple-distributed M* (λ*x*. (*X x*, *Y x*)) *Pxy*
shows *I*(*X* ; *Y*) = (∑ (*x*, *y*)∈(λ*x*. (*X x*, *Y x*))‘*space M*. *Pxy* (*x*, *y*) ∗ *log b* (*Pxy* (*x*, *y*) / (*Px x* ∗ *Py y*)))
proof (*subst mutual-information-distr*[*OF - - simple-distributed*[*OF X*] *- simple-distributed*[*OF*

Y] - *simple-distributed-joint*[*OF XY*]])
note *fin* = *simple-distributed-joint-finite*[*OF XY*, *simp*]
show *sigma-finite-measure* (*count-space* (*X* ‘ *space M*))
by (*simp add*: *sigma-finite-measure-count-space-finite*)
show *sigma-finite-measure* (*count-space* (*Y* ‘ *space M*))
by (*simp add*: *sigma-finite-measure-count-space-finite*)
let *?Pxy* = λ*x*. (*if x* ∈ (λ*x*. (*X x*, *Y x*)) ‘ *space M then Pxy x else 0*)
let *?f* = λ*x*. *?Pxy x* ∗ *log b* (*?Pxy x* / (*Px* (*fst x*) ∗ *Py* (*snd x*)))
have ⋀*x*. *?f x* = (*if x* ∈ (λ*x*. (*X x*, *Y x*)) ‘ *space M then Pxy x* ∗ *log b* (*Pxy x* / (*Px* (*fst x*) ∗ *Py* (*snd x*))) *else 0*)
by *auto*
with *fin* **show** (∫ *x*. *?f x* ∂(*count-space* (*X* ‘ *space M*) ⨂$_M$ *count-space* (*Y* ‘ *space M*))) =
(∑(*x*, *y*)∈(λ*x*. (*X x*, *Y x*)) ‘ *space M*. *Pxy* (*x*, *y*) ∗ *log b* (*Pxy* (*x*, *y*) / (*Px x* ∗ *Py y*)))
by (*auto simp add*: *pair-measure-count-space lebesgue-integral-count-space-finite sum.If-cases split-beta′*
intro!: *sum.cong*)
qed (*insert X Y XY*, *auto simp*: *simple-distributed-def*)

lemma (**in** *information-space*)
fixes *Pxy* :: ′*b* × ′*c* ⇒ *real* **and** *Px* :: ′*b* ⇒ *real* **and** *Py* :: ′*c* ⇒ *real*
assumes *Px*: *simple-distributed M X Px* **and** *Py*: *simple-distributed M Y Py*
assumes *Pxy*: *simple-distributed M* (λ*x*. (*X x*, *Y x*)) *Pxy*
assumes *ae*: ∀ *x*∈*space M*. *Pxy* (*X x*, *Y x*) = *Px* (*X x*) ∗ *Py* (*Y x*)
shows *mutual-information-eq-0-simple*: $\mathcal{I}(X\ ;\ Y) = 0$
proof (*subst mutual-information-simple-distributed*[*OF Px Py Pxy*])
have (∑(*x*, *y*)∈(λ*x*. (*X x*, *Y x*)) ‘ *space M*. *Pxy* (*x*, *y*) ∗ *log b* (*Pxy* (*x*, *y*) / (*Px x* ∗ *Py y*))) =
(∑(*x*, *y*)∈(λ*x*. (*X x*, *Y x*)) ‘ *space M*. *0*)
by (*intro sum.cong*) (*auto simp*: *ae*)
then show (∑(*x*, *y*)∈(λ*x*. (*X x*, *Y x*)) ‘ *space M*.
Pxy (*x*, *y*) ∗ *log b* (*Pxy* (*x*, *y*) / (*Px x* ∗ *Py y*))) = *0* **by** *simp*
qed

12.5 Entropy

definition (**in** *prob-space*) *entropy* :: *real* ⇒ ′*b measure* ⇒ (′*a* ⇒ ′*b*) ⇒ *real* **where**
entropy b S X = − *KL-divergence b S* (*distr M S X*)

abbreviation (**in** *information-space*)
entropy-Pow (ℋ′(-′)) **where**
ℋ(*X*) ≡ *entropy b* (*count-space* (*X*‘*space M*)) *X*

lemma (**in** *prob-space*) *distributed-RN-deriv*:
assumes *X*: *distributed M S X Px*
shows *AE x in S*. *RN-deriv S* (*density S Px*) *x* = *Px x*
proof −
note *D* = *distributed-measurable*[*OF X*] *distributed-borel-measurable*[*OF X*]

interpret *X*: *prob-space distr M S X*
using *D*(*1*) **by** (*rule prob-space-distr*)

have *sf*: *sigma-finite-measure* (*distr M S X*) **by** *standard*
show *?thesis*
using *D*
apply (*subst eq-commute*)
apply (*intro RN-deriv-unique-sigma-finite*)
apply (*auto simp*: *distributed-distr-eq-density*[*symmetric*, *OF X*] *sf*)
done
qed

lemma (**in** *information-space*)
fixes *X* :: *'a* ⇒ *'b*
assumes *X*[*measurable*]: *distributed M MX X f* **and** *nn*: ⋀*x*. *x* ∈ *space MX* ⟹ *0* ≤ *f x*
shows *entropy-distr*: *entropy b MX X* = − (∫ *x*. *f x* ∗ *log b* (*f x*) ∂*MX*) (**is** *?eq*)
proof −
note *D* = *distributed-measurable*[*OF X*] *distributed-borel-measurable*[*OF X*]
note *ae* = *distributed-RN-deriv*[*OF X*]
note *distributed-real-measurable*[*OF nn X*, *measurable*]

have *ae-eq*: *AE x in distr M MX X*. *log b* (*enn2real* (*RN-deriv MX* (*distr M MX X*) *x*)) =
log b (*f x*)
unfolding *distributed-distr-eq-density*[*OF X*]
apply (*subst AE-density*)
using *D* **apply** *simp*
using *ae* **apply** *eventually-elim*
apply *auto*
done

have *int-eq*: (∫ *x*. *f x* ∗ *log b* (*f x*) ∂*MX*) = (∫ *x*. *log b* (*f x*) ∂*distr M MX X*)
unfolding *distributed-distr-eq-density*[*OF X*]
using *D*
by (*subst integral-density*) (*auto simp*: *nn*)

show *?eq*
unfolding *entropy-def KL-divergence-def entropy-density-def comp-def int-eq neg-equal-iff-equal*
using *ae-eq* **by** (*intro integral-cong-AE*) (*auto simp*: *nn*)
qed

lemma (**in** *information-space*) *entropy-le*:
fixes *Px* :: *'b* ⇒ *real* **and** *MX* :: *'b measure*
assumes *X*[*measurable*]: *distributed M MX X Px* **and** *Px-nn*[*simp*]: ⋀*x*. *x* ∈ *space MX* ⟹ *0* ≤ *Px x*
and *fin*: *emeasure MX* {*x* ∈ *space MX*. *Px x* ≠ *0*} ≠ *top*
and *int*: *integrable MX* (λ*x*. − *Px x* ∗ *log b* (*Px x*))

shows *entropy b MX X* ≤ *log b* (*measure MX* {*x* ∈ *space MX*. *Px x* ≠ *0*})
proof −
note *Px* = *distributed-borel-measurable*[*OF X*]
interpret *X*: *prob-space distr M MX X*
using *distributed-measurable*[*OF X*] **by** (*rule prob-space-distr*)

have − *log b* (*measure MX* {*x* ∈ *space MX*. *Px x* ≠ *0*}) =
− *log b* (∫ *x*. *indicator* {*x* ∈ *space MX*. *Px x* ≠ *0*} *x* ∂*MX*)
using *Px Px-nn fin* **by** (*auto simp*: *measure-def*)
also have − *log b* (∫ *x*. *indicator* {*x* ∈ *space MX*. *Px x* ≠ *0*} *x* ∂*MX*) = − *log b* (∫ *x*. *1* / *Px x* ∂*distr M MX X*)
proof −
have *integral*^{L} *MX* (*indicator* {*x* ∈ *space MX*. *Px x* ≠ *0*}) = *LINT x*|*MX*. *Px x* ∗$_R$ (*1* / *Px x*)
by (*rule Bochner-Integration.integral-cong*) *auto*
also have ... = *LINT x*|*density MX* (λ*x*. *ennreal* (*Px x*)). *1* / *Px x*
by (*rule integral-density* [*symmetric*]) (*use Px Px-nn* **in** *auto*)
finally show *?thesis*
unfolding *distributed-distr-eq-density*[*OF X*] **by** *simp*
qed
also have ... ≤ (∫ *x*. − *log b* (*1* / *Px x*) ∂*distr M MX X*)
proof (*rule X.jensens-inequality*[*of* λ*x*. *1* / *Px x* {*0*<..} *0 1* λ*x*. − *log b x*])
show *AE x in distr M MX X*. *1* / *Px x* ∈ {*0*<..}
unfolding *distributed-distr-eq-density*[*OF X*]
using *Px* **by** (*auto simp*: *AE-density*)
have [*simp*]: ⋀*x*. *x* ∈ *space MX* ⟹ *ennreal* (*if Px x* = *0 then 0 else 1*) = *indicator* {*x* ∈ *space MX*. *Px x* ≠ *0*} *x*
by (*auto simp*: *one-ennreal-def*)
have (∫$^+$ *x*. *ennreal* (− (*if Px x* = *0 then 0 else 1*)) ∂*MX*) = (∫$^+$ *x*. *0* ∂*MX*)
by (*intro nn-integral-cong*) (*auto simp*: *ennreal-neg*)
then show *integrable* (*distr M MX X*) (λ*x*. *1* / *Px x*)
unfolding *distributed-distr-eq-density*[*OF X*] **using** *Px*
by (*auto simp*: *nn-integral-density real-integrable-def fin ennreal-neg ennreal-mult*[*symmetric*]
cong: *nn-integral-cong*)
have *integrable MX* (λ*x*. *Px x* ∗ *log b* (*1* / *Px x*)) =
integrable MX (λ*x*. − *Px x* ∗ *log b* (*Px x*))
using *Px*
by (*intro integrable-cong-AE*) (*auto simp*: *log-divide-eq less-le*)
then show *integrable* (*distr M MX X*) (λ*x*. − *log b* (*1* / *Px x*))
unfolding *distributed-distr-eq-density*[*OF X*]
using *Px int*
by (*subst integrable-real-density*) *auto*
qed (*auto simp*: *minus-log-convex*[*OF b-gt-1*])
also have ... = (∫ *x*. *log b* (*Px x*) ∂*distr M MX X*)
unfolding *distributed-distr-eq-density*[*OF X*] **using** *Px*
by (*intro integral-cong-AE*) (*auto simp*: *AE-density log-divide-eq*)
also have ... = − *entropy b MX X*
unfolding *distributed-distr-eq-density*[*OF X*] **using** *Px*

```
    by (subst entropy-distr[OF X]) (auto simp: integral-density)
  finally show ?thesis
    by simp
qed

lemma (in information-space) entropy-le-space:
  fixes Px :: 'b ⇒ real and MX :: 'b measure
  assumes X: distributed M MX X Px and Px-nn[simp]: ⋀x. x ∈ space MX ⟹
0 ≤ Px x
  and fin: finite-measure MX
  and int: integrable MX (λx. − Px x ∗ log b (Px x))
  shows entropy b MX X ≤ log b (measure MX (space MX))
proof −
  note Px = distributed-borel-measurable[OF X]
  interpret finite-measure MX by fact
  have entropy b MX X ≤ log b (measure MX {x ∈ space MX. Px x ≠ 0})
    using int X by (intro entropy-le) auto
  also have ... ≤ log b (measure MX (space MX))
    using Px distributed-imp-emeasure-nonzero[OF X]
    by (intro log-le)
       (auto intro!: finite-measure-mono b-gt-1 less-le[THEN iffD2]
             simp: emeasure-eq-measure cong: conj-cong)
  finally show ?thesis .
qed

lemma (in information-space) entropy-uniform:
   assumes X: distributed M MX X (λx. indicator A x / measure MX A) (is
distributed - - - ?f)
  shows entropy b MX X = log b (measure MX A)
proof (subst entropy-distr[OF X])
  have [simp]: emeasure MX A ≠ ∞
    using uniform-distributed-params[OF X] by (auto simp add: measure-def)
  have eq: (∫ x. indicator A x / measure MX A ∗ log b (indicator A x / measure
MX A) ∂MX) =
    (∫ x. (− log b (measure MX A) / measure MX A) ∗ indicator A x ∂MX)
    using uniform-distributed-params[OF X]
     by (intro Bochner-Integration.integral-cong) (auto split: split-indicator simp:
log-divide-eq zero-less-measure-iff)
  show − (∫ x. indicator A x / measure MX A ∗ log b (indicator A x / measure
MX A) ∂MX) =
    log b (measure MX A)
    unfolding eq using uniform-distributed-params[OF X]
   by (subst Bochner-Integration.integral-mult-right) (auto simp: measure-def less-top[symmetric]
intro!: integrable-real-indicator)
qed simp

lemma (in information-space) entropy-simple-distributed:
  simple-distributed M X f ⟹ ℋ(X) = − (∑ x∈X‘space M. f x ∗ log b (f x))
  by (subst entropy-distr[OF simple-distributed])
```

(*auto simp add*: *lebesgue-integral-count-space-finite*)

lemma (**in** *information-space*) *entropy-le-card-not-0*:
assumes *X*: *simple-distributed M X f*
shows $\mathcal{H}(X) \leq log\ b\ (card\ (X\ `\ space\ M \cap \{x.\ f\ x \neq 0\}))$
proof −
let *?X* = *count-space* (*X*'*space M*)
have $\mathcal{H}(X) \leq log\ b\ (measure\ ?X\ \{x \in space\ ?X.\ f\ x \neq 0\})$
by (*rule entropy-le*[*OF simple-distributed*[*OF X*]])
(*insert X*, *auto simp*: *simple-distributed-finite*[*OF X*] *subset-eq integrable-count-space emeasure-count-space*)
also have *measure ?X* $\{x \in space\ ?X.\ f\ x \neq 0\} = card\ (X\ `\ space\ M \cap \{x.\ f\ x \neq 0\})$
by (*simp-all add*: *simple-distributed-finite*[*OF X*] *subset-eq emeasure-count-space measure-def Int-def*)
finally show *?thesis* **.**
qed

lemma (**in** *information-space*) *entropy-le-card*:
assumes *X*: *simple-distributed M X f*
shows $\mathcal{H}(X) \leq log\ b\ (real\ (card\ (X\ `\ space\ M)))$
proof −
let *?X* = *count-space* (*X*'*space M*)
have $\mathcal{H}(X) \leq log\ b\ (measure\ ?X\ (space\ ?X))$
by (*rule entropy-le-space*[*OF simple-distributed*[*OF X*]])
(*insert X*, *auto simp*: *simple-distributed-finite*[*OF X*] *subset-eq integrable-count-space emeasure-count-space finite-measure-count-space*)
also have *measure ?X* (*space ?X*) = *card* (*X* ' *space M*)
by (*simp-all add*: *simple-distributed-finite*[*OF X*] *subset-eq emeasure-count-space measure-def*)
finally show *?thesis* **.**
qed

12.6 Conditional Mutual Information

definition (**in** *prob-space*)
conditional-mutual-information b MX MY MZ X Y Z ≡
mutual-information b MX (*MY* $\bigotimes_M$ *MZ*) *X* (λ*x*. (*Y x*, *Z x*)) −
mutual-information b MX MZ X Z

abbreviation (**in** *information-space*)
conditional-mutual-information-Pow ($\mathcal{I}'(\ -\ ;\ -\ |\ -\ ')$) **where**
$\mathcal{I}(X\ ;\ Y\ |\ Z) \equiv$ *conditional-mutual-information b*
(*count-space* (*X* ' *space M*)) (*count-space* (*Y* ' *space M*)) (*count-space* (*Z* ' *space M*)) *X Y Z*

lemma (**in** *information-space*)
assumes *S*: *sigma-finite-measure S* **and** *T*: *sigma-finite-measure T* **and** *P*: *sigma-finite-measure P*

assumes *Px*[*measurable*]: *distributed M S X Px*
and *Px-nn*[*simp*]: $\bigwedge x.\ x \in space\ S \Longrightarrow 0 \leq Px\ x$
assumes *Pz*[*measurable*]: *distributed M P Z Pz*
and *Pz-nn*[*simp*]: $\bigwedge z.\ z \in space\ P \Longrightarrow 0 \leq Pz\ z$
assumes *Pyz*[*measurable*]: *distributed M* $(T \bigotimes_M P)$ $(\lambda x.\ (Y\ x,\ Z\ x))$ *Pyz*
and *Pyz-nn*[*simp*]: $\bigwedge y\ z.\ y \in space\ T \Longrightarrow z \in space\ P \Longrightarrow 0 \leq Pyz\ (y,\ z)$
assumes *Pxz*[*measurable*]: *distributed M* $(S \bigotimes_M P)$ $(\lambda x.\ (X\ x,\ Z\ x))$ *Pxz*
and *Pxz-nn*[*simp*]: $\bigwedge x\ z.\ x \in space\ S \Longrightarrow z \in space\ P \Longrightarrow 0 \leq Pxz\ (x,\ z)$
assumes *Pxyz*[*measurable*]: *distributed M* $(S \bigotimes_M T \bigotimes_M P)$ $(\lambda x.\ (X\ x,\ Y\ x,\ Z\ x))$ *Pxyz*
and *Pxyz-nn*[*simp*]: $\bigwedge x\ y\ z.\ x \in space\ S \Longrightarrow y \in space\ T \Longrightarrow z \in space\ P \Longrightarrow 0 \leq Pxyz\ (x,\ y,\ z)$
assumes *I1*: *integrable* $(S \bigotimes_M T \bigotimes_M P)$ $(\lambda(x,\ y,\ z).\ Pxyz\ (x,\ y,\ z) * log\ b\ (Pxyz\ (x,\ y,\ z)\ /\ (Px\ x * Pyz\ (y,\ z))))$
assumes *I2*: *integrable* $(S \bigotimes_M T \bigotimes_M P)$ $(\lambda(x,\ y,\ z).\ Pxyz\ (x,\ y,\ z) * log\ b\ (Pxz\ (x,\ z)\ /\ (Px\ x * Pz\ z)))$
shows *conditional-mutual-information-generic-eq*: *conditional-mutual-information b S T P X Y Z*
$= (\int (x,\ y,\ z).\ Pxyz\ (x,\ y,\ z) * log\ b\ (Pxyz\ (x,\ y,\ z)\ /\ (Pxz\ (x,\ z) * (Pyz\ (y,z)\ /\ Pz\ z)))\ \partial(S \bigotimes_M T \bigotimes_M P))$ (**is** *?eq*)
and *conditional-mutual-information-generic-nonneg*: $0 \leq$ *conditional-mutual-information b S T P X Y Z* (**is** *?nonneg*)
proof −
have [*measurable*]: $Px \in S \rightarrow_M borel$
using *Px Px-nn* **by** (*intro distributed-real-measurable*)
have [*measurable*]: $Pz \in P \rightarrow_M borel$
using *Pz Pz-nn* **by** (*intro distributed-real-measurable*)
have *measurable-Pyz*[*measurable*]: $Pyz \in (T \bigotimes_M P) \rightarrow_M borel$
using *Pyz Pyz-nn* **by** (*intro distributed-real-measurable*) (*auto simp*: *space-pair-measure*)
have *measurable-Pxz*[*measurable*]: $Pxz \in (S \bigotimes_M P) \rightarrow_M borel$
using *Pxz Pxz-nn* **by** (*intro distributed-real-measurable*) (*auto simp*: *space-pair-measure*)
have *measurable-Pxyz*[*measurable*]: $Pxyz \in (S \bigotimes_M T \bigotimes_M P) \rightarrow_M borel$
using *Pxyz Pxyz-nn* **by** (*intro distributed-real-measurable*) (*auto simp*: *space-pair-measure*)

interpret *S*: *sigma-finite-measure S* **by** *fact*
interpret *T*: *sigma-finite-measure T* **by** *fact*
interpret *P*: *sigma-finite-measure P* **by** *fact*
interpret *TP*: *pair-sigma-finite T P* **..**
interpret *SP*: *pair-sigma-finite S P* **..**
interpret *ST*: *pair-sigma-finite S T* **..**
interpret *SPT*: *pair-sigma-finite* $S \bigotimes_M P\ T$ **..**
interpret *STP*: *pair-sigma-finite* $S\ T \bigotimes_M P$ **..**
interpret *TPS*: *pair-sigma-finite* $T \bigotimes_M P\ S$ **..**
have *TP*: *sigma-finite-measure* $(T \bigotimes_M P)$ **..**
have *SP*: *sigma-finite-measure* $(S \bigotimes_M P)$ **..**
have *YZ*: *random-variable* $(T \bigotimes_M P)$ $(\lambda x.\ (Y\ x,\ Z\ x))$
using *Pyz* **by** (*simp add*: *distributed-measurable*)

from *Pxz Pxyz* **have** *distr-eq*: *distr M* $(S \bigotimes_M P)$ $(\lambda x.\ (X\ x,\ Z\ x)) =$

distr (*distr M* (*S* $\bigotimes_M$ *T* $\bigotimes_M$ *P*) (λ*x*. (*X x*, *Y x*, *Z x*))) (*S* $\bigotimes_M$ *P*) (λ(*x*, *y*, *z*). (*x*, *z*))
by (*simp add*: *comp-def distr-distr*)

have *mutual-information b S P X Z* =
(∫ *x*. *Pxz x* ∗ *log b* (*Pxz x* / (*Px* (*fst x*) ∗ *Pz* (*snd x*))) ∂(*S* $\bigotimes_M$ *P*))
by (*rule mutual-information-distr*[*OF S P Px Px-nn Pz Pz-nn Pxz Pxz-nn*])
also have ... = (∫ (*x*,*y*,*z*). *Pxyz* (*x*,*y*,*z*) ∗ *log b* (*Pxz* (*x*,*z*) / (*Px x* ∗ *Pz z*)) ∂(*S* $\bigotimes_M$ *T* $\bigotimes_M$ *P*))
using *b-gt-1 Pxz Px Pz*
by (*subst distributed-transform-integral*[*OF Pxyz - Pxz -*, **where** *T*=λ(*x*, *y*, *z*). (*x*, *z*)])
(*auto simp*: *split-beta' space-pair-measure*)
finally have *mi-eq*:
mutual-information b S P X Z = (∫ (*x*,*y*,*z*). *Pxyz* (*x*,*y*,*z*) ∗ *log b* (*Pxz* (*x*,*z*) / (*Px x* ∗ *Pz z*)) ∂(*S* $\bigotimes_M$ *T* $\bigotimes_M$ *P*)) .

have *ae1*: *AE x in S* $\bigotimes_M$ *T* $\bigotimes_M$ *P*. *Px* (*fst x*) = *0* ⟶ *Pxyz x* = *0*
by (*intro subdensity-real*[*of fst*, *OF - Pxyz Px*]) (*auto simp*: *space-pair-measure*)
moreover have *ae2*: *AE x in S* $\bigotimes_M$ *T* $\bigotimes_M$ *P*. *Pz* (*snd* (*snd x*)) = *0* ⟶ *Pxyz x* = *0*
by (*intro subdensity-real*[*of* λ*x*. *snd* (*snd x*), *OF - Pxyz Pz*]) (*auto simp*: *space-pair-measure*)
moreover have *ae3*: *AE x in S* $\bigotimes_M$ *T* $\bigotimes_M$ *P*. *Pxz* (*fst x*, *snd* (*snd x*)) = *0* ⟶ *Pxyz x* = *0*
by (*intro subdensity-real*[*of* λ*x*. (*fst x*, *snd* (*snd x*)), *OF - Pxyz Pxz*]) (*auto simp*: *space-pair-measure*)
moreover have *ae4*: *AE x in S* $\bigotimes_M$ *T* $\bigotimes_M$ *P*. *Pyz* (*snd x*) = *0* ⟶ *Pxyz x* = *0*
by (*intro subdensity-real*[*of snd*, *OF - Pxyz Pyz*]) (*auto simp*: *space-pair-measure*)
ultimately have *ae*: *AE x in S* $\bigotimes_M$ *T* $\bigotimes_M$ *P*.
Pxyz x ∗ *log b* (*Pxyz x* / (*Px* (*fst x*) ∗ *Pyz* (*snd x*))) −
Pxyz x ∗ *log b* (*Pxz* (*fst x*, *snd* (*snd x*)) / (*Px* (*fst x*) ∗ *Pz* (*snd* (*snd x*)))) =
Pxyz x ∗ *log b* (*Pxyz x* ∗ *Pz* (*snd* (*snd x*)) / (*Pxz* (*fst x*, *snd* (*snd x*)) ∗ *Pyz* (*snd x*)))
using *AE-space*
proof *eventually-elim*
case (*elim x*)
show *?case*
proof *cases*
assume *Pxyz x* ≠ *0*
with *elim* **have** *0* < *Px* (*fst x*) *0* < *Pz* (*snd* (*snd x*)) *0* < *Pxz* (*fst x*, *snd* (*snd x*))
0 < *Pyz* (*snd x*) *0* < *Pxyz x*
by (*auto simp*: *space-pair-measure less-le*)
then show *?thesis*
using *b-gt-1* **by** (*simp add*: *log-simps less-imp-le field-simps*)
qed *simp*
qed

with *I1 I2* **show** *?eq*
unfolding *conditional-mutual-information-def*
apply (*subst mi-eq*)
apply (*subst mutual-information-distr*[*OF S TP Px Px-nn Pyz - Pxyz*])
apply (*auto simp*: *space-pair-measure*)
apply (*subst Bochner-Integration.integral-diff*[*symmetric*])
apply (*auto intro*!: *integral-cong-AE simp*: *split-beta' simp del*: *Bochner-Integration.integral-diff*)
done

let *?P* = *density* (*S* $\bigotimes_M$ *T* $\bigotimes_M$ *P*) *Pxyz*
interpret *P*: *prob-space ?P*
unfolding *distributed-distr-eq-density*[*OF Pxyz*, *symmetric*]
by (*rule prob-space-distr*) *simp*

let *?Q* = *density* (*T* $\bigotimes_M$ *P*) *Pyz*
interpret *Q*: *prob-space ?Q*
unfolding *distributed-distr-eq-density*[*OF Pyz*, *symmetric*]
by (*rule prob-space-distr*) *simp*

let *?f* = λ(*x*, *y*, *z*). *Pxz* (*x*, *z*) * (*Pyz* (*y*, *z*) / *Pz z*) / *Pxyz* (*x*, *y*, *z*)

from *subdensity-real*[*of snd*, *OF - Pyz Pz - AE-I2 AE-I2*]
have *aeX1*: *AE x in T* $\bigotimes_M$ *P*. *Pz* (*snd x*) = *0* $\longrightarrow$ *Pyz x* = *0*
by (*auto simp*: *comp-def space-pair-measure*)
have *aeX2*: *AE x in T* $\bigotimes_M$ *P*. *0* $\leq$ *Pz* (*snd x*)
using *Pz* **by** (*intro TP.AE-pair-measure*) (*auto simp*: *comp-def*)

have *aeX3*: *AE y in T* $\bigotimes_M$ *P*. ($\int^+$ *x*. *ennreal* (*Pxz* (*x*, *snd y*)) ∂*S*) = *ennreal* (*Pz* (*snd y*))
using *Pz distributed-marginal-eq-joint2*[*OF P S Pz Pxz*]
by (*intro TP.AE-pair-measure*) *auto*

have ($\int^+$ *x*. *?f x* ∂*?P*) $\leq$ ($\int^+$ (*x*, *y*, *z*). *Pxz* (*x*, *z*) * (*Pyz* (*y*, *z*) / *Pz z*) ∂(*S* $\bigotimes_M$ *T* $\bigotimes_M$ *P*))
by (*subst nn-integral-density*)
(*auto intro*!: *nn-integral-mono simp*: *space-pair-measure ennreal-mult*[*symmetric*])
also have ... = ($\int^+$(*y*, *z*). ($\int^+$ *x*. *ennreal* (*Pxz* (*x*, *z*)) * *ennreal* (*Pyz* (*y*, *z*) / *Pz z*) ∂*S*) ∂(*T* $\bigotimes_M$ *P*))
by (*subst STP.nn-integral-snd*[*symmetric*])
(*auto simp add*: *split-beta' ennreal-mult*[*symmetric*] *space-pair-measure intro*!: *nn-integral-cong*)
also have ... = ($\int^+$*x*. *ennreal* (*Pyz x*) * *1* ∂*T* $\bigotimes_M$ *P*)
apply (*rule nn-integral-cong-AE*)
using *aeX1 aeX2 aeX3 AE-space*
apply *eventually-elim*
proof (*case-tac x*, *simp add*: *space-pair-measure*)
fix *a b* **assume** *Pz b* = *0* $\longrightarrow$ *Pyz* (*a*, *b*) = *0 a* $\in$ *space T* $\wedge$ *b* $\in$ *space P*
($\int^+$ *x*. *ennreal* (*Pxz* (*x*, *b*)) ∂*S*) = *ennreal* (*Pz b*)
then show ($\int^+$ *x*. *ennreal* (*Pxz* (*x*, *b*)) * *ennreal* (*Pyz* (*a*, *b*) / *Pz b*) ∂*S*) =

ennreal (*Pyz* (*a*, *b*))
by (*subst nn-integral-multc*) (*auto split*: *prod.split simp*: *ennreal-mult*[*symmetric*])
qed
also have ... = *1*
using *Q.emeasure-space-1 distributed-distr-eq-density*[*OF Pyz*]
by (*subst nn-integral-density*[*symmetric*]) *auto*
finally have *le1*: $(\int^+ x.\ ?f\ x\ \partial ?P) \leq 1$ **.**
also have ... $< \infty$ **by** *simp*
finally have *fin*: $(\int^+ x.\ ?f\ x\ \partial ?P) \neq \infty$ **by** *simp*

have *pos*: $(\int^+ x.\ ?f\ x\ \partial ?P) \neq 0$
apply (*subst nn-integral-density*)
apply (*simp-all add*: *split-beta'*)
proof
let *?g* = $\lambda x.$ *ennreal* (*Pxyz x*) * (*Pxz* (*fst x*, *snd* (*snd x*)) * *Pyz* (*snd* (*snd x*)) / (*Pz* (*snd* (*snd x*)) * *Pxyz x*))
assume $(\int^+ x.\ ?g\ x\ \partial(S \bigotimes_M T \bigotimes_M P)) = 0$
then have *AE x in* $S \bigotimes_M T \bigotimes_M P.$ *?g x* = *0*
by (*intro nn-integral-0-iff-AE*[*THEN iffD1*]) *auto*
then have *AE x in* $S \bigotimes_M T \bigotimes_M P.$ *Pxyz x* = *0*
using *ae1 ae2 ae3 ae4 AE-space*
by *eventually-elim* (*auto split*: *if-split-asm simp*: *mult-le-0-iff divide-le-0-iff space-pair-measure*)
then have $(\int^+ x.\ ennreal\ (Pxyz\ x)\ \partial S \bigotimes_M T \bigotimes_M P) = 0$
by (*subst nn-integral-cong-AE*[*of - λx. 0*]) *auto*
with *P.emeasure-space-1* **show** *False*
by (*subst* (*asm*) *emeasure-density*) (*auto cong*: *nn-integral-cong*)
qed

have *neg*: $(\int^+ x.\ -\ ?f\ x\ \partial ?P) = 0$
apply (*rule nn-integral-0-iff-AE*[*THEN iffD2*])
apply *simp*
apply (*subst AE-density*)
apply (*auto simp*: *space-pair-measure ennreal-neg*)
done

have *I3*: *integrable* $(S \bigotimes_M T \bigotimes_M P)$ $(\lambda(x, y, z).\ Pxyz\ (x, y, z) * log\ b\ (Pxyz\ (x, y, z) / (Pxz\ (x, z) * (Pyz\ (y,z) / Pz\ z))))$
apply (*rule integrable-cong-AE*[*THEN iffD1*, *OF - - - Bochner-Integration.integrable-diff*[*OF I1 I2*]])
using *ae*
apply (*auto simp*: *split-beta'*)
done

have $-\ log\ b\ 1 \leq -\ log\ b\ (integral^L\ ?P\ ?f)$
proof (*intro le-imp-neg-le log-le*[*OF b-gt-1*])
have *If*: *integrable ?P ?f*
unfolding *real-integrable-def*
proof (*intro conjI*)

from *neg* **show** $(\int^+ x.\ -\ ?f\ x\ \partial ?P) \neq \infty$
by *simp*
from *fin* **show** $(\int^+ x.\ ?f\ x\ \partial ?P) \neq \infty$
by *simp*
qed *simp*
then have $(\int^+ x.\ ?f\ x\ \partial ?P) = (\int x.\ ?f\ x\ \partial ?P)$
apply (*rule nn-integral-eq-integral*)
apply (*subst AE-density*)
apply *simp*
apply (*auto simp*: *space-pair-measure ennreal-neg*)
done
with *pos le1*
show $0 < (\int x.\ ?f\ x\ \partial ?P)\ (\int x.\ ?f\ x\ \partial ?P) \leq 1$
by (*simp-all add*: *one-ennreal.rep-eq zero-less-iff-neq-zero*[*symmetric*])
qed
also have $-\ log\ b\ (integral^L\ ?P\ ?f) \leq (\int\ x.\ -\ log\ b\ (?f\ x)\ \partial ?P)$
proof (*rule P.jensens-inequality*[**where** $a{=}0$ **and** $b{=}1$ **and** $I{=}\{0{<}..\}$])
show *AE x in ?P*. $?f\ x \in \{0{<}..\}$
unfolding *AE-density*[*OF distributed-borel-measurable*[*OF Pxyz*]]
using *ae1 ae2 ae3 ae4 AE-space*
by *eventually-elim* (*auto simp*: *space-pair-measure less-le*)
show *integrable ?P ?f*
unfolding *real-integrable-def*
using *fin neg* **by** (*auto simp*: *split-beta'*)
show *integrable ?P* $(\lambda x.\ -\ log\ b\ (?f\ x))$
apply (*subst integrable-real-density*)
apply *simp*
apply (*auto simp*: *space-pair-measure*) []
apply *simp*
apply (*rule integrable-cong-AE*[*THEN iffD1*, *OF - - - I3*])
apply *simp*
apply *simp*
using *ae1 ae2 ae3 ae4 AE-space*
apply *eventually-elim*
apply (*auto simp*: *log-divide-eq log-mult-eq zero-le-mult-iff zero-less-mult-iff zero-less-divide-iff field-simps*
less-le space-pair-measure)
done
qed (*auto simp*: *b-gt-1 minus-log-convex*)
also have $\ldots$ = *conditional-mutual-information b S T P X Y Z*
unfolding ‹*?eq*›
apply (*subst integral-real-density*)
apply *simp*
apply (*auto simp*: *space-pair-measure*) []
apply *simp*
apply (*intro integral-cong-AE*)
using *ae1 ae2 ae3 ae4*
apply (*auto simp*: *log-divide-eq zero-less-mult-iff zero-less-divide-iff field-simps*
space-pair-measure less-le)

done
finally show *?nonneg*
by *simp*
qed

lemma (**in** *information-space*)
fixes *Px* :: - ⇒ *real*
assumes *S*: *sigma-finite-measure S* **and** *T*: *sigma-finite-measure T* **and** *P*: *sigma-finite-measure P*
assumes *Fx*: *finite-entropy S X Px*
assumes *Fz*: *finite-entropy P Z Pz*
assumes *Fyz*: *finite-entropy* (*T* ⨂$_M$ *P*) (λ*x*. (*Y x*, *Z x*)) *Pyz*
assumes *Fxz*: *finite-entropy* (*S* ⨂$_M$ *P*) (λ*x*. (*X x*, *Z x*)) *Pxz*
assumes *Fxyz*: *finite-entropy* (*S* ⨂$_M$ *T* ⨂$_M$ *P*) (λ*x*. (*X x*, *Y x*, *Z x*)) *Pxyz*
shows *conditional-mutual-information-generic-eq′*: *conditional-mutual-information b S T P X Y Z*
= (∫ (*x*, *y*, *z*). *Pxyz* (*x*, *y*, *z*) ∗ *log b* (*Pxyz* (*x*, *y*, *z*) / (*Pxz* (*x*, *z*) ∗ (*Pyz* (*y*,*z*) / *Pz z*))) ∂(*S* ⨂$_M$ *T* ⨂$_M$ *P*)) (**is** *?eq*)
and *conditional-mutual-information-generic-nonneg′*: *0* ≤ *conditional-mutual-information b S T P X Y Z* (**is** *?nonneg*)
proof −
note *Px* = *Fx*[*THEN finite-entropy-distributed*, *measurable*]
note *Pz* = *Fz*[*THEN finite-entropy-distributed*, *measurable*]
note *Pyz* = *Fyz*[*THEN finite-entropy-distributed*, *measurable*]
note *Pxz* = *Fxz*[*THEN finite-entropy-distributed*, *measurable*]
note *Pxyz* = *Fxyz*[*THEN finite-entropy-distributed*, *measurable*]

note *Px-nn* = *Fx*[*THEN finite-entropy-nn*]
note *Pz-nn* = *Fz*[*THEN finite-entropy-nn*]
note *Pyz-nn* = *Fyz*[*THEN finite-entropy-nn*]
note *Pxz-nn* = *Fxz*[*THEN finite-entropy-nn*]
note *Pxyz-nn* = *Fxyz*[*THEN finite-entropy-nn*]

note *Px′* = *Fx*[*THEN finite-entropy-measurable*, *measurable*]
note *Pz′* = *Fz*[*THEN finite-entropy-measurable*, *measurable*]
note *Pyz′* = *Fyz*[*THEN finite-entropy-measurable*, *measurable*]
note *Pxz′* = *Fxz*[*THEN finite-entropy-measurable*, *measurable*]
note *Pxyz′* = *Fxyz*[*THEN finite-entropy-measurable*, *measurable*]

interpret *S*: *sigma-finite-measure S* **by** *fact*
interpret *T*: *sigma-finite-measure T* **by** *fact*
interpret *P*: *sigma-finite-measure P* **by** *fact*
interpret *TP*: *pair-sigma-finite T P* **..**
interpret *SP*: *pair-sigma-finite S P* **..**
interpret *ST*: *pair-sigma-finite S T* **..**
interpret *SPT*: *pair-sigma-finite S* ⨂$_M$ *P T* **..**
interpret *STP*: *pair-sigma-finite S T* ⨂$_M$ *P* **..**
interpret *TPS*: *pair-sigma-finite T* ⨂$_M$ *P S* **..**
have *TP*: *sigma-finite-measure* (*T* ⨂$_M$ *P*) **..**

have *SP*: *sigma-finite-measure* ($S \bigotimes_M P$) **..**

from *Pxz Pxyz* **have** *distr-eq*: *distr M* ($S \bigotimes_M P$) ($\lambda x.$ ($X\ x$, $Z\ x$)) =
distr (*distr M* ($S \bigotimes_M T \bigotimes_M P$) ($\lambda x.$ ($X\ x$, $Y\ x$, $Z\ x$))) ($S \bigotimes_M P$) ($\lambda(x, y, z).$ (x, z))
by (*simp add*: *distr-distr comp-def*)

have *mutual-information b S P X Z* =
($\int x.$ *Pxz x* $*$ *log b* (*Pxz x* / (*Px* (*fst x*) $*$ *Pz* (*snd x*))) ∂($S \bigotimes_M P$))
using *Px Px-nn Pz Pz-nn Pxz Pxz-nn*
by (*rule mutual-information-distr*[*OF S P*]) (*auto simp*: *space-pair-measure*)
also have ... = ($\int (x,y,z).$ *Pxyz* (*x*,*y*,*z*) $*$ *log b* (*Pxz* (*x*,*z*) / (*Px x* $*$ *Pz z*)) ∂($S \bigotimes_M T \bigotimes_M P$))
using *b-gt-1 Pxz Pxz-nn Pxyz Pxyz-nn*
by (*subst distributed-transform-integral*[*OF Pxyz - Pxz*, **where** *T*=$\lambda(x, y, z).$ (x, z)])
(*auto simp*: *split-beta'*)
finally have *mi-eq*:
mutual-information b S P X Z = ($\int (x,y,z).$ *Pxyz* (*x*,*y*,*z*) $*$ *log b* (*Pxz* (*x*,*z*) / (*Px x* $*$ *Pz z*)) ∂($S \bigotimes_M T \bigotimes_M P$)) **.**

have *ae1*: *AE x in* $S \bigotimes_M T \bigotimes_M P$. *Px* (*fst x*) = *0* $\longrightarrow$ *Pxyz x* = *0*
by (*intro subdensity-finite-entropy*[*of fst*, *OF - Fxyz Fx*]) *auto*
moreover have *ae2*: *AE x in* $S \bigotimes_M T \bigotimes_M P$. *Pz* (*snd* (*snd x*)) = *0* $\longrightarrow$ *Pxyz x* = *0*
by (*intro subdensity-finite-entropy*[*of* $\lambda x.$ *snd* (*snd x*), *OF - Fxyz Fz*]) *auto*
moreover have *ae3*: *AE x in* $S \bigotimes_M T \bigotimes_M P$. *Pxz* (*fst x*, *snd* (*snd x*)) = *0* $\longrightarrow$ *Pxyz x* = *0*
by (*intro subdensity-finite-entropy*[*of* $\lambda x.$ (*fst x*, *snd* (*snd x*)), *OF - Fxyz Fxz*]) *auto*
moreover have *ae4*: *AE x in* $S \bigotimes_M T \bigotimes_M P$. *Pyz* (*snd x*) = *0* $\longrightarrow$ *Pxyz x* = *0*
by (*intro subdensity-finite-entropy*[*of snd*, *OF - Fxyz Fyz*]) *auto*
ultimately have *ae*: *AE x in* $S \bigotimes_M T \bigotimes_M P$.
Pxyz x $*$ *log b* (*Pxyz x* / (*Px* (*fst x*) $*$ *Pyz* (*snd x*))) $-$
Pxyz x $*$ *log b* (*Pxz* (*fst x*, *snd* (*snd x*)) / (*Px* (*fst x*) $*$ *Pz* (*snd* (*snd x*)))) =
Pxyz x $*$ *log b* (*Pxyz x* $*$ *Pz* (*snd* (*snd x*)) / (*Pxz* (*fst x*, *snd* (*snd x*)) $*$ *Pyz* (*snd x*)))
using *AE-space*
proof *eventually-elim*
case (*elim x*)
show *?case*
proof *cases*
assume *Pxyz x* $\neq$ *0*
with *elim* **have** *0* $<$ *Px* (*fst x*) *0* $<$ *Pz* (*snd* (*snd x*)) *0* $<$ *Pxz* (*fst x*, *snd* (*snd x*))
0 $<$ *Pyz* (*snd x*) *0* $<$ *Pxyz x*
using *Px-nn Pz-nn Pxz-nn Pyz-nn Pxyz-nn*
by (*auto simp*: *space-pair-measure less-le*)

```
    then show ?thesis
      using b-gt-1 by (simp add: log-simps less-imp-le field-simps)
  qed simp
 qed

 have integrable (S ⨂M T ⨂M P)
   (λx. Pxyz x * log b (Pxyz x) − Pxyz x * log b (Px (fst x)) − Pxyz x * log b (Pyz
(snd x)))
   using finite-entropy-integrable[OF Fxyz]
   using finite-entropy-integrable-transform[OF Fx Pxyz Pxyz-nn, of fst]
   using finite-entropy-integrable-transform[OF Fyz Pxyz Pxyz-nn, of snd]
   by simp
 moreover have (λ(x, y, z). Pxyz (x, y, z) * log b (Pxyz (x, y, z) / (Px x * Pyz
(y, z)))) ∈ borel-measurable (S ⨂M T ⨂M P)
   using Pxyz Px Pyz by simp
 ultimately have I1: integrable (S ⨂M T ⨂M P) (λ(x, y, z). Pxyz (x, y, z) *
log b (Pxyz (x, y, z) / (Px x * Pyz (y, z))))
   apply (rule integrable-cong-AE-imp)
   using ae1 ae4 AE-space
   by eventually-elim
     (insert Px-nn Pyz-nn Pxyz-nn,
    auto simp: log-divide-eq log-mult-eq field-simps zero-less-mult-iff space-pair-measure
less-le)

 have integrable (S ⨂M T ⨂M P)
   (λx. Pxyz x * log b (Pxz (fst x, snd (snd x))) − Pxyz x * log b (Px (fst x)) −
Pxyz x * log b (Pz (snd (snd x))))
   using finite-entropy-integrable-transform[OF Fxz Pxyz Pxyz-nn, of λx. (fst x,
snd (snd x))]
   using finite-entropy-integrable-transform[OF Fx Pxyz Pxyz-nn, of fst]
   using finite-entropy-integrable-transform[OF Fz Pxyz Pxyz-nn, of snd ∘ snd]
   by simp
 moreover have (λ(x, y, z). Pxyz (x, y, z) * log b (Pxz (x, z) / (Px x * Pz z)))
∈ borel-measurable (S ⨂M T ⨂M P)
   using Pxyz Px Pz
   by auto
 ultimately have I2: integrable (S ⨂M T ⨂M P) (λ(x, y, z). Pxyz (x, y, z) *
log b (Pxz (x, z) / (Px x * Pz z)))
   apply (rule integrable-cong-AE-imp)
   using ae1 ae2 ae3 ae4 AE-space
   by eventually-elim
     (insert Px-nn Pz-nn Pxz-nn Pyz-nn Pxyz-nn,
         auto simp: log-divide-eq log-mult-eq field-simps zero-less-mult-iff less-le
space-pair-measure)

 from ae I1 I2 show ?eq
   unfolding conditional-mutual-information-def
   apply (subst mi-eq)
    apply (subst mutual-information-distr[OF S TP Px Px-nn Pyz Pyz-nn Pxyz
```

Pxyz-nn])
apply *simp*
apply *simp*
apply (*simp add*: *space-pair-measure*)
apply (*subst Bochner-Integration.integral-diff*[*symmetric*])
apply (*auto intro*!: *integral-cong-AE simp*: *split-beta' simp del*: *Bochner-Integration.integral-diff*)
done

let *?P* = *density* (*S* $\bigotimes_M$ *T* $\bigotimes_M$ *P*) *Pxyz*
interpret *P*: *prob-space ?P*
unfolding *distributed-distr-eq-density*[*OF Pxyz*, *symmetric*] **by** (*rule prob-space-distr*) *simp*

let *?Q* = *density* (*T* $\bigotimes_M$ *P*) *Pyz*
interpret *Q*: *prob-space ?Q*
unfolding *distributed-distr-eq-density*[*OF Pyz*, *symmetric*] **by** (*rule prob-space-distr*) *simp*

let *?f* = $\lambda(x, y, z)$. *Pxz* (*x*, *z*) * (*Pyz* (*y*, *z*) / *Pz z*) / *Pxyz* (*x*, *y*, *z*)

from *subdensity-finite-entropy*[*of snd*, *OF - Fyz Fz*]
have *aeX1*: *AE x in T* $\bigotimes_M$ *P*. *Pz* (*snd x*) = *0* $\longrightarrow$ *Pyz x* = *0* **by** (*auto simp*: *comp-def*)
have *aeX2*: *AE x in T* $\bigotimes_M$ *P*. *0* $\leq$ *Pz* (*snd x*)
using *Pz* **by** (*intro TP.AE-pair-measure*) (*auto intro*: *Pz-nn*)

have *aeX3*: *AE y in T* $\bigotimes_M$ *P*. ($\int^+$ *x*. *ennreal* (*Pxz* (*x*, *snd y*)) ∂S) = *ennreal* (*Pz* (*snd y*))
using *Pz distributed-marginal-eq-joint2*[*OF P S Pz Pxz*]
by (*intro TP.AE-pair-measure*) (*auto*)
have ($\int^+$ *x*. *?f x* ∂*?P*) $\leq$ ($\int^+$ (*x*, *y*, *z*). *Pxz* (*x*, *z*) * (*Pyz* (*y*, *z*) / *Pz z*) ∂(*S* $\bigotimes_M$ *T* $\bigotimes_M$ *P*))
using *Px-nn Pz-nn Pxz-nn Pyz-nn Pxyz-nn*
by (*subst nn-integral-density*)
(*auto intro*!: *nn-integral-mono simp*: *space-pair-measure ennreal-mult*[*symmetric*])
also have ... = ($\int^+$(*y*, *z*). $\int^+$ *x*. *ennreal* (*Pxz* (*x*, *z*)) * *ennreal* (*Pyz* (*y*, *z*) / *Pz z*) ∂S ∂T $\bigotimes_M$ *P*)
using *Px-nn Pz-nn Pxz-nn Pyz-nn Pxyz-nn*
by (*subst STP.nn-integral-snd*[*symmetric*])
(*auto simp add*: *split-beta' ennreal-mult*[*symmetric*] *space-pair-measure intro*!: *nn-integral-cong*)
also have ... = ($\int^+$*x*. *ennreal* (*Pyz x*) * *1* ∂T $\bigotimes_M$ *P*)
apply (*rule nn-integral-cong-AE*)
using *aeX1 aeX2 aeX3 AE-space*
apply *eventually-elim*
proof (*case-tac x*, *simp add*: *space-pair-measure*)
fix *a b* **assume** *Pz b* = *0* $\longrightarrow$ *Pyz* (*a*, *b*) = *0* *0* $\leq$ *Pz b* *a* $\in$ *space T* $\wedge$ *b* $\in$ *space P*
($\int^+$ *x*. *ennreal* (*Pxz* (*x*, *b*)) ∂S) = *ennreal* (*Pz b*)

then show $(\int^+ x.\ ennreal\ (Pxz\ (x,\ b)) * ennreal\ (Pyz\ (a,\ b)\ /\ Pz\ b)\ \partial S) = ennreal\ (Pyz\ (a,\ b))$
using *Pyz-nn*[*of* (*a*,*b*)]
by (*subst nn-integral-multc*) (*auto simp*: *space-pair-measure ennreal-mult*[*symmetric*])
qed
also have $\ldots = 1$
using *Q.emeasure-space-1 Pyz-nn distributed-distr-eq-density*[*OF Pyz*]
by (*subst nn-integral-density*[*symmetric*]) *auto*
finally have *le1*: $(\int^+ x.\ ?f\ x\ \partial ?P) \leq 1$ **.**
also have $\ldots < \infty$ **by** *simp*
finally have *fin*: $(\int^+ x.\ ?f\ x\ \partial ?P) \neq \infty$ **by** *simp*

have $(\int^+ x.\ ?f\ x\ \partial ?P) \neq 0$
using *Pxyz-nn*
apply (*subst nn-integral-density*)
apply (*simp-all add*: *split-beta′ ennreal-mult′*[*symmetric*] *cong*: *nn-integral-cong*)
proof
let $?g = \lambda x.\ ennreal\ (if\ Pxyz\ x = 0\ then\ 0\ else\ Pxz\ (fst\ x,\ snd\ (snd\ x)) * Pyz\ (snd\ x)\ /\ Pz\ (snd\ (snd\ x)))$
assume $(\int^+ x.\ ?g\ x\ \partial(S \bigotimes_M T \bigotimes_M P)) = 0$
then have $AE\ x\ in\ S \bigotimes_M T \bigotimes_M P.\ ?g\ x = 0$
by (*intro nn-integral-0-iff-AE*[*THEN iffD1*]) *auto*
then have $AE\ x\ in\ S \bigotimes_M T \bigotimes_M P.\ Pxyz\ x = 0$
using *ae1 ae2 ae3 ae4 AE-space*
by *eventually-elim*
(*insert Px-nn Pz-nn Pxz-nn Pyz-nn*,
auto split: *if-split-asm simp*: *mult-le-0-iff divide-le-0-iff space-pair-measure*)
then have $(\int^+ x.\ ennreal\ (Pxyz\ x)\ \partial S \bigotimes_M T \bigotimes_M P) = 0$
by (*subst nn-integral-cong-AE*[*of* - $\lambda x.\ 0$]) *auto*
with *P.emeasure-space-1* **show** *False*
by (*subst* (*asm*) *emeasure-density*) (*auto cong*: *nn-integral-cong*)
qed
then have *pos*: $0 < (\int^+ x.\ ?f\ x\ \partial ?P)$
by (*simp add*: *zero-less-iff-neq-zero*)

have *neg*: $(\int^+ x.\ -\ ?f\ x\ \partial ?P) = 0$
using *Pz-nn Pxz-nn Pyz-nn Pxyz-nn*
by (*intro nn-integral-0-iff-AE*[*THEN iffD2*])
(*auto simp*: *split-beta′ AE-density space-pair-measure intro*!: *AE-I2 ennreal-neg*)

have *I3*: *integrable* $(S \bigotimes_M T \bigotimes_M P)\ (\lambda(x,\ y,\ z).\ Pxyz\ (x,\ y,\ z) * log\ b\ (Pxyz\ (x,\ y,\ z)\ /\ (Pxz\ (x,\ z) * (Pyz\ (y,z)\ /\ Pz\ z))))$
apply (*rule integrable-cong-AE*[*THEN iffD1*, *OF - - - Bochner-Integration.integrable-diff*[*OF I1 I2*]])
using *ae*
apply (*auto simp*: *split-beta′*)
done

have $-$ *log b 1* $\leq -$ *log b* (*integral*^{L} *?P ?f*)
proof (*intro le-imp-neg-le log-le*[*OF b-gt-1*])
have *If*: *integrable ?P ?f*
unfolding *real-integrable-def*
proof (*intro conjI*)
from *neg* **show** $(\int^+ x.\ -\ ?f\ x\ \partial ?P) \neq \infty$
by *simp*
from *fin* **show** $(\int^+ x.\ ?f\ x\ \partial ?P) \neq \infty$
by *simp*
qed *simp*
then have $(\int^+ x.\ ?f\ x\ \partial ?P) = (\int x.\ ?f\ x\ \partial ?P)$
using *Pz-nn Pxz-nn Pyz-nn Pxyz-nn*
by (*intro nn-integral-eq-integral*)
(*auto simp*: *AE-density space-pair-measure*)
with *pos le1*
show $0 < (\int x.\ ?f\ x\ \partial ?P)\ (\int x.\ ?f\ x\ \partial ?P) \leq 1$
by (*simp-all add*:)
qed
also have $-$ *log b* (*integral*^{L} *?P ?f*) $\leq (\int\ x.\ -$ *log b* (*?f x*) $\partial ?P)$
proof (*rule P.jensens-inequality*[**where** *a=0* **and** *b=1* **and** *I=*{*0<..*}])
show *AE x in ?P. ?f x* $\in$ {*0<..*}
unfolding *AE-density*[*OF distributed-borel-measurable*[*OF Pxyz*]]
using *ae1 ae2 ae3 ae4 AE-space*
by *eventually-elim* (*insert Pxyz-nn Pyz-nn Pz-nn Pxz-nn*, *auto simp*: *space-pair-measure less-le*)
show *integrable ?P ?f*
unfolding *real-integrable-def*
using *fin neg* **by** (*auto simp*: *split-beta'*)
show *integrable ?P* ($\lambda x.\ -$ *log b* (*?f x*))
using *Pz-nn Pxz-nn Pyz-nn Pxyz-nn*
apply (*subst integrable-real-density*)
apply *simp*
apply *simp*
apply *simp*
apply (*rule integrable-cong-AE*[*THEN iffD1*, *OF - - - I3*])
apply *simp*
apply *simp*
using *ae1 ae2 ae3 ae4 AE-space*
apply *eventually-elim*
apply (*auto simp*: *log-divide-eq log-mult-eq zero-le-mult-iff zero-less-mult-iff zero-less-divide-iff field-simps space-pair-measure less-le*)
done
qed (*auto simp*: *b-gt-1 minus-log-convex*)
also have ... = *conditional-mutual-information b S T P X Y Z*
unfolding ‹*?eq*›
using *Pz-nn Pxz-nn Pyz-nn Pxyz-nn*
apply (*subst integral-real-density*)
apply *simp*
apply *simp*

apply *simp*
apply (*intro integral-cong-AE*)
using *ae1 ae2 ae3 ae4 AE-space*
apply (*auto simp*: *log-divide-eq zero-less-mult-iff zero-less-divide-iff field-simps space-pair-measure less-le*)
done
finally show *?nonneg*
by *simp*
qed

lemma (**in** *information-space*) *conditional-mutual-information-eq*:
assumes *Pz*: *simple-distributed M Z Pz*
assumes *Pyz*: *simple-distributed M* (λ*x*. (*Y x*, *Z x*)) *Pyz*
assumes *Pxz*: *simple-distributed M* (λ*x*. (*X x*, *Z x*)) *Pxz*
assumes *Pxyz*: *simple-distributed M* (λ*x*. (*X x*, *Y x*, *Z x*)) *Pxyz*
shows $\mathcal{I}(X ; Y \mid Z) =$
$(\sum (x, y, z) \in (\lambda x.\ (X\ x,\ Y\ x,\ Z\ x))\text{`}space\ M.\ Pxyz\ (x, y, z) * log\ b\ (Pxyz\ (x, y, z) / (Pxz\ (x, z) * (Pyz\ (y,z) / Pz\ z))))$
proof (*subst conditional-mutual-information-generic-eq*[*OF - - - - - simple-distributed*[*OF Pz*] *- simple-distributed-joint*[*OF Pyz*] *- simple-distributed-joint*[*OF Pxz*] *- simple-distributed-joint2*[*OF Pxyz*]])
note *simple-distributed-joint2-finite*[*OF Pxyz*, *simp*]
show *sigma-finite-measure* (*count-space* (*X* ‘ *space M*))
by (*simp add*: *sigma-finite-measure-count-space-finite*)
show *sigma-finite-measure* (*count-space* (*Y* ‘ *space M*))
by (*simp add*: *sigma-finite-measure-count-space-finite*)
show *sigma-finite-measure* (*count-space* (*Z* ‘ *space M*))
by (*simp add*: *sigma-finite-measure-count-space-finite*)
have *count-space* (*X* ‘ *space M*) $\bigotimes_M$ *count-space* (*Y* ‘ *space M*) $\bigotimes_M$ *count-space* (*Z* ‘ *space M*) =
count-space (*X*‘*space M* × *Y*‘*space M* × *Z*‘*space M*)
(**is** *?P* = *?C*)
by (*simp add*: *pair-measure-count-space*)

let *?Px* = λ*x*. *measure M* (*X* −‘ {*x*} ∩ *space M*)
have (λ*x*. (*X x*, *Z x*)) ∈ *measurable M* (*count-space* (*X* ‘ *space M*) $\bigotimes_M$ *count-space* (*Z* ‘ *space M*))
using *simple-distributed-joint*[*OF Pxz*] **by** (*rule distributed-measurable*)
from *measurable-comp*[*OF this measurable-fst*]
have *random-variable* (*count-space* (*X* ‘ *space M*)) *X*
by (*simp add*: *comp-def*)
then have *simple-function M X*
unfolding *simple-function-def* **by** (*auto simp*: *measurable-count-space-eq2*)
then have *simple-distributed M X ?Px*
by (*rule simple-distributedI*) (*auto simp*: *measure-nonneg*)
then show *distributed M* (*count-space* (*X* ‘ *space M*)) *X ?Px*
by (*rule simple-distributed*)

let *?f* = (λ*x*. *if x* ∈ (λ*x*. (*X x*, *Y x*, *Z x*)) ‘ *space M then Pxyz x else 0*)
let *?g* = (λ*x*. *if x* ∈ (λ*x*. (*Y x*, *Z x*)) ‘ *space M then Pyz x else 0*)
let *?h* = (λ*x*. *if x* ∈ (λ*x*. (*X x*, *Z x*)) ‘ *space M then Pxz x else 0*)
show
integrable ?P (λ(*x*, *y*, *z*). *?f* (*x*, *y*, *z*) ∗ *log b* (*?f* (*x*, *y*, *z*) / (*?Px x* ∗ *?g* (*y*, *z*))))
integrable ?P (λ(*x*, *y*, *z*). *?f* (*x*, *y*, *z*) ∗ *log b* (*?h* (*x*, *z*) / (*?Px x* ∗ *Pz z*)))
by (*auto intro*!: *integrable-count-space simp*: *pair-measure-count-space*)
let *?i* = λ*x y z*. *?f* (*x*, *y*, *z*) ∗ *log b* (*?f* (*x*, *y*, *z*) / (*?h* (*x*, *z*) ∗ (*?g* (*y*, *z*) / *Pz z*)))
let *?j* = λ*x y z*. *Pxyz* (*x*, *y*, *z*) ∗ *log b* (*Pxyz* (*x*, *y*, *z*) / (*Pxz* (*x*, *z*) ∗ (*Pyz* (*y*,*z*) / *Pz z*)))
have (λ(*x*, *y*, *z*). *?i x y z*) = (λ*x*. *if x* ∈ (λ*x*. (*X x*, *Y x*, *Z x*)) ‘ *space M then ?j* (*fst x*) (*fst* (*snd x*)) (*snd* (*snd x*)) *else 0*)
by (*auto intro*!: *ext*)
then show (∫ (*x*, *y*, *z*). *?i x y z* ∂*?P*) = (∑ (*x*, *y*, *z*)∈(λ*x*. (*X x*, *Y x*, *Z x*)) ‘ *space M*. *?j x y z*)
by (*auto intro*!: *sum.cong simp add*: ‹*?P* = *?C*› *lebesgue-integral-count-space-finite simple-distributed-finite sum.If-cases split-beta'*)
qed (*insert Pz Pyz Pxz Pxyz*, *auto intro*: *measure-nonneg*)

lemma (**in** *information-space*) *conditional-mutual-information-nonneg*:
assumes *X*: *simple-function M X* **and** *Y*: *simple-function M Y* **and** *Z*: *simple-function M Z*
shows *0* ≤ *I*(*X* ; *Y* | *Z*)
proof −
have [*simp*]: *count-space* (*X* ‘ *space M*) ⨂$_M$ *count-space* (*Y* ‘ *space M*) ⨂$_M$ *count-space* (*Z* ‘ *space M*) =
count-space (*X*‘*space M* × *Y*‘*space M* × *Z*‘*space M*)
by (*simp add*: *pair-measure-count-space X Y Z simple-functionD*)
note *sf* = *sigma-finite-measure-count-space-finite*[*OF simple-functionD*(*1*)]
note *sd* = *simple-distributedI*[*OF - - refl*]
note *sp* = *simple-function-Pair*
show *?thesis*
apply (*rule conditional-mutual-information-generic-nonneg*[*OF sf*[*OF X*] *sf*[*OF Y*] *sf*[*OF Z*]])
apply (*rule simple-distributed*[*OF sd*[*OF X*]])
apply *simp*
apply *simp*
apply (*rule simple-distributed*[*OF sd*[*OF Z*]])
apply *simp*
apply *simp*
apply (*rule simple-distributed-joint*[*OF sd*[*OF sp*[*OF Y Z*]]])
apply *simp*
apply *simp*
apply (*rule simple-distributed-joint*[*OF sd*[*OF sp*[*OF X Z*]]])
apply *simp*
apply *simp*
apply (*rule simple-distributed-joint2*[*OF sd*[*OF sp*[*OF X sp*[*OF Y Z*]]]])

apply *simp*
apply *simp*
apply (*auto intro*!: *integrable-count-space simp*: *X Y Z simple-functionD*)
done
qed

12.7 Conditional Entropy

definition (**in** *prob-space*)
conditional-entropy b S T X Y = − (∫ (*x*, *y*). *log b* (*enn2real* (*RN-deriv* (*S* ⨂$_M$ *T*) (*distr M* (*S* ⨂$_M$ *T*) (λ*x*. (*X x*, *Y x*))) (*x*, *y*)) /
enn2real (*RN-deriv T* (*distr M T Y*) *y*)) ∂*distr M* (*S* ⨂$_M$ *T*) (λ*x*. (*X x*, *Y x*)))

abbreviation (**in** *information-space*)
conditional-entropy-Pow (*H*′(- | -′)) **where**
H(*X* | *Y*) ≡ *conditional-entropy b* (*count-space* (*X*‘*space M*)) (*count-space* (*Y*‘*space M*)) *X Y*

lemma (**in** *information-space*) *conditional-entropy-generic-eq*:
fixes *Pxy* :: - ⇒ *real* **and** *Py* :: *'c* ⇒ *real*
assumes *S*: *sigma-finite-measure S* **and** *T*: *sigma-finite-measure T*
assumes *Py*[*measurable*]: *distributed M T Y Py* **and** *Py-nn*[*simp*]: ⋀*x*. *x* ∈ *space T* ⟹ *0* ≤ *Py x*
assumes *Pxy*[*measurable*]: *distributed M* (*S* ⨂$_M$ *T*) (λ*x*. (*X x*, *Y x*)) *Pxy*
and *Pxy-nn*[*simp*]: ⋀*x y*. *x* ∈ *space S* ⟹ *y* ∈ *space T* ⟹ *0* ≤ *Pxy* (*x*, *y*)
shows *conditional-entropy b S T X Y* = − (∫ (*x*, *y*). *Pxy* (*x*, *y*) ∗ *log b* (*Pxy* (*x*, *y*) / *Py y*) ∂(*S* ⨂$_M$ *T*))
proof −
interpret *S*: *sigma-finite-measure S* **by** *fact*
interpret *T*: *sigma-finite-measure T* **by** *fact*
interpret *ST*: *pair-sigma-finite S T* **..**

have [*measurable*]: *Py* ∈ *T* →$_M$ *borel*
using *Py Py-nn* **by** (*intro distributed-real-measurable*)
have *measurable-Pxy*[*measurable*]: *Pxy* ∈ (*S* ⨂$_M$ *T*) →$_M$ *borel*
using *Pxy Pxy-nn* **by** (*intro distributed-real-measurable*) (*auto simp*: *space-pair-measure*)

have *AE x in density* (*S* ⨂$_M$ *T*) (λ*x*. *ennreal* (*Pxy x*)). *Pxy x* = *enn2real* (*RN-deriv* (*S* ⨂$_M$ *T*) (*distr M* (*S* ⨂$_M$ *T*) (λ*x*. (*X x*, *Y x*))) *x*)
unfolding *AE-density*[*OF distributed-borel-measurable*, *OF Pxy*]
unfolding *distributed-distr-eq-density*[*OF Pxy*]
using *distributed-RN-deriv*[*OF Pxy*]
by *auto*
moreover
have *AE x in density* (*S* ⨂$_M$ *T*) (λ*x*. *ennreal* (*Pxy x*)). *Py* (*snd x*) = *enn2real* (*RN-deriv T* (*distr M T Y*) (*snd x*))
unfolding *AE-density*[*OF distributed-borel-measurable*, *OF Pxy*]
unfolding *distributed-distr-eq-density*[*OF Py*]

```
    apply (rule ST.AE-pair-measure)
    apply auto
    using distributed-RN-deriv[OF Py]
    apply auto
    done
  ultimately
  have conditional-entropy b S T X Y = - (∫ x. Pxy x * log b (Pxy x / Py (snd
x)) ∂(S ⨂M T))
    unfolding conditional-entropy-def neg-equal-iff-equal
    apply (subst integral-real-density[symmetric])
    apply (auto simp: distributed-distr-eq-density[OF Pxy] space-pair-measure
              intro!: integral-cong-AE)
    done
  then show ?thesis by (simp add: split-beta')
qed

lemma (in information-space) conditional-entropy-eq-entropy:
  fixes Px :: 'b ⇒ real and Py :: 'c ⇒ real
  assumes S: sigma-finite-measure S and T: sigma-finite-measure T
  assumes Py[measurable]: distributed M T Y Py
    and Py-nn[simp]: ⋀x. x ∈ space T ⟹ 0 ≤ Py x
  assumes Pxy[measurable]: distributed M (S ⨂M T) (λx. (X x, Y x)) Pxy
    and Pxy-nn[simp]: ⋀x y. x ∈ space S ⟹ y ∈ space T ⟹ 0 ≤ Pxy (x, y)
  assumes I1: integrable (S ⨂M T) (λx. Pxy x * log b (Pxy x))
  assumes I2: integrable (S ⨂M T) (λx. Pxy x * log b (Py (snd x)))
  shows conditional-entropy b S T X Y = entropy b (S ⨂M T) (λx. (X x, Y x))
- entropy b T Y
proof -
  interpret S: sigma-finite-measure S by fact
  interpret T: sigma-finite-measure T by fact
  interpret ST: pair-sigma-finite S T ..

  have [measurable]: Py ∈ T →M borel
    using Py Py-nn by (intro distributed-real-measurable)
  have measurable-Pxy[measurable]: Pxy ∈ (S ⨂M T) →M borel
   using Pxy Pxy-nn by (intro distributed-real-measurable) (auto simp: space-pair-measure)

  have entropy b T Y = - (∫ y. Py y * log b (Py y) ∂T)
    by (rule entropy-distr[OF Py Py-nn])
  also have ... = - (∫ (x,y). Pxy (x,y) * log b (Py y) ∂(S ⨂M T))
    using b-gt-1
    by (subst distributed-transform-integral[OF Pxy - Py, where T=snd])
       (auto intro!: Bochner-Integration.integral-cong simp: space-pair-measure)
  finally have e-eq: entropy b T Y = - (∫ (x,y). Pxy (x,y) * log b (Py y) ∂(S
⨂M T)) .

  have **: ⋀x. x ∈ space (S ⨂M T) ⟹ 0 ≤ Pxy x
    by (auto simp: space-pair-measure)
```

have *ae2*: *AE x in S* $\bigotimes_M$ *T. Py* (*snd x*) = *0* $\longrightarrow$ *Pxy x = 0*
by (*intro subdensity-real*[*of snd, OF - Pxy Py*])
(*auto intro*: *measurable-Pair simp*: *space-pair-measure*)
moreover have *ae4*: *AE x in S* $\bigotimes_M$ *T. 0* $\leq$ *Py* (*snd x*)
using *Py* **by** (*intro ST.AE-pair-measure*) (*auto simp*: *comp-def intro*!: *measurable-snd''*)
ultimately have *AE x in S* $\bigotimes_M$ *T. 0* $\leq$ *Pxy x* $\wedge$ *0* $\leq$ *Py* (*snd x*) $\wedge$
(*Pxy x = 0* $\vee$ (*Pxy x* $\neq$ *0* $\longrightarrow$ *0* $<$ *Pxy x* $\wedge$ *0* $<$ *Py* (*snd x*)))
using *AE-space* **by** *eventually-elim* (*auto simp*: *space-pair-measure less-le*)
then have *ae*: *AE x in S* $\bigotimes_M$ *T.*
Pxy x $*$ *log b* (*Pxy x*) $-$ *Pxy x* $*$ *log b* (*Py* (*snd x*)) = *Pxy x* $*$ *log b* (*Pxy x* / *Py* (*snd x*))
by *eventually-elim* (*auto simp*: *log-simps field-simps b-gt-1*)
have *conditional-entropy b S T X Y* =
$-$ ($\int$ *x. Pxy x* $*$ *log b* (*Pxy x*) $-$ *Pxy x* $*$ *log b* (*Py* (*snd x*)) ∂(*S* $\bigotimes_M$ *T*))
unfolding *conditional-entropy-generic-eq*[*OF S T Py Py-nn Pxy Pxy-nn, simplified*] *neg-equal-iff-equal*
apply (*intro integral-cong-AE*)
using *ae*
apply *auto*
done
also have ... = $-$ ($\int$ *x. Pxy x* $*$ *log b* (*Pxy x*) ∂(*S* $\bigotimes_M$ *T*)) $-$ $-$ ($\int$ *x.* *Pxy x* $*$ *log b* (*Py* (*snd x*)) ∂(*S* $\bigotimes_M$ *T*))
by (*simp add*: *Bochner-Integration.integral-diff*[*OF I1 I2*])
finally show *?thesis*
using *conditional-entropy-generic-eq*[*OF S T Py Py-nn Pxy Pxy-nn, simplified*] *entropy-distr*[*OF Pxy* $**$, *simplified*] *e-eq*
by (*simp add*: *split-beta'*)
qed

lemma (**in** *information-space*) *conditional-entropy-eq-entropy-simple*:
assumes *X*: *simple-function M X* **and** *Y*: *simple-function M Y*
shows $\mathcal{H}(X \mid Y)$ = *entropy b* (*count-space* (*X*‘*space M*) $\bigotimes_M$ *count-space* (*Y*‘*space M*)) (λ*x.* (*X x*, *Y x*)) $-$ $\mathcal{H}(Y)$
proof $-$
have *count-space* (*X* ‘ *space M*) $\bigotimes_M$ *count-space* (*Y* ‘ *space M*) = *count-space* (*X*‘*space M* $\times$ *Y*‘*space M*)
(**is** *?P = ?C*) **using** *X Y* **by** (*simp add*: *simple-functionD pair-measure-count-space*)
show *?thesis*
by (*rule conditional-entropy-eq-entropy sigma-finite-measure-count-space-finite simple-functionD X Y simple-distributed simple-distributedI*[*OF - - refl*] *simple-distributed-joint simple-function-Pair integrable-count-space measure-nonneg*)+
(*auto simp*: ‹*?P = ?C*› *measure-nonneg intro*!: *integrable-count-space simple-functionD X Y*)
qed

lemma (**in** *information-space*) *conditional-entropy-eq*:
assumes *Y*: *simple-distributed M Y Py*

assumes *XY*: *simple-distributed M* (λ*x*. (*X x*, *Y x*)) *Pxy*
shows $\mathcal{H}(X \mid Y) = -$ (∑ (*x*, *y*)∈(λ*x*. (*X x*, *Y x*)) ‘ *space M*. *Pxy* (*x*, *y*) ∗ *log b* (*Pxy* (*x*, *y*) / *Py y*))
proof (*subst conditional-entropy-generic-eq*[*OF* - - *simple-distributed*[*OF Y*] - *simple-distributed-joint*[*OF XY*]])
have *finite* ((λ*x*. (*X x*, *Y x*))‘*space M*)
using *XY* **unfolding** *simple-distributed-def* **by** *auto*
from *finite-imageI*[*OF this*, *of fst*]
have [*simp*]: *finite* (*X*‘*space M*)
by (*simp add*: *image-comp comp-def*)
note *Y*[*THEN simple-distributed-finite*, *simp*]
show *sigma-finite-measure* (*count-space* (*X* ‘ *space M*))
by (*simp add*: *sigma-finite-measure-count-space-finite*)
show *sigma-finite-measure* (*count-space* (*Y* ‘ *space M*))
by (*simp add*: *sigma-finite-measure-count-space-finite*)
let *?f* = (λ*x*. *if x* ∈ (λ*x*. (*X x*, *Y x*)) ‘ *space M then Pxy x else 0*)
have *count-space* (*X* ‘ *space M*) ⨂$_M$ *count-space* (*Y* ‘ *space M*) = *count-space* (*X*‘*space M* × *Y*‘*space M*)
(**is** *?P* = *?C*)
using *Y* **by** (*simp add*: *simple-distributed-finite pair-measure-count-space*)
have *eq*: (λ(*x*, *y*). *?f* (*x*, *y*) ∗ *log b* (*?f* (*x*, *y*) / *Py y*)) =
(λ*x*. *if x* ∈ (λ*x*. (*X x*, *Y x*)) ‘ *space M then Pxy x* ∗ *log b* (*Pxy x* / *Py* (*snd x*)) *else 0*)
by *auto*
from *Y* **show** − (∫ (*x*, *y*). *?f* (*x*, *y*) ∗ *log b* (*?f* (*x*, *y*) / *Py y*) ∂*?P*) =
− (∑ (*x*, *y*)∈(λ*x*. (*X x*, *Y x*)) ‘ *space M*. *Pxy* (*x*, *y*) ∗ *log b* (*Pxy* (*x*, *y*) / *Py y*))
by (*auto intro*!: *sum.cong simp add*: ‹*?P* = *?C*› *lebesgue-integral-count-space-finite simple-distributed-finite eq sum.If-cases split-beta*′)
qed (*insert Y XY*, *auto*)

lemma (**in** *information-space*) *conditional-mutual-information-eq-conditional-entropy*:
assumes *X*: *simple-function M X* **and** *Y*: *simple-function M Y*
shows $\mathcal{I}(X ; X \mid Y) = \mathcal{H}(X \mid Y)$
proof −
define *Py* **where** *Py x* = (*if x* ∈ *Y*‘*space M then measure M* (*Y* −‘ {*x*} ∩ *space M*) *else 0*) **for** *x*
define *Pxy* **where** *Pxy x* =
(*if x* ∈ (λ*x*. (*X x*, *Y x*))‘*space M then measure M* ((λ*x*. (*X x*, *Y x*)) −‘ {*x*} ∩ *space M*) *else 0*)
for *x*
define *Pxxy* **where** *Pxxy x* =
(*if x* ∈ (λ*x*. (*X x*, *X x*, *Y x*))‘*space M then measure M* ((λ*x*. (*X x*, *X x*, *Y x*)) −‘ {*x*} ∩ *space M*)
else 0)
for *x*
let *?M* = *X*‘*space M* × *X*‘*space M* × *Y*‘*space M*

note *XY* = *simple-function-Pair*[*OF X Y*]

note *XXY* = *simple-function-Pair*[*OF X XY*]
have *Py*: *simple-distributed M Y Py*
using *Y* **by** (*rule simple-distributedI*) (*auto simp*: *Py-def measure-nonneg*)
have *Pxy*: *simple-distributed M* (λ*x*. (*X x*, *Y x*)) *Pxy*
using *XY* **by** (*rule simple-distributedI*) (*auto simp*: *Pxy-def measure-nonneg*)
have *Pxxy*: *simple-distributed M* (λ*x*. (*X x*, *X x*, *Y x*)) *Pxxy*
using *XXY* **by** (*rule simple-distributedI*) (*auto simp*: *Pxxy-def measure-nonneg*)
have *eq*: (λ*x*. (*X x*, *X x*, *Y x*)) ' *space M* = (λ(*x*, *y*). (*x*, *x*, *y*)) ' (λ*x*. (*X x*, *Y x*)) ' *space M*
by *auto*
have *inj*: ⋀*A*. *inj-on* (λ(*x*, *y*). (*x*, *x*, *y*)) *A*
by (*auto simp*: *inj-on-def*)
have *Pxxy-eq*: ⋀*x y*. *Pxxy* (*x*, *x*, *y*) = *Pxy* (*x*, *y*)
by (*auto simp*: *Pxxy-def Pxy-def intro*!: *arg-cong*[**where** *f*=*prob*])
have *AE x in count-space* ((λ*x*. (*X x*, *Y x*))'*space M*). *Py* (*snd x*) = *0* ⟶ *Pxy x* = *0*
using *Py Pxy*
by (*intro subdensity-real*[*of snd*, *OF* - *Pxy*[*THEN simple-distributed*] *Py*[*THEN simple-distributed*]])
(*auto intro*: *measurable-Pair simp*: *AE-count-space*)
then show *?thesis*
apply (*subst conditional-mutual-information-eq*[*OF Py Pxy Pxy Pxxy*])
apply (*subst conditional-entropy-eq*[*OF Py Pxy*])
apply (*auto intro*!: *sum.cong simp*: *Pxxy-eq sum-negf*[*symmetric*] *eq sum.reindex*[*OF inj*]
log-simps zero-less-mult-iff zero-le-mult-iff field-simps mult-less-0-iff AE-count-space)
using *Py*[*THEN simple-distributed*] *Pxy*[*THEN simple-distributed*]
apply (*auto simp add*: *not-le AE-count-space less-le antisym*
simple-distributed-nonneg[*OF Py*] *simple-distributed-nonneg*[*OF Pxy*])
done
qed

lemma (**in** *information-space*) *conditional-entropy-nonneg*:
assumes *X*: *simple-function M X* **and** *Y*: *simple-function M Y* **shows** *0* ≤ ℋ(*X* | *Y*)
using *conditional-mutual-information-eq-conditional-entropy*[*OF X Y*] *conditional-mutual-information-nonne*
X X Y]
by *simp*

12.8 Equalities

lemma (**in** *information-space*) *mutual-information-eq-entropy-conditional-entropy-distr*:
fixes *Px* :: *'b* ⇒ *real* **and** *Py* :: *'c* ⇒ *real* **and** *Pxy* :: (*'b* × *'c*) ⇒ *real*
assumes *S*: *sigma-finite-measure S* **and** *T*: *sigma-finite-measure T*
assumes *Px*[*measurable*]: *distributed M S X Px*
and *Px-nn*[*simp*]: ⋀*x*. *x* ∈ *space S* ⟹ *0* ≤ *Px x*
and *Py*[*measurable*]: *distributed M T Y Py*
and *Py-nn*[*simp*]: ⋀*x*. *x* ∈ *space T* ⟹ *0* ≤ *Py x*

and *Pxy*[*measurable*]: *distributed M* ($S \bigotimes_M T$) ($\lambda x.\ (X\ x,\ Y\ x)$) *Pxy*
and *Pxy-nn*[*simp*]: $\bigwedge x\ y.\ x \in space\ S \Longrightarrow y \in space\ T \Longrightarrow 0 \leq Pxy\ (x,\ y)$
assumes *Ix*: *integrable*($S \bigotimes_M T$) ($\lambda x.\ Pxy\ x * log\ b\ (Px\ (fst\ x))$)
assumes *Iy*: *integrable*($S \bigotimes_M T$) ($\lambda x.\ Pxy\ x * log\ b\ (Py\ (snd\ x))$)
assumes *Ixy*: *integrable*($S \bigotimes_M T$) ($\lambda x.\ Pxy\ x * log\ b\ (Pxy\ x)$)
shows *mutual-information b S T X Y* = *entropy b S X* + *entropy b T Y* − *entropy b* ($S \bigotimes_M T$) ($\lambda x.\ (X\ x,\ Y\ x)$)
proof −
have [*measurable*]: $Px \in S \to_M borel$
using *Px Px-nn* **by** (*intro distributed-real-measurable*)
have [*measurable*]: $Py \in T \to_M borel$
using *Py Py-nn* **by** (*intro distributed-real-measurable*)
have *measurable-Pxy*[*measurable*]: $Pxy \in (S \bigotimes_M T) \to_M borel$
using *Pxy Pxy-nn* **by** (*intro distributed-real-measurable*) (*auto simp*: *space-pair-measure*)

have *X*: *entropy b S X* = − ($\int x.\ Pxy\ x * log\ b\ (Px\ (fst\ x))\ \partial(S \bigotimes_M T)$)
using *b-gt-1*
apply (*subst entropy-distr*[*OF Px Px-nn*], *simp*)
apply (*subst distributed-transform-integral*[*OF Pxy - Px*, **where** *T*=*fst*])
apply (*auto intro*!: *integral-cong simp*: *space-pair-measure*)
done

have *Y*: *entropy b T Y* = − ($\int x.\ Pxy\ x * log\ b\ (Py\ (snd\ x))\ \partial(S \bigotimes_M T)$)
using *b-gt-1*
apply (*subst entropy-distr*[*OF Py Py-nn*], *simp*)
apply (*subst distributed-transform-integral*[*OF Pxy - Py*, **where** *T*=*snd*])
apply (*auto intro*!: *integral-cong simp*: *space-pair-measure*)
done

interpret *S*: *sigma-finite-measure S* **by** *fact*
interpret *T*: *sigma-finite-measure T* **by** *fact*
interpret *ST*: *pair-sigma-finite S T* **..**
have *ST*: *sigma-finite-measure* ($S \bigotimes_M T$) **..**

have *XY*: *entropy b* ($S \bigotimes_M T$) ($\lambda x.\ (X\ x,\ Y\ x)$) = − ($\int x.\ Pxy\ x * log\ b\ (Pxy\ x)\ \partial(S \bigotimes_M T)$)
by (*subst entropy-distr*[*OF Pxy*]) (*auto intro*!: *integral-cong simp*: *space-pair-measure*)

have *AE x in* $S \bigotimes_M T$. $Px\ (fst\ x) = 0 \longrightarrow Pxy\ x = 0$
by (*intro subdensity-real*[*of fst*, *OF - Pxy Px*]) (*auto intro*: *measurable-Pair simp*: *space-pair-measure*)
moreover have *AE x in* $S \bigotimes_M T$. $Py\ (snd\ x) = 0 \longrightarrow Pxy\ x = 0$
by (*intro subdensity-real*[*of snd*, *OF - Pxy Py*]) (*auto intro*: *measurable-Pair simp*: *space-pair-measure*)
moreover have *AE x in* $S \bigotimes_M T$. $0 \leq Px\ (fst\ x)$
using *Px* **by** (*intro ST.AE-pair-measure*) (*auto simp*: *comp-def intro*!: *measurable-fst'*)
moreover have *AE x in* $S \bigotimes_M T$. $0 \leq Py\ (snd\ x)$
using *Py* **by** (*intro ST.AE-pair-measure*) (*auto simp*: *comp-def intro*!: *measur-*

able-snd'')
ultimately have *AE x in* $S \bigotimes_M T$. $Pxy\ x * log\ b\ (Pxy\ x) - Pxy\ x * log\ b\ (Px\ (fst\ x)) - Pxy\ x * log\ b\ (Py\ (snd\ x)) =$
$Pxy\ x * log\ b\ (Pxy\ x\ /\ (Px\ (fst\ x) * Py\ (snd\ x)))$
(**is** *AE x in -. ?f x = ?g x*)
using *AE-space*
proof *eventually-elim*
case (*elim x*)
show *?case*
proof *cases*
assume $Pxy\ x \neq 0$
with *elim* **have** $0 < Px\ (fst\ x)$ $0 < Py\ (snd\ x)$ $0 < Pxy\ x$
by (*auto simp*: *space-pair-measure less-le*)
then show *?thesis*
using *b-gt-1* **by** (*simp add*: *log-simps less-imp-le field-simps*)
qed *simp*
qed

have *entropy b S X + entropy b T Y - entropy b* $(S \bigotimes_M T)$ $(\lambda x.\ (X\ x,\ Y\ x)) = integral^L\ (S \bigotimes_M T)\ ?f$
unfolding *X Y XY*
apply (*subst Bochner-Integration.integral-diff*)
apply (*intro Bochner-Integration.integrable-diff Ixy Ix Iy*)+
apply (*subst Bochner-Integration.integral-diff*)
apply (*intro Ixy Ix Iy*)+
apply (*simp add*: *field-simps*)
done
also have $\ldots = integral^L\ (S \bigotimes_M T)\ ?g$
using ‹*AE x in -. ?f x = ?g x*› **by** (*intro integral-cong-AE*) *auto*
also have $\ldots$ = *mutual-information b S T X Y*
by (*rule mutual-information-distr*[*OF S T Px - Py - Pxy - , symmetric*])
(*auto simp*: *space-pair-measure*)
finally show *?thesis* **..**
qed

lemma (**in** *information-space*) *mutual-information-eq-entropy-conditional-entropy'*:
fixes $Px :: 'b \Rightarrow real$ **and** $Py :: 'c \Rightarrow real$ **and** $Pxy :: ('b \times 'c) \Rightarrow real$
assumes *S*: *sigma-finite-measure S* **and** *T*: *sigma-finite-measure T*
assumes *Px*: *distributed M S X Px* $\bigwedge x.\ x \in space\ S \Longrightarrow 0 \leq Px\ x$
and *Py*: *distributed M T Y Py* $\bigwedge x.\ x \in space\ T \Longrightarrow 0 \leq Py\ x$
assumes *Pxy*: *distributed M* $(S \bigotimes_M T)$ $(\lambda x.\ (X\ x,\ Y\ x))$ *Pxy*
$\bigwedge x.\ x \in space\ (S \bigotimes_M T) \Longrightarrow 0 \leq Pxy\ x$
assumes *Ix*: *integrable*$(S \bigotimes_M T)$ $(\lambda x.\ Pxy\ x * log\ b\ (Px\ (fst\ x)))$
assumes *Iy*: *integrable*$(S \bigotimes_M T)$ $(\lambda x.\ Pxy\ x * log\ b\ (Py\ (snd\ x)))$
assumes *Ixy*: *integrable*$(S \bigotimes_M T)$ $(\lambda x.\ Pxy\ x * log\ b\ (Pxy\ x))$
shows *mutual-information b S T X Y = entropy b S X - conditional-entropy b S T X Y*
using
mutual-information-eq-entropy-conditional-entropy-distr[*OF S T Px Py Pxy Ix*

Iy Ixy]
conditional-entropy-eq-entropy[*OF S T Py Pxy Ixy Iy*]
by (*simp add*: *space-pair-measure*)

lemma (**in** *information-space*) *mutual-information-eq-entropy-conditional-entropy*:
assumes *sf-X*: *simple-function M X* **and** *sf-Y*: *simple-function M Y*
shows $\mathcal{I}(X\ ;\ Y) = \mathcal{H}(X) - \mathcal{H}(X \mid Y)$
proof −
have *X*: *simple-distributed M X* (λ*x*. *measure M* (*X* −‘ {*x*} ∩ *space M*))
using *sf-X* **by** (*rule simple-distributedI*) (*auto simp*: *measure-nonneg*)
have *Y*: *simple-distributed M Y* (λ*x*. *measure M* (*Y* −‘ {*x*} ∩ *space M*))
using *sf-Y* **by** (*rule simple-distributedI*) (*auto simp*: *measure-nonneg*)
have *sf-XY*: *simple-function M* (λ*x*. (*X x*, *Y x*))
using *sf-X sf-Y* **by** (*rule simple-function-Pair*)
then have *XY*: *simple-distributed M* (λ*x*. (*X x*, *Y x*)) (λ*x*. *measure M* ((λ*x*. (*X x*, *Y x*)) −‘ {*x*} ∩ *space M*))
by (*rule simple-distributedI*) (*auto simp*: *measure-nonneg*)
from *simple-distributed-joint-finite*[*OF this*, *simp*]
have *eq*: *count-space* (*X* ‘ *space M*) ⨂$_M$ *count-space* (*Y* ‘ *space M*) = *count-space* (*X* ‘ *space M* × *Y* ‘ *space M*)
by (*simp add*: *pair-measure-count-space*)

have $\mathcal{I}(X\ ;\ Y) = \mathcal{H}(X) + \mathcal{H}(Y) -$ *entropy b* (*count-space* (*X*‘*space M*) ⨂$_M$ *count-space* (*Y*‘*space M*)) (λ*x*. (*X x*, *Y x*))
using *sigma-finite-measure-count-space-finite*
sigma-finite-measure-count-space-finite
simple-distributed[*OF X*] *measure-nonneg simple-distributed*[*OF Y*] *measure-nonneg simple-distributed-joint*[*OF XY*]
by (*rule mutual-information-eq-entropy-conditional-entropy-distr*)
(*auto simp*: *eq integrable-count-space measure-nonneg*)
then show *?thesis*
unfolding *conditional-entropy-eq-entropy-simple*[*OF sf-X sf-Y*] **by** *simp*
qed

lemma (**in** *information-space*) *mutual-information-nonneg-simple*:
assumes *sf-X*: *simple-function M X* **and** *sf-Y*: *simple-function M Y*
shows $0 \le \mathcal{I}(X\ ;\ Y)$
proof −
have *X*: *simple-distributed M X* (λ*x*. *measure M* (*X* −‘ {*x*} ∩ *space M*))
using *sf-X* **by** (*rule simple-distributedI*) (*auto simp*: *measure-nonneg*)
have *Y*: *simple-distributed M Y* (λ*x*. *measure M* (*Y* −‘ {*x*} ∩ *space M*))
using *sf-Y* **by** (*rule simple-distributedI*) (*auto simp*: *measure-nonneg*)

have *sf-XY*: *simple-function M* (λ*x*. (*X x*, *Y x*))
using *sf-X sf-Y* **by** (*rule simple-function-Pair*)
then have *XY*: *simple-distributed M* (λ*x*. (*X x*, *Y x*)) (λ*x*. *measure M* ((λ*x*. (*X x*, *Y x*)) −‘ {*x*} ∩ *space M*))
by (*rule simple-distributedI*) (*auto simp*: *measure-nonneg*)

from *simple-distributed-joint-finite*[*OF this*, *simp*]
have *eq*: *count-space* (*X* ' *space M*) $\bigotimes_M$ *count-space* (*Y* ' *space M*) = *count-space* (*X* ' *space M* × *Y* ' *space M*)
by (*simp add*: *pair-measure-count-space*)

show *?thesis*
by (*rule mutual-information-nonneg*[*OF - - simple-distributed*[*OF X*] *- simple-distributed*[*OF Y*] *- simple-distributed-joint*[*OF XY*]])
(*simp-all add*: *eq integrable-count-space sigma-finite-measure-count-space-finite measure-nonneg*)
qed

lemma (**in** *information-space*) *conditional-entropy-less-eq-entropy*:
assumes *X*: *simple-function M X* **and** *Z*: *simple-function M Z*
shows $\mathcal{H}(X \mid Z) \leq \mathcal{H}(X)$
proof −
have $0 \leq \mathcal{I}(X ; Z)$ **using** *X Z* **by** (*rule mutual-information-nonneg-simple*)
also have $\mathcal{I}(X ; Z) = \mathcal{H}(X) - \mathcal{H}(X \mid Z)$ **using** *mutual-information-eq-entropy-conditional-entropy*[*OF assms*] **.**
finally show *?thesis* **by** *auto*
qed

lemma (**in** *information-space*)
fixes *Px* :: *'b* ⇒ *real* **and** *Py* :: *'c* ⇒ *real* **and** *Pxy* :: (*'b* × *'c*) ⇒ *real*
assumes *S*: *sigma-finite-measure S* **and** *T*: *sigma-finite-measure T*
assumes *Px*: *finite-entropy S X Px* **and** *Py*: *finite-entropy T Y Py*
assumes *Pxy*: *finite-entropy* (*S* $\bigotimes_M$ *T*) (λ*x*. (*X x*, *Y x*)) *Pxy*
shows *conditional-entropy b S T X Y* ≤ *entropy b S X*
proof −

have *0* ≤ *mutual-information b S T X Y*
by (*rule mutual-information-nonneg'*) *fact+*
also have ... = *entropy b S X* − *conditional-entropy b S T X Y*
apply (*rule mutual-information-eq-entropy-conditional-entropy'*)
using *assms*
by (*auto intro*!: *finite-entropy-integrable finite-entropy-distributed*
finite-entropy-integrable-transform[*OF Px*]
finite-entropy-integrable-transform[*OF Py*]
intro: *finite-entropy-nn*)
finally show *?thesis* **by** *auto*
qed

lemma (**in** *information-space*) *entropy-chain-rule*:
assumes *X*: *simple-function M X* **and** *Y*: *simple-function M Y*
shows $\mathcal{H}(\lambda x.\ (X\ x,\ Y\ x)) = \mathcal{H}(X) + \mathcal{H}(Y|X)$
proof −
note *XY* = *simple-distributedI*[*OF simple-function-Pair*[*OF X Y*] *measure-nonneg refl*]
note *YX* = *simple-distributedI*[*OF simple-function-Pair*[*OF Y X*] *measure-nonneg*

refl]
note *simple-distributed-joint-finite*[*OF this*, *simp*]
let *?f* = λ*x*. *prob* ((λ*x*. (*X x*, *Y x*)) −‘ {*x*} ∩ *space M*)
let *?g* = λ*x*. *prob* ((λ*x*. (*Y x*, *X x*)) −‘ {*x*} ∩ *space M*)
let *?h* = λ*x*. *if x* ∈ (λ*x*. (*Y x*, *X x*)) ‘ *space M then prob* ((λ*x*. (*Y x*, *X x*)) −‘ {*x*} ∩ *space M*) *else 0*
have $\mathcal{H}$(λ*x*. (*X x*, *Y x*)) = − (∑ *x*∈(λ*x*. (*X x*, *Y x*)) ‘ *space M*. *?f x* ∗ *log b* (*?f x*))
using *XY* **by** (*rule entropy-simple-distributed*)
also have ... = − (∑ *x*∈(λ(*x*, *y*). (*y*, *x*)) ‘ (λ*x*. (*X x*, *Y x*)) ‘ *space M*. *?g x* ∗ *log b* (*?g x*))
by (*subst* (*2*) *sum.reindex*) (*auto simp*: *inj-on-def intro*!: *sum.cong arg-cong*[**where** *f*=λ*A*. *prob A* ∗ *log b* (*prob A*)])
also have ... = − (∑ *x*∈(λ*x*. (*Y x*, *X x*)) ‘ *space M*. *?h x* ∗ *log b* (*?h x*))
by (*auto intro*!: *sum.cong*)
also have ... = *entropy b* (*count-space* (*Y* ‘ *space M*) ⨂$_M$ *count-space* (*X* ‘ *space M*)) (λ*x*. (*Y x*, *X x*))
by (*subst entropy-distr*[*OF simple-distributed-joint*[*OF YX*]])
(*auto simp*: *pair-measure-count-space sigma-finite-measure-count-space-finite lebesgue-integral-count-space-finite*
cong del: *sum.cong-simp intro*!: *sum.mono-neutral-left measure-nonneg*)
finally have $\mathcal{H}$(λ*x*. (*X x*, *Y x*)) = *entropy b* (*count-space* (*Y* ‘ *space M*) ⨂$_M$ *count-space* (*X* ‘ *space M*)) (λ*x*. (*Y x*, *X x*)) .
then show *?thesis*
unfolding *conditional-entropy-eq-entropy-simple*[*OF Y X*] **by** *simp*
qed

lemma (**in** *information-space*) *entropy-partition*:
assumes *X*: *simple-function M X*
shows $\mathcal{H}(X) = \mathcal{H}(f \circ X) + \mathcal{H}(X|f \circ X)$
proof −
note *fX* = *simple-function-compose*[*OF X*, *of f*]
have *eq*: (λ*x*. ((*f* ∘ *X*) *x*, *X x*)) ‘ *space M* = (λ*x*. (*f x*, *x*)) ‘ *X* ‘ *space M* **by** *auto*
have *inj*: ⋀*A*. *inj-on* (λ*x*. (*f x*, *x*)) *A*
by (*auto simp*: *inj-on-def*)
show *?thesis*
apply (*subst entropy-chain-rule*[*symmetric*, *OF fX X*])
apply (*subst entropy-simple-distributed*[*OF simple-distributedI*[*OF simple-function-Pair*[*OF fX X*] *measure-nonneg refl*]])
apply (*subst entropy-simple-distributed*[*OF simple-distributedI*[*OF X measure-nonneg refl*]])
unfolding *eq*
apply (*subst sum.reindex*[*OF inj*])
apply (*auto intro*!: *sum.cong arg-cong*[**where** *f*=λ*A*. *prob A* ∗ *log b* (*prob A*)])
done
qed

corollary (**in** *information-space*) *entropy-data-processing*:
assumes *X*: *simple-function M X* **shows** $\mathcal{H}(f \circ X) \leq \mathcal{H}(X)$

proof −
note *fX* = *simple-function-compose*[*OF X*, *of f*]
from *X* **have** $\mathcal{H}(X) = \mathcal{H}(f \circ X) + \mathcal{H}(X|f \circ X)$ **by** (*rule entropy-partition*)
then show $\mathcal{H}(f \circ X) \leq \mathcal{H}(X)$
by (*simp only*: *conditional-entropy-nonneg* [*OF X fX*] *le-add-same-cancel1*)
qed

corollary (**in** *information-space*) *entropy-of-inj*:
assumes *X*: *simple-function M X* **and** *inj*: *inj-on f* (*X*'*space M*)
shows $\mathcal{H}(f \circ X) = \mathcal{H}(X)$
proof (*rule antisym*)
show $\mathcal{H}(f \circ X) \leq \mathcal{H}(X)$ **using** *entropy-data-processing*[*OF X*] **.**
next
have *sf*: *simple-function M* (*f* ∘ *X*)
using *X* **by** *auto*
have $\mathcal{H}(X) = \mathcal{H}$(*the-inv-into* (*X*'*space M*) *f* ∘ (*f* ∘ *X*))
unfolding *o-assoc*
apply (*subst entropy-simple-distributed*[*OF simple-distributedI*[*OF X measure-nonneg refl*]])
apply (*subst entropy-simple-distributed*[*OF simple-distributedI*[*OF simple-function-compose*[*OF X*]], **where** *f*=λ*x*. *prob* (*X* −' {*x*} ∩ *space M*)])
apply (*auto intro*!: *sum.cong arg-cong*[**where** *f*=*prob*] *image-eqI simp*: *the-inv-into-f-f*[*OF inj*] *comp-def measure-nonneg*)
done
also have ... ≤ $\mathcal{H}(f \circ X)$
using *entropy-data-processing*[*OF sf*] **.**
finally show $\mathcal{H}(X) \leq \mathcal{H}(f \circ X)$ **.**
qed

end

13 Properties of Various Distributions

theory *Distributions*
imports *Convolution Information*
begin

lemma (**in** *prob-space*) *distributed-affine*:
fixes *f* :: *real* ⇒ *ennreal*
assumes *f*: *distributed M lborel X f*
assumes *c*: *c* ≠ *0*
shows *distributed M lborel* (λ*x*. *t* + *c* ∗ *X x*) (λ*x*. *f* ((*x* − *t*) / *c*) / |*c*|)
unfolding *distributed-def*
proof *safe*
have [*measurable*]: *f* ∈ *borel-measurable borel*
using *f* **by** (*simp add*: *distributed-def*)
have [*measurable*]: *X* ∈ *borel-measurable M*
using *f* **by** (*simp add*: *distributed-def*)

```
  show (λx. f ((x − t) / c) / |c|) ∈ borel-measurable lborel
    by simp
  show random-variable lborel (λx. t + c * X x)
    by simp

  have eq: ennreal |c| * (f x / ennreal |c|) = f x for x
    using c
    by (cases f x)
       (auto simp: divide-ennreal ennreal-mult[symmetric] ennreal-top-divide en-
nreal-mult-top)

  have density lborel f = distr M lborel X
    using f by (simp add: distributed-def)
  with c show distr M lborel (λx. t + c * X x) = density lborel (λx. f ((x − t) /
c) / ennreal |c|)
    by (subst (2) lborel-real-affine[where c=c and t=t])
       (simp-all add: density-density-eq density-distr distr-distr field-simps eq cong:
distr-cong)
qed

lemma (in prob-space) distributed-affineI:
  fixes f :: real ⇒ ennreal and c :: real
  assumes f: distributed M lborel (λx. (X x − t) / c) (λx. |c| * f (x * c + t))
  assumes c: c ≠ 0
  shows distributed M lborel X f
proof −
  have eq: f x * ennreal |c| / ennreal |c| = f x for x
    using c by (simp add: ennreal-times-divide[symmetric])

  show ?thesis
    using distributed-affine[OF f c, where t=t] c
    by (simp add: field-simps eq)
qed

lemma (in prob-space) distributed-AE2:
  assumes [measurable]: distributed M N X f Measurable.pred N P
  shows (AE x in M. P (X x)) ⟷ (AE x in N. 0 < f x ⟶ P x)
proof −
  have (AE x in M. P (X x)) ⟷ (AE x in distr M N X. P x)
    by (simp add: AE-distr-iff)
  also have ... ⟷ (AE x in density N f. P x)
    unfolding distributed-distr-eq-density[OF assms(1)] ..
  also have ... ⟷ (AE x in N. 0 < f x ⟶ P x)
    by (rule AE-density) simp
  finally show ?thesis .
qed
```

13.1 Erlang

lemma *nn-intergal-power-times-exp-Icc*:
assumes [*arith*]: $0 \leq a$
shows $(\int^+ x.\ ennreal\ (x\hat{}k * exp\ (-x)) * indicator\ \{0\ ..\ a\}\ x\ \partial lborel) =$
$(1 - (\sum n{\leq}k.\ (a\hat{}n * exp\ (-a))\ /\ fact\ n)) * fact\ k$ (**is** *?I* = -)
proof −
let *?f* = $\lambda k\ x.\ x\hat{}k * exp\ (-x)\ /\ fact\ k$
let *?F* = $\lambda k\ x.\ -\ (\sum n{\leq}k.\ (x\hat{}n * exp\ (-x))\ /\ fact\ n)$
have *?I* $*$ (*inverse* (*real-of-nat* (*fact k*))) =
$(\int^+ x.\ ennreal\ (x\hat{}k * exp\ (-x)) * indicator\ \{0\ ..\ a\}\ x * (inverse\ (real\text{-}of\text{-}nat$
(*fact k*))) $\partial lborel$)
by (*intro nn-integral-multc*[*symmetric*]) *auto*
also have ... = $(\int^+ x.\ ennreal\ (?f\ k\ x) * indicator\ \{0\ ..\ a\}\ x\ \partial lborel)$
by (*intro nn-integral-cong*)
(*simp add*: *field-simps ennreal-mult'*[*symmetric*] *indicator-mult-ennreal*)
also have ... = *ennreal* (*?F k a* − *?F k 0*)
proof (*rule nn-integral-FTC-Icc*)
fix *x* **assume** $x \in \{0..a\}$
show *DERIV* (*?F k*) *x* :> *?f k x*
proof(*induction k*)
case *0* **show** *?case* **by** (*auto intro*!: *derivative-eq-intros*)
next
case (*Suc k*)
have *DERIV* ($\lambda x.$ *?F k x* − ($x\hat{}$*Suc k* $*$ *exp* (−*x*)) / *fact* (*Suc k*)) *x* :>
?f k x − ((*real* (*Suc k*) − *x*) $*$ *x* ^ *k* $*$ *exp* (− *x*)) / (*fact* (*Suc k*))
by (*intro DERIV-diff Suc*)
(*auto intro*!: *derivative-eq-intros simp del*: *fact-Suc power-Suc*
simp add: *field-simps power-Suc*[*symmetric*])
also have ($\lambda x.$ *?F k x* − ($x\hat{}$*Suc k* $*$ *exp* (−*x*)) / *fact* (*Suc k*)) = *?F* (*Suc k*)
by *simp*
also have *?f k x* − ((*real* (*Suc k*) − *x*) $*$ *x* ^ *k* $*$ *exp* (− *x*)) / (*fact* (*Suc k*))
= *?f* (*Suc k*) *x*
by (*auto simp*: *field-simps simp del*: *fact-Suc*)
(*simp-all add*: *of-nat-Suc field-simps*)
finally show *?case* **.**
qed
qed *auto*
also have ... = *ennreal* (1 − ($\sum n{\leq}k.\ (a\hat{}n * exp\ (-a))\ /\ fact\ n$))
by (*auto simp*: *power-0-left if-distrib*[**where** *f*=$\lambda x.\ x\ /\ a$ **for** *a*] *sum.If-cases*)
also have ... = *ennreal* ((1 − ($\sum n{\leq}k.\ (a\hat{}n * exp\ (-a))\ /\ fact\ n$)) $*$ *fact k*) $*$
ennreal (*inverse* (*fact k*))
by (*subst ennreal-mult''*[*symmetric*]) (*auto intro*!: *arg-cong*[**where** *f*=*ennreal*])
finally show *?thesis*
by (*auto simp add*: *mult-right-ennreal-cancel le-less*)
qed

lemma *nn-intergal-power-times-exp-Ici*:
shows $(\int^+ x.\ ennreal\ (x\hat{}k * exp\ (-x)) * indicator\ \{0\ ..\}\ x\ \partial lborel) = real\text{-}of\text{-}nat$
(*fact k*)

proof (*rule LIMSEQ-unique*)
let *?X* = λ*n*. ∫⁺ *x*. *ennreal* (*x*^*k* ∗ *exp* (−*x*)) ∗ *indicator* {*0* .. *real n*} *x* ∂*lborel*
show *?X* ⟶ (∫⁺*x*. *ennreal* (*x*^*k* ∗ *exp* (−*x*)) ∗ *indicator* {*0* ..} *x* ∂*lborel*)
apply (*intro nn-integral-LIMSEQ*)
apply (*auto simp*: *incseq-def le-fun-def eventually-sequentially*
split: *split-indicator intro*!: *tendsto-eventually*)
apply (*metis nat-ceiling-le-eq*)
done

have ((λ*x*::*real*. (*1* − (∑ *n*≤*k*. (*x* ^ *n* / *exp x*) / (*fact n*))) ∗ *fact k*) ⟶
(*1* − (∑ *n*≤*k*. *0* / (*fact n*))) ∗ *fact k*) *at-top*
by (*intro tendsto-intros tendsto-power-div-exp-0*) *simp*
then show *?X* ⟶ *real-of-nat* (*fact k*)
by (*subst nn-intergal-power-times-exp-Icc*)
(*auto simp*: *exp-minus field-simps intro*!: *filterlim-compose*[*OF - filterlim-real-sequentially*])
qed

definition *erlang-density* :: *nat* ⇒ *real* ⇒ *real* ⇒ *real* **where**
erlang-density k l x = (*if x* < *0 then 0 else* (*l*^(*Suc k*) ∗ *x*^*k* ∗ *exp* (− *l* ∗ *x*)) / *fact k*)

definition *erlang-CDF* :: *nat* ⇒ *real* ⇒ *real* ⇒ *real* **where**
erlang-CDF k l x = (*if x* < *0 then 0 else 1* − (∑ *n*≤*k*. ((*l* ∗ *x*)^*n* ∗ *exp* (− *l* ∗ *x*) / *fact n*)))

lemma *erlang-density-nonneg*[*simp*]: *0* ≤ *l* ⟹ *0* ≤ *erlang-density k l x*
by (*simp add*: *erlang-density-def*)

lemma *borel-measurable-erlang-density*[*measurable*]: *erlang-density k l* ∈ *borel-measurable borel*
by (*auto simp add*: *erlang-density-def*[*abs-def*])

lemma *erlang-CDF-transform*: *0* < *l* ⟹ *erlang-CDF k l a* = *erlang-CDF k 1* (*l* ∗ *a*)
by (*auto simp add*: *erlang-CDF-def mult-less-0-iff*)

lemma *erlang-CDF-nonneg*[*simp*]: **assumes** *0* < *l* **shows** *0* ≤ *erlang-CDF k l x*
unfolding *erlang-CDF-def*
proof (*clarsimp simp*: *not-less*)
assume *0* ≤ *x*
have (∑ *n*≤*k*. (*l* ∗ *x*) ^ *n* ∗ *exp* (− (*l* ∗ *x*)) / *fact n*) =
exp (− (*l* ∗ *x*)) ∗ (∑ *n*≤*k*. (*l* ∗ *x*) ^ *n* / *fact n*)
unfolding *sum-distrib-left* **by** (*intro sum.cong*) (*auto simp*: *field-simps*)
also have ... = (∑ *n*≤*k*. (*l* ∗ *x*) ^ *n* / *fact n*) / *exp* (*l* ∗ *x*)
by (*simp add*: *exp-minus field-simps*)
also have ... ≤ *1*
proof (*subst divide-le-eq-1-pos*)
show (∑ *n*≤*k*. (*l* ∗ *x*) ^ *n* / *fact n*) ≤ *exp* (*l* ∗ *x*)
using ‹*0* < *l*› ‹*0* ≤ *x*› *summable-exp-generic*[*of l* ∗ *x*]

by (*auto simp*: *exp-def divide-inverse ac-simps intro*!: *sum-le-suminf*)
qed *simp*
finally show $(\sum n{\le}k.\ (l * x)\ \hat{}\ n * exp\ (-\ (l * x))\ /\ fact\ n) \le 1$ **.**
qed

lemma *nn-integral-erlang-density*:
assumes [*arith*]: $0 < l$
shows $(\int^+ x.\ ennreal\ (erlang\text{-}density\ k\ l\ x) * indicator\ \{..\ a\}\ x\ \partial lborel) = erlang\text{-}CDF\ k\ l\ a$
proof (*cases* $0 \le a$)
case [*arith*]: *True*
have *eq*: $\bigwedge x.\ indicator\ \{0..a\}\ (x\ /\ l) = indicator\ \{0..a{*}l\}\ x$
by (*simp add*: *field-simps split*: *split-indicator*)
have $(\int^+ x.\ ennreal\ (erlang\text{-}density\ k\ l\ x) * indicator\ \{..\ a\}\ x\ \partial lborel) =$
$(\int^+ x.\ (l/fact\ k) * (ennreal\ ((l{*}x)\hat{}k * exp\ (-\ (l{*}x))) * indicator\ \{0\ ..\ a\}\ x)\ \partial lborel)$
by (*intro nn-integral-cong*)
(*auto simp*: *erlang-density-def power-mult-distrib ennreal-mult*[*symmetric*] *split*: *split-indicator*)
also have $\ldots = (l/fact\ k) * (\int^+ x.\ ennreal\ ((l{*}x)\hat{}k * exp\ (-\ (l{*}x))) * indicator\ \{0\ ..\ a\}\ x\ \partial lborel)$
by (*intro nn-integral-cmult*) *auto*
also have $\ldots = ennreal\ (l/fact\ k) * ((1/l) * (\int^+ x.\ ennreal\ (x\hat{}k * exp\ (-\ x)) * indicator\ \{0\ ..\ l * a\}\ x\ \partial lborel))$
by (*subst nn-integral-real-affine*[**where** *c=1 / l* **and** *t=0*]) (*auto simp*: *field-simps eq*)
also have $\ldots = (1\ -\ (\sum n{\le}k.\ ((l * a)\hat{}n * exp\ (-(l * a)))\ /\ fact\ n))$
by (*subst nn-intergal-power-times-exp-Icc*) (*auto simp*: *ennreal-mult'*[*symmetric*])
also have $\ldots = erlang\text{-}CDF\ k\ l\ a$
by (*auto simp*: *erlang-CDF-def*)
finally show *?thesis* **.**
next
case *False*
then have $(\int^+ x.\ ennreal\ (erlang\text{-}density\ k\ l\ x) * indicator\ \{..\ a\}\ x\ \partial lborel) = (\int^+ x.\ 0\ \partial(lborel::real\ measure))$
by (*intro nn-integral-cong*) (*auto simp*: *erlang-density-def*)
with *False* **show** *?thesis*
by (*simp add*: *erlang-CDF-def*)
qed

lemma *emeasure-erlang-density*:
$0 < l \Longrightarrow emeasure\ (density\ lborel\ (erlang\text{-}density\ k\ l))\ \{..\ a\} = erlang\text{-}CDF\ k\ l\ a$
by (*simp add*: *emeasure-density nn-integral-erlang-density*)

lemma *nn-integral-erlang-ith-moment*:
fixes *k i* :: *nat* **and** *l* :: *real*
assumes [*arith*]: $0 < l$
shows $(\int^+ x.\ ennreal\ (erlang\text{-}density\ k\ l\ x * x\ \hat{}\ i)\ \partial lborel) = fact\ (k + i)\ /$

(*fact k * l ^ i*)
proof −
have *eq*: ⋀*x*. *indicator* {*0*..} (*x* / *l*) = *indicator* {*0*..} *x*
by (*simp add*: *field-simps split*: *split-indicator*)
have (∫⁺ *x*. *ennreal* (*erlang-density k l x * x^i*) *∂lborel*) =
(∫⁺*x*. (*l*/(*fact k * l^i*)) * (*ennreal* ((*l**x*)^(*k*+*i*) * *exp* (− (*l**x*))) * *indicator* {*0*
..} *x*) *∂lborel*)
by (*intro nn-integral-cong*)
(*auto simp*: *erlang-density-def power-mult-distrib power-add ennreal-mult'*[*symmetric*]
split: *split-indicator*)
also have ... = (*l*/(*fact k * l^i*)) * (∫⁺*x*. *ennreal* ((*l**x*)^(*k*+*i*) * *exp* (− (*l**x*)))
* *indicator* {*0* ..} *x ∂lborel*)
by (*intro nn-integral-cmult*) *auto*
also have ... = *ennreal* (*l*/(*fact k * l^i*)) * ((*1*/*l*) * (∫⁺*x*. *ennreal* (*x^(k+i)* *
exp (− *x*)) * *indicator* {*0* ..} *x ∂lborel*))
by (*subst nn-integral-real-affine*[**where** *c=1* / *l* **and** *t=0*]) (*auto simp*: *field-simps*
eq)
also have ... = *fact* (*k* + *i*) / (*fact k * l ^ i*)
by (*subst nn-intergal-power-times-exp-Ici*) (*auto simp*: *ennreal-mult'*[*symmetric*])
finally show *?thesis* **.**
qed

lemma *prob-space-erlang-density*:
assumes *l*[*arith*]: *0* < *l*
shows *prob-space* (*density lborel* (*erlang-density k l*)) (**is** *prob-space ?D*)
proof
show *emeasure ?D* (*space ?D*) = *1*
using *nn-integral-erlang-ith-moment*[*OF l*, **where** *k=k* **and** *i=0*] **by** (*simp*
add: *emeasure-density*)
qed

lemma (**in** *prob-space*) *erlang-distributed-le*:
assumes *D*: *distributed M lborel X* (*erlang-density k l*)
assumes [*simp*, *arith*]: *0* < *l 0* ≤ *a*
shows *P*(*x in M*. *X x* ≤ *a*) = *erlang-CDF k l a*
proof −
have *emeasure M* {*x* ∈ *space M*. *X x* ≤ *a* } = *emeasure* (*distr M lborel X*) {..
a}
using *distributed-measurable*[*OF D*]
by (*subst emeasure-distr*) (*auto intro*!: *arg-cong2*[**where** *f=emeasure*])
also have ... = *emeasure* (*density lborel* (*erlang-density k l*)) {.. *a*}
unfolding *distributed-distr-eq-density*[*OF D*] **..**
also have ... = *erlang-CDF k l a*
by (*auto intro*!: *emeasure-erlang-density*)
finally show *?thesis*
by (*auto simp*: *emeasure-eq-measure measure-nonneg*)
qed

lemma (**in** *prob-space*) *erlang-distributed-gt*:

assumes *D*[*simp*]: *distributed M lborel X* (*erlang-density k l*)
assumes [*arith*]: *0 < l 0 ≤ a*
shows $\mathcal{P}(x \text{ in } M.\ a < X\ x\ ) = 1 - (\text{erlang-CDF } k\ l\ a)$
proof −
have *1 − (erlang-CDF k l a) = 1 − P*(*x in M. X x ≤ a*) **by** (*subst erlang-distributed-le*) *auto*
also have ... = *prob* (*space M − {x ∈ space M. X x ≤ a }*)
using *distributed-measurable*[*OF D*] **by** (*auto simp*: *prob-compl*)
also have ... = $\mathcal{P}(x \text{ in } M.\ a < X\ x\ )$ **by** (*auto intro*!: *arg-cong*[**where** *f*=*prob*] *simp*: *not-le*)
finally show *?thesis* **by** *simp*
qed

lemma *erlang-CDF-at0*: *erlang-CDF k l 0 = 0*
by (*induction k*) (*auto simp*: *erlang-CDF-def*)

lemma *erlang-distributedI*:
assumes *X*[*measurable*]: *X ∈ borel-measurable M* **and** [*arith*]: *0 < l*
and *X-distr*: ⋀*a. 0 ≤ a ⟹ emeasure M* {*x*∈*space M. X x ≤ a*} = *erlang-CDF k l a*
shows *distributed M lborel X* (*erlang-density k l*)
proof (*rule distributedI-borel-atMost*)
fix *a* :: *real*
{ assume *a ≤ 0*
with *X* **have** *emeasure M* {*x*∈*space M. X x ≤ a*} ≤ *emeasure M* {*x*∈*space M. X x ≤ 0*}
by (*intro emeasure-mono*) *auto*
also have ... = *0* **by** (*auto intro*!: *erlang-CDF-at0 simp*: *X-distr*[*of 0*])
finally have *emeasure M* {*x*∈*space M. X x ≤ a*} ≤ *0* **by** *simp*
then have *emeasure M* {*x*∈*space M. X x ≤ a*} = *0* **by** *simp*
}
note *eq-0* = *this*

show ($\int^+ x.$ *erlang-density k l x ∗ indicator* {..*a*} *x* ∂*lborel*) = *ennreal* (*erlang-CDF k l a*)
using *nn-integral-erlang-density*[*of l k a*]
by (*simp add*: *ennreal-indicator ennreal-mult*)

show *emeasure M* {*x*∈*space M. X x ≤ a*} = *ennreal* (*erlang-CDF k l a*)
using *X-distr*[*of a*] *eq-0* **by** (*auto simp*: *one-ennreal-def erlang-CDF-def*)
qed *simp-all*

lemma (**in** *prob-space*) *erlang-distributed-iff*:
assumes [*arith*]: *0<l*
shows *distributed M lborel X* (*erlang-density k l*) ⟷
(*X ∈ borel-measurable M* ∧ *0 < l* ∧ (∀ *a*≥*0*. $\mathcal{P}(x \text{ in } M.\ X\ x \le a)$ = *erlang-CDF k l a*))
using
distributed-measurable[*of M lborel X erlang-density k l*]

emeasure-erlang-density[*of l*]
erlang-distributed-le[*of X k l*]
by (*auto intro*!: *erlang-distributedI simp*: *one-ennreal-def emeasure-eq-measure*)

lemma (**in** *prob-space*) *erlang-distributed-mult-const*:
assumes *erlX*: *distributed M lborel X* (*erlang-density k l*)
assumes *a-pos*[*arith*]: $0 < \alpha$ $0 < l$
shows *distributed M lborel* ($\lambda x.\ \alpha * X\ x$) (*erlang-density k* ($l$ / α))
proof (*subst erlang-distributed-iff*, *safe*)
have [*measurable*]: *random-variable borel X* **and** [*arith*]: $0 < l$
and [*simp*]: $\bigwedge a.\ 0 \le a \Longrightarrow$ *prob* $\{x \in$ *space M*. *X x* $\le a\}$ = *erlang-CDF k l a*
by(*insert erlX*, *auto simp*: *erlang-distributed-iff*)

show *random-variable borel* ($\lambda x.\ \alpha * X\ x$) $0 < l$ / α $0 < l$ / α
by (*auto simp*:*field-simps*)

fix *a*:: *real* **assume** [*arith*]: $0 \le a$
obtain *b*:: *real* **where** [*simp*, *arith*]: $b = a/\ \alpha$ **by** *blast*

have [*arith*]: $0 \le b$ **by** (*auto simp*: *divide-nonneg-pos*)

have *prob* $\{x \in$ *space M*. $\alpha * X\ x \le a\}$ = *prob* $\{x \in$ *space M*. *X x* $\le b\}$
by (*rule arg-cong*[**where** *f*= *prob*]) (*auto simp*:*field-simps*)

moreover have *prob* $\{x \in$ *space M*. *X x* $\le b\}$ = *erlang-CDF k l b* **by** *auto*
moreover have *erlang-CDF k* (l / α) *a* = *erlang-CDF k l b* **unfolding** *erlang-CDF-def* **by** *auto*
ultimately show *prob* $\{x \in$ *space M*. $\alpha * X\ x \le a\}$ = *erlang-CDF k* (l / α) *a*
by *fastforce*
qed

lemma (**in** *prob-space*) *has-bochner-integral-erlang-ith-moment*:
fixes *k i* :: *nat* **and** *l* :: *real*
assumes [*arith*]: $0 < l$ **and** *D*: *distributed M lborel X* (*erlang-density k l*)
shows *has-bochner-integral M* ($\lambda x.\ X\ x\ \hat{}\ i$) (*fact* ($k + i$) / (*fact k* $*$ $l\ \hat{}\ i$))
proof (*rule has-bochner-integral-nn-integral*)
show *AE x in M*. $0 \le X\ x\ \hat{}\ i$
by (*subst distributed-AE2*[*OF D*]) (*auto simp*: *erlang-density-def*)
show ($\int^+ x.$ *ennreal* (*X x* $\hat{}$ *i*) ∂M) = *ennreal* (*fact* ($k + i$) / (*fact k* $*$ $l\ \hat{}\ i$))
using *nn-integral-erlang-ith-moment*[*of l k i*]
by (*subst distributed-nn-integral*[*symmetric*, *OF D*]) (*auto simp*: *ennreal-mult'*)
qed (*insert distributed-measurable*[*OF D*], *auto*)

lemma (**in** *prob-space*) *erlang-ith-moment-integrable*:
$0 < l \Longrightarrow$ *distributed M lborel X* (*erlang-density k l*) $\Longrightarrow$ *integrable M* ($\lambda x.\ X\ x\ \hat{}\ i$)
by *rule* (*rule has-bochner-integral-erlang-ith-moment*)

lemma (**in** *prob-space*) *erlang-ith-moment*:

0 < l ⟹ distributed M lborel X (erlang-density k l) ⟹
*expectation (λx. X x ^ i) = fact (k + i) / (fact k * l ^ i)*
by (*rule has-bochner-integral-integral-eq*) (*rule has-bochner-integral-erlang-ith-moment*)

lemma (**in** *prob-space*) *erlang-distributed-variance*:
assumes [*arith*]: *0 < l* **and** *distributed M lborel X (erlang-density k l)*
shows *variance X = (k + 1) / l²*
proof (*subst variance-eq*)
show *integrable M X integrable M (λx. (X x)²)*
using *erlang-ith-moment-integrable*[*OF assms*, *of 1*] *erlang-ith-moment-integrable*[*OF assms*, *of 2*]
by *auto*

show *expectation (λx. (X x)²) − (expectation X)² = real (k + 1) / l²*
using *erlang-ith-moment*[*OF assms*, *of 1*] *erlang-ith-moment*[*OF assms*, *of 2*]
by *simp* (*auto simp*: *power2-eq-square field-simps of-nat-Suc*)
qed

13.2 Exponential distribution

abbreviation *exponential-density* :: *real ⇒ real ⇒ real* **where**
exponential-density ≡ erlang-density 0

lemma *exponential-density-def*:
*exponential-density l x = (if x < 0 then 0 else l * exp (− x * l))*
by (*simp add*: *fun-eq-iff erlang-density-def*)

lemma *erlang-CDF-0*: *erlang-CDF 0 l a = (if 0 ≤ a then 1 − exp (− l * a) else 0)*
by (*simp add*: *erlang-CDF-def*)

lemma *prob-space-exponential-density*: *0 < l ⟹ prob-space* (*density lborel* (*exponential-density l*))
by (*rule prob-space-erlang-density*)

lemma (**in** *prob-space*) *exponential-distributedD-le*:
assumes *D*: *distributed M lborel X* (*exponential-density l*) **and** *a*: *0 ≤ a* **and** *l*: *0 < l*
shows *𝒫*(*x in M. X x ≤ a*) = *1 − exp (− a * l)*
using *erlang-distributed-le*[*OF D l a*] *a* **by** (*simp add*: *erlang-CDF-def*)

lemma (**in** *prob-space*) *exponential-distributedD-gt*:
assumes *D*: *distributed M lborel X* (*exponential-density l*) **and** *a*: *0 ≤ a* **and** *l*: *0 < l*
shows *𝒫*(*x in M. a < X x*) = *exp (− a * l)*
using *erlang-distributed-gt*[*OF D l a*] *a* **by** (*simp add*: *erlang-CDF-def*)

lemma (**in** *prob-space*) *exponential-distributed-memoryless*:
assumes *D*: *distributed M lborel X* (*exponential-density l*) **and** *a*: *0 ≤ a* **and** *l*:

0 < l **and** *t*: *0 ≤ t*
shows $\mathcal{P}(x \text{ in } M.\ a + t < X\ x \mid a < X\ x) = \mathcal{P}(x \text{ in } M.\ t < X\ x)$
proof −
have $\mathcal{P}(x \text{ in } M.\ a + t < X\ x \mid a < X\ x) = \mathcal{P}(x \text{ in } M.\ a + t < X\ x) \ / \ \mathcal{P}(x \text{ in } M.\ a < X\ x)$
using ‹*0 ≤ t*› **by** (*auto simp*: *cond-prob-def intro*!: *arg-cong*[**where** *f*=*prob*] *arg-cong2*[**where** *f*=(/)])
also have ... = *exp* (− (*a* + *t*) ∗ *l*) / *exp* (− *a* ∗ *l*)
using *a t* **by** (*simp add*: *exponential-distributedD-gt*[*OF D* - *l*])
also have ... = *exp* (− *t* ∗ *l*)
using *l* **by** (*auto simp*: *field-simps exp-add*[*symmetric*])
finally show *?thesis*
using *t* **by** (*simp add*: *exponential-distributedD-gt*[*OF D* - *l*])
qed

lemma *exponential-distributedI*:
assumes *X*[*measurable*]: *X* ∈ *borel-measurable M* **and** [*arith*]: *0 < l*
and *X-distr*: ⋀*a*. *0 ≤ a* ⟹ *emeasure M* {*x*∈*space M*. *X x* ≤ *a*} = *1* − *exp* (− *a* ∗ *l*)
shows *distributed M lborel X* (*exponential-density l*)
proof (*rule erlang-distributedI*)
fix *a* :: *real* **assume** *0 ≤ a* **then show** *emeasure M* {*x* ∈ *space M*. *X x* ≤ *a*} = *ennreal* (*erlang-CDF 0 l a*)
using *X-distr*[*of a*] **by** (*simp add*: *erlang-CDF-def ennreal-minus ennreal-1*[*symmetric*] *del*: *ennreal-1*)
qed *fact*+

lemma (**in** *prob-space*) *exponential-distributed-iff*:
assumes *0 < l*
shows *distributed M lborel X* (*exponential-density l*) ⟷
(*X* ∈ *borel-measurable M* ∧ (∀ *a*≥*0*. $\mathcal{P}(x \text{ in } M.\ X\ x \le a) = 1 - exp\ (-\ a * l)$))
using *assms erlang-distributed-iff*[*of l X 0*] **by** (*auto simp*: *erlang-CDF-0*)

lemma (**in** *prob-space*) *exponential-distributed-expectation*:
0 < l ⟹ *distributed M lborel X* (*exponential-density l*) ⟹ *expectation X = 1 / l*
using *erlang-ith-moment*[*of l X 0 1*] **by** *simp*

lemma *exponential-density-nonneg*: *0 < l* ⟹ *0 ≤ exponential-density l x*
by (*auto simp*: *exponential-density-def*)

lemma (**in** *prob-space*) *exponential-distributed-min*:
assumes *0 < l 0 < u*
assumes *expX*: *distributed M lborel X* (*exponential-density l*)
assumes *expY*: *distributed M lborel Y* (*exponential-density u*)
assumes *ind*: *indep-var borel X borel Y*
shows *distributed M lborel* (λ*x*. *min* (*X x*) (*Y x*)) (*exponential-density* (*l* + *u*))
proof (*subst exponential-distributed-iff*, *safe*)

have *randX*: *random-variable borel X*
using *expX* ‹*0 < l*› **by** (*simp add*: *exponential-distributed-iff*)
moreover have *randY*: *random-variable borel Y*
using *expY* ‹*0 < u*› **by** (*simp add*: *exponential-distributed-iff*)
ultimately show *random-variable borel* (λ*x*. *min* (*X x*) (*Y x*)) **by** *auto*

show *0 < l + u*
using ‹*0 < l*› ‹*0 < u*› **by** *auto*

fix *a*::*real* **assume** *a*[*arith*]: *0 ≤ a*
have *gt1*[*simp*]: $\mathcal{P}$(*x in M*. *a < X x*) = *exp* (− *a* ∗ *l*)
by (*rule exponential-distributedD-gt*[*OF expX a*]) *fact*
have *gt2*[*simp*]: $\mathcal{P}$(*x in M*. *a < Y x*) = *exp* (− *a* ∗ *u*)
by (*rule exponential-distributedD-gt*[*OF expY a*]) *fact*

have $\mathcal{P}$(*x in M*. *a* < (*min* (*X x*) (*Y x*))) = $\mathcal{P}$(*x in M*. *a* < (*X x*) ∧ *a* < (*Y x*))
by (*auto intro*!:*arg-cong*[**where** *f*=*prob*])

also have ... = $\mathcal{P}$(*x in M*. *a* < (*X x*)) ∗ $\mathcal{P}$(*x in M*. *a*< (*Y x*))
using *prob-indep-random-variable*[*OF ind*, *of* {*a* <..} {*a* <..}] **by** *simp*
also have ... = *exp* (− *a* ∗ (*l* + *u*)) **by** (*auto simp*:*field-simps mult-exp-exp*)
finally have *indep-prob*: $\mathcal{P}$(*x in M*. *a* < (*min* (*X x*) (*Y x*))) = *exp* (− *a* ∗ (*l* + *u*)) **.**

have {*x* ∈ *space M*. (*min* (*X x*) (*Y x*)) ≤*a* } = (*space M* − {*x* ∈ *space M*. *a*<(*min* (*X x*) (*Y x*)) })
by *auto*
then have *1* − *prob* {*x* ∈ *space M*. *a* < (*min* (*X x*) (*Y x*))} = *prob* {*x* ∈ *space M*. (*min* (*X x*) (*Y x*)) ≤ *a*}
using *randX randY* **by** (*auto simp*: *prob-compl*)
then show *prob* {*x* ∈ *space M*. (*min* (*X x*) (*Y x*)) ≤ *a*} = *1* − *exp* (− *a* ∗ (*l* + *u*))
using *indep-prob* **by** *auto*
qed

lemma (**in** *prob-space*) *exponential-distributed-Min*:
assumes *finI*: *finite I*
assumes *A*: *I* ≠ {}
assumes *l*: ⋀*i*. *i* ∈ *I* ⟹ *0 < l i*
assumes *expX*: ⋀*i*. *i* ∈ *I* ⟹ *distributed M lborel* (*X i*) (*exponential-density* (*l i*))
assumes *ind*: *indep-vars* (λ*i*. *borel*) *X I*
shows *distributed M lborel* (λ*x*. *Min* ((λ*i*. *X i x*) ‘*I*)) (*exponential-density* (∑ *i*∈*I*. *l i*))
using *assms*
proof (*induct rule*: *finite-ne-induct*)
case (*singleton i*) **then show** *?case* **by** *simp*
next
case (*insert i I*)

then have *distributed M lborel* ($\lambda x.$ *min* (*X i x*) (*Min* ((λ*i*. *X i x*) ‘*I*))) (*exponential-density* (*l i* + (∑ *i*∈*I*. *l i*)))
by (*intro exponential-distributed-min indep-vars-Min insert*)
(*auto intro*: *indep-vars-subset sum-pos*)
then show *?case*
using *insert* **by** *simp*
qed

lemma (**in** *prob-space*) *exponential-distributed-variance*:
$0 < l \Longrightarrow$ *distributed M lborel X* (*exponential-density l*) $\Longrightarrow$ *variance X* = *1* / l^2
using *erlang-distributed-variance*[*of l X 0*] **by** *simp*

lemma *nn-integral-zero'*: *AE x in M*. *f x* = *0* $\Longrightarrow$ (∫$^+$*x*. *f x* ∂*M*) = *0*
by (*simp cong*: *nn-integral-cong-AE*)

lemma *convolution-erlang-density*:
fixes k_1 k_2 :: *nat*
assumes [*simp*, *arith*]: *0* < *l*
shows (λ*x*. ∫$^+$*y*. *ennreal* (*erlang-density* k_1 *l* (*x* − *y*)) ∗ *ennreal* (*erlang-density* k_2 *l y*) ∂*lborel*) =
(*erlang-density* (*Suc* k_1 + *Suc* k_2 − *1*) *l*)
(**is** *?LHS* = *?RHS*)
proof
fix *x* :: *real*
have *x* ≤ *0* ∨ *0* < *x*
by *arith*
then show *?LHS x* = *?RHS x*
proof
assume *x* ≤ *0* **then show** *?thesis*
apply (*subst nn-integral-zero'*)
apply (*rule AE-I*[**where** *N*={*0*}])
apply (*auto simp add*: *erlang-density-def not-less*)
done
next
note *zero-le-mult-iff*[*simp*] *zero-le-divide-iff*[*simp*]

have *I-eq1*: *integral*^{N} *lborel* (*erlang-density* (*Suc* k_1 + *Suc* k_2 − *1*) *l*) = *1*
using *nn-integral-erlang-ith-moment*[*of l Suc* k_1 + *Suc* k_2 − *1 0*] **by** (*simp del*: *fact-Suc*)

have *1*: (∫$^+$ *x*. *ennreal* (*erlang-density* (*Suc* k_1 + *Suc* k_2 − *1*) *l x* ∗ *indicator* {*0*<..} *x*) ∂*lborel*) = *1*
apply (*subst I-eq1*[*symmetric*])
unfolding *erlang-density-def*
by (*auto intro*!: *nn-integral-cong split*:*split-indicator*)

have *prob-space* (*density lborel ?LHS*)
by (*intro prob-space-convolution-density*)
(*auto intro*!: *prob-space-erlang-density erlang-density-nonneg*)

```
  then have 2: integral^N lborel ?LHS = 1
    by (auto dest!: prob-space.emeasure-space-1 simp: emeasure-density)

  let ?I = (integral^N lborel (λy. ennreal ((1 − y)^ k1 ∗ y^k2 ∗ indicator {0..1} y)))
  let ?C = (fact (Suc (k1 + k2))) / ((fact k1) ∗ (fact k2))
  let ?s = Suc k1 + Suc k2 − 1
  let ?L = (λx. ∫+y. ennreal (erlang-density k1 l (x− y) ∗ erlang-density k2 l y
∗ indicator {0..x} y) ∂lborel)

  { fix x :: real assume [arith]: 0 < x
    have ∗: ⋀x y n. (x − y ∗ x::real)^n = x^n ∗ (1 − y)^n
      unfolding power-mult-distrib[symmetric] by (simp add: field-simps)

    have ?LHS x = ?L x
      unfolding erlang-density-def
      by (auto intro!: nn-integral-cong simp: ennreal-mult split:split-indicator)
    also have ... = (λx. ennreal ?C ∗ ?I ∗ erlang-density ?s l x) x
      apply (subst nn-integral-real-affine[where c=x and t = 0])
     apply (simp-all add: nn-integral-cmult[symmetric] nn-integral-multc[symmetric]
del: fact-Suc)
      apply (intro nn-integral-cong)
     apply (auto simp add: erlang-density-def mult-less-0-iff exp-minus field-simps
exp-diff power-add ∗
                           ennreal-mult[symmetric]
                 simp del: fact-Suc split: split-indicator)
      done
     finally have (∫+y. ennreal (erlang-density k1 l (x − y) ∗ erlang-density k2
l y) ∂lborel) =
      (λx. ennreal ?C ∗ ?I ∗ erlang-density ?s l x) x
      by (simp add: ennreal-mult) }
  note ∗ = this

  assume [arith]: 0 < x
  have 3: 1 = integral^N lborel (λxa. ?LHS xa ∗ indicator {0<..} xa)
    by (subst 2[symmetric])
       (auto intro!: nn-integral-cong-AE AE-I[where N={0}]
             simp: erlang-density-def  nn-integral-multc[symmetric] indicator-def
split: if-split-asm)
  also have ... = integral^N lborel (λx. (ennreal (?C) ∗ ?I) ∗ ((erlang-density ?s
l x) ∗ indicator {0<..} x))
    by (auto intro!: nn-integral-cong simp: ennreal-mult[symmetric] ∗ split: split-indicator)
  also have ... = ennreal (?C) ∗ ?I
    using 1
    by (auto simp: nn-integral-cmult)
  finally have  ennreal (?C) ∗ ?I = 1 by presburger

  then show ?thesis
    using ∗ by (simp add: ennreal-mult)
```

qed
qed

lemma (**in** *prob-space*) *sum-indep-erlang*:
assumes *indep*: *indep-var borel X borel Y*
assumes [*simp*, *arith*]: *0 < l*
assumes *erlX*: *distributed M lborel X* (*erlang-density* k_1 *l*)
assumes *erlY*: *distributed M lborel Y* (*erlang-density* k_2 *l*)
shows *distributed M lborel* (λ*x*. *X x* + *Y x*) (*erlang-density* (*Suc* k_1 + *Suc* k_2 − *1*) *l*)
using *assms*
apply (*subst convolution-erlang-density*[*symmetric*, *OF* ‹*0*<*l*›])
apply (*intro distributed-convolution*)
apply *auto*
done

lemma (**in** *prob-space*) *erlang-distributed-sum*:
assumes *finI* : *finite I*
assumes *A*: *I* ≠ {}
assumes [*simp*, *arith*]: *0 < l*
assumes *expX*: ⋀*i*. *i* ∈ *I* ⟹ *distributed M lborel* (*X i*) (*erlang-density* (*k i*) *l*)
assumes *ind*: *indep-vars* (λ*i*. *borel*) *X I*
shows *distributed M lborel* (λ*x*. ∑ *i*∈*I*. *X i x*) (*erlang-density* ((∑ *i*∈*I*. *Suc* (*k i*)) − *1*) *l*)
using *assms*
proof (*induct rule*: *finite-ne-induct*)
case (*singleton i*) **then show** *?case* **by** *auto*
next
case (*insert i I*)
then have *distributed M lborel* (λ*x*. (*X i x*) + (∑ *i*∈ *I*. *X i x*)) (*erlang-density* (*Suc* (*k i*) + *Suc* ((∑ *i*∈*I*. *Suc* (*k i*)) − *1*) − *1*) *l*)
by(*intro sum-indep-erlang indep-vars-sum*) (*auto intro*!: *indep-vars-subset*)
also have (λ*x*. (*X i x*) + (∑ *i*∈ *I*. *X i x*)) = (λ*x*. ∑ *i*∈*insert i I*. *X i x*)
using *insert* **by** *auto*
also have *Suc*(*k i*) + *Suc* ((∑ *i*∈*I*. *Suc* (*k i*)) − *1*) − *1* = (∑ *i*∈*insert i I*. *Suc* (*k i*)) − *1*
using *insert* **by** (*auto intro*!: *Suc-pred simp*: *ac-simps*)
finally show *?case* **by** *fast*
qed

lemma (**in** *prob-space*) *exponential-distributed-sum*:
assumes *finI*: *finite I*
assumes *A*: *I* ≠ {}
assumes *l*: *0 < l*
assumes *expX*: ⋀*i*. *i* ∈ *I* ⟹ *distributed M lborel* (*X i*) (*exponential-density l*)
assumes *ind*: *indep-vars* (λ*i*. *borel*) *X I*
shows *distributed M lborel* (λ*x*. ∑ *i*∈*I*. *X i x*) (*erlang-density* ((*card I*) − *1*) *l*)
using *erlang-distributed-sum*[*OF assms*] **by** *simp*

lemma (**in** *information-space*) *entropy-exponential*:
assumes *l*[*simp*, *arith*]: *0 < l*
assumes *D*: *distributed M lborel X* (*exponential-density l*)
shows *entropy b lborel X = log b* (*exp 1 / l*)
proof −
have [*simp*]: *integrable lborel* (*exponential-density l*)
using *distributed-integrable*[*OF D*, *of* λ-. *1*] **by** *simp*

have [*simp*]: *integral*^{L} *lborel* (*exponential-density l*) = *1*
using *distributed-integral*[*OF D*, *of* λ-. *1*] **by** (*simp add*: *prob-space*)

have [*simp*]: *integrable lborel* (λ*x*. *exponential-density l x* ∗ *x*)
using *erlang-ith-moment-integrable*[*OF l D*, *of 1*] *distributed-integrable*[*OF D*, *of* λ*x*. *x*] **by** *simp*

have [*simp*]: *integral*^{L} *lborel* (λ*x*. *exponential-density l x* ∗ *x*) = *1 / l*
using *erlang-ith-moment*[*OF l D*, *of 1*] *distributed-integral*[*OF D*, *of* λ*x*. *x*] **by** *simp*

have *entropy b lborel X* = − (∫ *x*. *exponential-density l x* ∗ *log b* (*exponential-density l x*) *∂lborel*)
using *D* **by** (*rule entropy-distr*) *simp*
also have (∫ *x*. *exponential-density l x* ∗ *log b* (*exponential-density l x*) *∂lborel*) =
(∫ *x*. (*ln l* ∗ *exponential-density l x* − *l* ∗ (*exponential-density l x* ∗ *x*)) / *ln b* *∂lborel*)
by (*intro Bochner-Integration.integral-cong*) (*auto simp*: *log-def ln-mult exponential-density-def field-simps*)
also have ... = (*ln l* − *1*) / *ln b*
by *simp*
finally show *?thesis*
by (*simp add*: *log-def ln-div*) (*simp add*: *field-split-simps*)
qed

13.3 Uniform distribution

lemma *uniform-distrI*:
assumes *X*: *X* ∈ *measurable M M′*
and *A*: *A* ∈ *sets M′ emeasure M′ A* ≠ ∞ *emeasure M′ A* ≠ *0*
assumes *distr*: ⋀*B*. *B* ∈ *sets M′* ⟹ *emeasure M* (*X* −‘ *B* ∩ *space M*) = *emeasure M′* (*A* ∩ *B*) / *emeasure M′ A*
shows *distr M M′ X* = *uniform-measure M′ A*
unfolding *uniform-measure-def*
proof (*intro measure-eqI*)
let *?f* = λ*x*. *indicator A x* / *emeasure M′ A*
fix *B* **assume** *B*: *B* ∈ *sets* (*distr M M′ X*)
with *X* **have** *emeasure M* (*X* −‘ *B* ∩ *space M*) = *emeasure M′* (*A* ∩ *B*) / *emeasure M′ A*
by (*simp add*: *distr*[*of B*] *measurable-sets*)

also have ... = (*1 / emeasure M' A*) * *emeasure M' (A ∩ B)*
by (*simp add*: *divide-ennreal-def ac-simps*)
also have ... = (∫⁺ *x*. (*1 / emeasure M' A*) * *indicator (A ∩ B) x* ∂*M'*)
using *A B*
by (*intro nn-integral-cmult-indicator*[*symmetric*]) (*auto intro*!:)
also have ... = (∫⁺ *x*. *?f x* * *indicator B x* ∂*M'*)
by (*rule nn-integral-cong*) (*auto split*: *split-indicator*)
finally show *emeasure* (*distr M M' X*) *B* = *emeasure* (*density M' ?f*) *B*
using *A B X* **by** (*auto simp add*: *emeasure-distr emeasure-density*)
qed *simp*

lemma *uniform-distrI-borel*:
fixes *A* :: *real set*
assumes *X*[*measurable*]: *X* ∈ *borel-measurable M* **and** *A*: *emeasure lborel A* = *ennreal r 0* < *r*
and [*measurable*]: *A* ∈ *sets borel*
assumes *distr*: ⋀*a*. *emeasure M* {*x*∈*space M*. *X x* ≤ *a*} = *emeasure lborel* (*A* ∩ {.. *a*}) / *r*
shows *distributed M lborel X* (λ*x*. *indicator A x* / *measure lborel A*)
proof (*rule distributedI-borel-atMost*)
let *?f* = λ*x*. *1 / r* * *indicator A x*
fix *a*
have *emeasure lborel* (*A* ∩ {..*a*}) ≤ *emeasure lborel A*
using *A* **by** (*intro emeasure-mono*) *auto*
also have ... < ∞
using *A* **by** *simp*
finally have *fin*: *emeasure lborel* (*A* ∩ {..*a*}) ≠ *top*
by *simp*
from *emeasure-eq-ennreal-measure*[*OF this*]
have *fin-eq*: *emeasure lborel* (*A* ∩ {..*a*}) / *r* = *ennreal* (*measure lborel* (*A* ∩ {..*a*}) / *r*)
using *A* **by** (*simp add*: *divide-ennreal measure-nonneg*)
then show *emeasure M* {*x*∈*space M*. *X x* ≤ *a*} = *ennreal* (*measure lborel* (*A* ∩ {..*a*}) / *r*)
using *distr* **by** *simp*

have (∫⁺ *x*. *ennreal* (*indicator A x* / *measure lborel A* * *indicator* {..*a*} *x*) ∂*lborel*) =
(∫⁺ *x*. *ennreal* (*1 / measure lborel A*) * *indicator* (*A* ∩ {..*a*}) *x* ∂*lborel*)
by (*auto intro*!: *nn-integral-cong split*: *split-indicator*)
also have ... = *ennreal* (*1 / measure lborel A*) * *emeasure lborel* (*A* ∩ {..*a*})
using ‹*A* ∈ *sets borel*›
by (*intro nn-integral-cmult-indicator*) (*auto simp*: *measure-nonneg*)
also have ... = *ennreal* (*measure lborel* (*A* ∩ {..*a*}) / *r*)
unfolding *emeasure-eq-ennreal-measure*[*OF fin*] **using** *A*
by (*simp add*: *measure-def ennreal-mult'*[*symmetric*])
finally show (∫⁺ *x*. *ennreal* (*indicator A x* / *measure lborel A* * *indicator* {..*a*} *x*) ∂*lborel*) =
ennreal (*measure lborel* (*A* ∩ {..*a*}) / *r*) .

qed (*auto simp*: *measure-nonneg*)

lemma (**in** *prob-space*) *uniform-distrI-borel-atLeastAtMost*:
fixes *a b* :: *real*
assumes *X*: $X \in$ *borel-measurable M* **and** $a < b$
assumes *distr*: $\bigwedge t.\ a \leq t \Longrightarrow t \leq b \Longrightarrow \mathcal{P}(x\ in\ M.\ X\ x \leq t) = (t - a)\ /\ (b - a)$
shows *distributed M lborel X* ($\lambda x.$ *indicator* $\{a..b\}$ *x / measure lborel* $\{a..b\}$)
proof (*rule uniform-distrI-borel*)
fix *t*
have $t < a \vee (a \leq t \wedge t \leq b) \vee b < t$
by *auto*
then show *emeasure M* $\{x \in space\ M.\ X\ x \leq t\}$ = *emeasure lborel* $(\{a\ ..\ b\} \cap \{..t\})\ /\ (b - a)$
proof (*elim disjE conjE*)
assume $t < a$
then have *emeasure M* $\{x \in space\ M.\ X\ x \leq t\} \leq$ *emeasure M* $\{x \in space\ M.\ X\ x \leq a\}$
using *X* **by** (*auto intro*!: *emeasure-mono measurable-sets*)
also have $\ldots = 0$
using *distr*[*of a*] ‹$a < b$› **by** (*simp add*: *emeasure-eq-measure*)
finally have *emeasure M* $\{x \in space\ M.\ X\ x \leq t\} = 0$
by (*simp add*: *antisym measure-nonneg*)
with ‹$t < a$› **show** *?thesis* **by** *simp*
next
assume *bnds*: $a \leq t\ t \leq b$
have $\{a..b\} \cap \{..t\} = \{a..t\}$
using *bnds* **by** *auto*
then show *?thesis* **using** ‹$a \leq t$› ‹$a < b$›
using *distr*[*OF bnds*] **by** (*simp add*: *emeasure-eq-measure divide-ennreal*)
next
assume $b < t$
have 1 = *emeasure M* $\{x \in space\ M.\ X\ x \leq b\}$
using *distr*[*of b*] ‹$a < b$› **by** (*simp add*: *one-ennreal-def emeasure-eq-measure*)
also have $\ldots \leq$ *emeasure M* $\{x \in space\ M.\ X\ x \leq t\}$
using *X* ‹$b < t$› **by** (*auto intro*!: *emeasure-mono measurable-sets*)
finally have *emeasure M* $\{x \in space\ M.\ X\ x \leq t\} = 1$
by (*simp add*: *antisym emeasure-eq-measure*)
with ‹$b < t$› ‹$a < b$› **show** *?thesis* **by** (*simp add*: *measure-def divide-ennreal*)
qed
qed (*insert X* ‹$a < b$›, *auto*)

lemma (**in** *prob-space*) *uniform-distributed-measure*:
fixes *a b* :: *real*
assumes *D*: *distributed M lborel X* ($\lambda x.$ *indicator* $\{a\ ..\ b\}$ *x / measure lborel* $\{a\ ..\ b\}$)
assumes *t*: $a \leq t\ t \leq b$
shows $\mathcal{P}(x\ in\ M.\ X\ x \leq t) = (t - a)\ /\ (b - a)$
proof −

have *emeasure M* {$x \in$ *space M*. *X x* $\leq$ *t*} = *emeasure* (*distr M lborel X*) {.. *t*}
using *distributed-measurable*[*OF D*]
by (*subst emeasure-distr*) (*auto intro*!: *arg-cong2*[**where** *f*=*emeasure*])
also have ... = ($\int^+ x$. *ennreal* (*1* / (*b* − *a*)) ∗ *indicator* {*a* .. *t*} *x* ∂*lborel*)
using *distributed-borel-measurable*[*OF D*] ‹*a* ≤ *t*› ‹*t* ≤ *b*›
unfolding *distributed-distr-eq-density*[*OF D*]
by (*subst emeasure-density*)
(*auto intro*!: *nn-integral-cong simp*: *measure-def split*: *split-indicator*)
also have ... = *ennreal* (*1* / (*b* − *a*)) ∗ (*t* − *a*)
using ‹*a* ≤ *t*› ‹*t* ≤ *b*›
by (*subst nn-integral-cmult-indicator*) *auto*
finally show *?thesis*
using *t* **by** (*simp add*: *emeasure-eq-measure ennreal-mult''*[*symmetric*] *measure-nonneg*)
qed

lemma (**in** *prob-space*) *uniform-distributed-bounds*:
fixes *a b* :: *real*
assumes *D*: *distributed M lborel X* (λ*x*. *indicator* {*a* .. *b*} *x* / *measure lborel* {*a* .. *b*})
shows *a* < *b*
proof (*rule ccontr*)
assume ¬ *a* < *b*
then have {*a* .. *b*} = {} ∨ {*a* .. *b*} = {*a* .. *a*} **by** *simp*
with *uniform-distributed-params*[*OF D*] **show** *False*
by (*auto simp*: *measure-def*)
qed

lemma (**in** *prob-space*) *uniform-distributed-iff*:
fixes *a b* :: *real*
shows *distributed M lborel X* (λ*x*. *indicator* {*a*..*b*} *x* / *measure lborel* {*a*..*b*}) ⟷
(*X* ∈ *borel-measurable M* ∧ *a* < *b* ∧ (∀ *t*∈{*a* .. *b*}. $\mathcal{P}$(*x in M*. *X x* ≤ *t*)= (*t* − *a*) / (*b* − *a*)))
using
uniform-distributed-bounds[*of X a b*]
uniform-distributed-measure[*of X a b*]
distributed-measurable[*of M lborel X*]
by (*auto intro*!: *uniform-distrI-borel-atLeastAtMost simp del*: *content-real-if*)

lemma (**in** *prob-space*) *uniform-distributed-expectation*:
fixes *a b* :: *real*
assumes *D*: *distributed M lborel X* (λ*x*. *indicator* {*a* .. *b*} *x* / *measure lborel* {*a* .. *b*})
shows *expectation X* = (*a* + *b*) / *2*
proof (*subst distributed-integral*[*OF D*, *of* λ*x*. *x*, *symmetric*])
have *a* < *b*
using *uniform-distributed-bounds*[*OF D*] .

have ($\int$ *x*. *indicator* {*a* .. *b*} *x* / *measure lborel* {*a* .. *b*} ∗ *x* ∂*lborel*) =

```
    (∫ x. (x / measure lborel {a .. b}) * indicator {a .. b} x ∂lborel)
    by (intro Bochner-Integration.integral-cong) auto
  also have (∫ x. (x / measure lborel {a .. b}) * indicator {a .. b} x ∂lborel) =
(a + b) / 2
  proof (subst integral-FTC-Icc-real)
    fix x
    show DERIV (λx. x² / (2 * measure lborel {a..b})) x :> x / measure lborel
{a..b}
      using uniform-distributed-params[OF D]
      by (auto intro!: derivative-eq-intros simp del: content-real-if)
    show isCont (λx. x / Sigma-Algebra.measure lborel {a..b}) x
      using uniform-distributed-params[OF D]
      by (auto intro!: isCont-divide)
    have *: b² / (2 * measure lborel {a..b}) - a² / (2 * measure lborel {a..b}) =
      (b*b - a * a) / (2 * (b - a))
      using ‹a < b›
      by (auto simp: measure-def power2-eq-square diff-divide-distrib[symmetric])
    show b² / (2 * measure lborel {a..b}) - a² / (2 * measure lborel {a..b}) = (a
+ b) / 2
      using ‹a < b›
      unfolding * square-diff-square-factored by (auto simp: field-simps)
  qed (insert ‹a < b›, simp)
  finally show (∫ x. indicator {a .. b} x / measure lborel {a .. b} * x ∂lborel) =
(a + b) / 2 .
qed (auto simp: measure-nonneg)

lemma (in prob-space) uniform-distributed-variance:
  fixes a b :: real
  assumes D: distributed M lborel X (λx. indicator {a .. b} x / measure lborel {a
.. b})
  shows variance X = (b - a)² / 12
proof (subst distributed-variance)
  have [arith]: a < b using uniform-distributed-bounds[OF D] .
  let ?μ = expectation X let ?D = λx. indicator {a..b} (x + ?μ) / measure lborel
{a..b}
  have (∫ x. x² * (?D x) ∂lborel) = (∫ x. x² * (indicator {a - ?μ .. b - ?μ} x) /
measure lborel {a .. b} ∂lborel)
    by (intro Bochner-Integration.integral-cong) (auto split: split-indicator)
  also have ... = (b - a)² / 12
    by (simp add: integral-power uniform-distributed-expectation[OF D])
       (simp add: eval-nat-numeral field-simps )
  finally show (∫ x. x² * ?D x ∂lborel) = (b - a)² / 12 .
qed (auto intro: D simp del: content-real-if)
```

13.4 Normal distribution

```
definition normal-density :: real ⇒ real ⇒ real ⇒ real where
  normal-density μ σ x = 1 / sqrt (2 * pi * σ²) * exp (-(x - μ)²/ (2 * σ²))
```

abbreviation *std-normal-density* :: *real* ⇒ *real* **where**
std-normal-density ≡ *normal-density 0 1*

lemma *std-normal-density-def*: *std-normal-density x* = (*1 / sqrt* (*2* ∗ *pi*)) ∗ *exp* (− x^2 / *2*)
unfolding *normal-density-def* **by** *simp*

lemma *normal-density-nonneg*[*simp*]: *0* ≤ *normal-density* μ σ *x*
by (*auto simp*: *normal-density-def*)

lemma *normal-density-pos*: *0* < σ ⟹ *0* < *normal-density* μ σ *x*
by (*auto simp*: *normal-density-def*)

lemma *borel-measurable-normal-density*[*measurable*]: *normal-density* μ σ ∈ *borel-measurable borel*
by (*auto simp*: *normal-density-def*[*abs-def*])

lemma *gaussian-moment-0*:
has-bochner-integral lborel (λ*x*. *indicator* {*0*..} *x* ∗$_R$ *exp* (− x^2)) (*sqrt pi / 2*)
proof −
let *?pI* = λ*f*. (∫$^+$*s*. *f* (*s*::*real*) ∗ *indicator* {*0*..} *s* ∂*lborel*)
let *?gauss* = λ*x*. *exp* (− x^2)

let *?I* = *indicator* {*0*<..} :: *real* ⇒ *real*
let *?ff*= λ*x s*. *x* ∗ *exp* (− x^2 ∗ (*1* + s^2)) :: *real*

have ∗: *?pI ?gauss* = (∫$^+$*x*. *?gauss x* ∗ *?I x* ∂*lborel*)
by (*intro nn-integral-cong-AE AE-I*[**where** *N*={*0*}]) (*auto split*: *split-indicator*)

have *?pI ?gauss* ∗ *?pI ?gauss* = (∫$^+$*x*. ∫$^+$*s*. *?gauss x* ∗ *?gauss s* ∗ *?I s* ∗ *?I x* ∂*lborel* ∂*lborel*)
by (*auto simp*: *nn-integral-cmult*[*symmetric*] *nn-integral-multc*[*symmetric*] ∗ *ennreal-mult*[*symmetric*]
intro!: *nn-integral-cong split*: *split-indicator*)
also have ... = (∫$^+$*x*. ∫$^+$*s*. *?ff x s* ∗ *?I s* ∗ *?I x* ∂*lborel* ∂*lborel*)
proof (*rule nn-integral-cong*, *cases*)
fix *x* :: *real* **assume** *x* ≠ *0*
then show (∫$^+$*s*. *?gauss x* ∗ *?gauss s* ∗ *?I s* ∗ *?I x* ∂*lborel*) = (∫$^+$*s*. *?ff x s* ∗ *?I s* ∗ *?I x* ∂*lborel*)
by (*subst nn-integral-real-affine*[**where** *t*=*0* **and** *c*=*x*])
(*auto simp*: *mult-exp-exp nn-integral-cmult*[*symmetric*] *field-simps zero-less-mult-iff ennreal-mult*[*symmetric*]
intro!: *nn-integral-cong split*: *split-indicator*)
qed *simp*
also have ... = ∫$^+$*s*. ∫$^+$*x*. *?ff x s* ∗ *?I s* ∗ *?I x* ∂*lborel* ∂*lborel*
by (*rule lborel-pair.Fubini'*[*symmetric*]) *auto*
also have ... = *?pI* (λ*s*. *?pI* (λ*x*. *?ff x s*))
by (*rule nn-integral-cong-AE*)
(*auto intro*!: *nn-integral-cong-AE AE-I*[**where** *N*={*0*}] *split*: *split-indicator-asm*)

```
  also have ... = ?pI (λs. ennreal (1 / (2 * (1 + s²))))
  proof (intro nn-integral-cong ennreal-mult-right-cong)
    fix s :: real show ?pI (λx. ?ff x s) = ennreal (1 / (2 * (1 + s²)))
    proof (subst nn-integral-FTC-atLeast)
      have ((λa. - (exp (- (a² * (1 + s²))) / (2 + 2 * s²))) ⟶ (- (0 / (2 +
2 * s²)))) at-top
      apply (intro tendsto-intros filterlim-compose[OF exp-at-bot] filterlim-compose[OF
filterlim-uminus-at-bot-at-top])
        apply (subst mult.commute)
        apply (auto intro!: filterlim-tendsto-pos-mult-at-top
                            filterlim-at-top-mult-at-top[OF filterlim-ident filterlim-ident]
                    simp: add-pos-nonneg  power2-eq-square add-nonneg-eq-0-iff)
        done
      then show ((λa. - (exp (- a² - s² * a²) / (2 + 2 * s²))) ⟶ 0) at-top
        by (simp add: field-simps)
    qed (auto intro!: derivative-eq-intros simp: field-simps add-nonneg-eq-0-iff)
  qed
  also have ... = ennreal (pi / 4)
  proof (subst nn-integral-FTC-atLeast)
    show ((λa. arctan a / 2) ⟶ (pi / 2) / 2 ) at-top
      by (intro tendsto-intros) (simp-all add: tendsto-arctan-at-top)
 qed (auto intro!: derivative-eq-intros simp: add-nonneg-eq-0-iff field-simps power2-eq-square)
  finally have ?pI ?gauss^2 = pi / 4
    by (simp add: power2-eq-square)
  then have ?pI ?gauss = sqrt (pi / 4)
    using power-eq-iff-eq-base[of 2 enn2real (?pI ?gauss) sqrt (pi / 4)]
     by (cases ?pI ?gauss) (auto simp: power2-eq-square ennreal-mult[symmetric]
ennreal-top-mult)
  also have ?pI ?gauss = (∫⁺x. indicator {0..} x *ᵣ exp (- x²) ∂lborel)
    by (intro nn-integral-cong) (simp split: split-indicator)
  also have sqrt (pi / 4) = sqrt pi / 2
    by (simp add: real-sqrt-divide)
  finally show ?thesis
    by (rule has-bochner-integral-nn-integral[rotated 3])
       auto
qed

lemma gaussian-moment-1:
  has-bochner-integral lborel (λx::real. indicator {0..} x *ᵣ (exp (- x²) * x)) (1 /
2)
proof -
  have (∫⁺x. indicator {0..} x *ᵣ (exp (- x²) * x) ∂lborel) =
    (∫⁺x. ennreal (x * exp (- x²)) * indicator {0..} x ∂lborel)
    by (intro nn-integral-cong)
       (auto simp: ac-simps split: split-indicator)
  also have ... = ennreal (0 - (- exp (- 0²) / 2))
  proof (rule nn-integral-FTC-atLeast)
    have ((λx::real. - exp (- x²) / 2) ⟶ - 0 / 2) at-top
      by (intro tendsto-divide tendsto-minus filterlim-compose[OF exp-at-bot]
```

```
                filterlim-compose[OF filterlim-uminus-at-bot-at-top]
                filterlim-pow-at-top filterlim-ident)
        auto
    then show ((λa::real. − exp (− a²) / 2) ⟶ 0) at-top
      by simp
  qed (auto intro!: derivative-eq-intros)
  also have ... = ennreal (1 / 2)
    by simp
  finally show ?thesis
    by (rule has-bochner-integral-nn-integral[rotated 3])
       (auto split: split-indicator)
qed

lemma
  fixes k :: nat
  shows gaussian-moment-even-pos:
     has-bochner-integral lborel (λx::real. indicator {0..} x *R (exp (−x²)*x^(2 *
k)))
       ((sqrt pi / 2) * (fact (2 * k) / (2 ^ (2 * k) * fact k)))
       (is ?even)
    and gaussian-moment-odd-pos:
      has-bochner-integral lborel (λx::real. indicator {0..} x *R (exp (−x²)*x^(2 *
k + 1))) (fact k / 2)
      (is ?odd)
proof −
  let ?M = λk x. exp (− x²) * x^k :: real

  { fix k I assume Mk: has-bochner-integral lborel (λx. indicator {0..} x *R ?M k
x) I
    have 2 ≠ (0::real)
      by linarith
    let ?f = λb. ∫ x. indicator {0..} x *R ?M (k + 2) x * indicator {..b} x ∂lborel
    have ((λb. (k + 1) / 2 * (∫ x. indicator {..b} x *R (indicator {0..} x *R ?M
k x) ∂lborel) − ?M (k + 1) b / 2) ⟶
          (k + 1) / 2 * (∫ x. indicator {0..} x *R ?M k x ∂lborel) − 0 / 2) at-top
(is ?tendsto)
    proof (intro tendsto-intros ‹2 ≠ 0› tendsto-integral-at-top sets-lborel Mk[THEN
integrable.intros])
      show (?M (k + 1) ⟶ 0) at-top
      proof cases
        assume even k
        have ((λx. ((x²)^(k div 2 + 1) / exp (x²)) * (1 / x) :: real) ⟶ 0 * 0)
at-top
        by (intro tendsto-intros tendsto-divide-0[OF tendsto-const] filterlim-compose[OF
tendsto-power-div-exp-0]
                  filterlim-at-top-imp-at-infinity filterlim-ident filterlim-pow-at-top
filterlim-ident)
           auto
        also have (λx. ((x²)^(k div 2 + 1) / exp (x²)) * (1 / x) :: real) = ?M (k
```

+ 1)
using ‹*even k*› **by** (*auto simp*: *fun-eq-iff exp-minus field-simps power2-eq-square power-mult elim*: *evenE*)
finally show *?thesis* **by** *simp*
next
assume *odd k*
have ((λ*x*. ((*x*²)^((*k* − *1*) *div* 2 + *1*) / *exp* (*x*²)) :: *real*) ⟶ *0*) *at-top*
by (*intro filterlim-compose*[*OF tendsto-power-div-exp-0*] *filterlim-at-top-imp-at-infinity filterlim-ident filterlim-pow-at-top*)
auto
also have (λ*x*. ((*x*²)^((*k* − *1*) *div* 2 + *1*) / *exp* (*x*²)) :: *real*) = *?M* (*k* + *1*)
using ‹*odd k*› **by** (*auto simp*: *fun-eq-iff exp-minus field-simps power2-eq-square power-mult elim*: *oddE*)
finally show *?thesis* **by** *simp*
qed
qed
also have *?tendsto* ⟷ ((*?f* ⟶ (*k* + *1*) / *2* ∗ (∫ *x*. *indicator* {*0*..} *x* ∗$_R$ *?M k x* ∂*lborel*) − *0* / *2*) *at-top*)
proof (*intro filterlim-cong refl eventually-at-top-linorder*[*THEN iffD2*] *exI*[*of - 0*] *allI impI*)
fix *b* :: *real* **assume** *b*: *0* ≤ *b*
have *Suc k* ∗ (∫ *x*. *indicator* {*0*..*b*} *x* ∗$_R$ *?M k x* ∂*lborel*) = (∫ *x*. *indicator* {*0*..*b*} *x* ∗$_R$ (*exp* (− *x*²) ∗ ((*Suc k*) ∗ *x* ^ *k*)) ∂*lborel*)
unfolding *integral-mult-right-zero*[*symmetric*] **by** (*intro Bochner-Integration.integral-cong*) *auto*
also have ... = *exp* (− *b*²) ∗ *b* ^ (*Suc k*) − *exp* (− *0*²) ∗ *0* ^ (*Suc k*) − (∫ *x*. *indicator* {*0*..*b*} *x* ∗$_R$ (− *2* ∗ *x* ∗ *exp* (− *x*²) ∗ *x* ^ (*Suc k*)) ∂*lborel*)
by (*rule integral-by-parts'*)
(*auto intro*!: *derivative-eq-intros b*
simp: *diff-Suc of-nat-Suc field-simps split*: *nat.split*)
also have ... = *exp* (− *b*²) ∗ *b* ^ (*Suc k*) − (∫ *x*. *indicator* {*0*..*b*} *x* ∗$_R$ (− *2* ∗ (*exp* (− *x*²) ∗ *x* ^ (*k* + *2*))) ∂*lborel*)
by (*auto simp*: *intro*!: *Bochner-Integration.integral-cong*)
also have ... = *exp* (− *b*²) ∗ *b* ^ (*Suc k*) + *2* ∗ (∫ *x*. *indicator* {*0*..*b*} *x* ∗$_R$ *?M* (*k* + *2*) *x* ∂*lborel*)
unfolding *Bochner-Integration.integral-mult-right-zero* [*symmetric*]
by (*simp del*: *real-scaleR-def*)
finally have *Suc k* ∗ (∫ *x*. *indicator* {*0*..*b*} *x* ∗$_R$ *?M k x* ∂*lborel*) = *exp* (− *b*²) ∗ *b* ^ (*Suc k*) + *2* ∗ (∫ *x*. *indicator* {*0*..*b*} *x* ∗$_R$ *?M* (*k* + *2*) *x* ∂*lborel*) .
then show (*k* + *1*) / *2* ∗ (∫ *x*. *indicator* {..*b*} *x* ∗$_R$ (*indicator* {*0*..} *x* ∗$_R$ *?M k x*)∂*lborel*) − *exp* (− *b*²) ∗ *b* ^ (*k* + *1*) / *2* = *?f b*
by (*simp add*: *field-simps atLeastAtMost-def indicator-inter-arith*)
qed
finally have *int-M-at-top*: ((*?f* ⟶ (*k* + *1*) / *2* ∗ (∫ *x*. *indicator* {*0*..} *x* ∗$_R$ *?M k x* ∂*lborel*)) *at-top*)
by *simp*

have *has-bochner-integral lborel* (λ*x*. *indicator* {*0*..} *x* ∗$_R$ *?M* (*k* + *2*) *x*) ((*k*

*+ 1) / 2 * I)*
proof (*rule has-bochner-integral-monotone-convergence-at-top*)
fix *y* :: *real*
have *∗*: (*λx. indicator {0..} x ∗R ?M (k + 2) x * indicator {..y} x*::*real*) =
(*λx. indicator {0..y} x ∗R ?M (k + 2) x*)
by *rule* (*simp split*: *split-indicator*)
show *integrable lborel* (*λx. indicator {0..} x ∗R* (*?M (k + 2) x*) *∗ indicator {..y} x*::*real*)
unfolding *∗* **by** (*rule borel-integrable-compact*) (*auto intro*!: *continuous-intros*)
show ((*?f ⟶ (k + 1) / 2 * I*) *at-top*)
using *int-M-at-top has-bochner-integral-integral-eq*[*OF Mk*] **by** *simp*
qed (*auto split*: *split-indicator*) **}**
note *step* = *this*

show *?even*
proof (*induct k*)
case (*Suc k*)
note *step*[*OF this*]
also have (*real (2 * k + 1) / 2 * (sqrt pi / 2 * ((fact (2 * k)) / ((2*::*real)^(2*k) * fact k)))) =*
*sqrt pi / 2 * ((fact (2 * Suc k)) / ((2*::*real)^(2 * Suc k) * fact (Suc k)))*
apply (*simp add*: *field-simps del*: *fact-Suc*)
apply (*simp add*: *of-nat-mult field-simps*)
done
finally show *?case*
by *simp*
qed (*insert gaussian-moment-0*, *simp*)

show *?odd*
proof (*induct k*)
case (*Suc k*)
note *step*[*OF this*]
also have (*real (2 * k + 1 + 1) / (2*::*real) * ((fact k) / 2)) = (fact (Suc k)) / 2*
by (*simp add*: *field-simps of-nat-Suc field-split-simps del*: *fact-Suc*) (*simp add*: *field-simps*)
finally show *?case*
by *simp*
qed (*insert gaussian-moment-1*, *simp*)
qed

context
fixes *k* :: *nat* **and** *μ σ* :: *real* **assumes** [*arith*]: *0 < σ*
begin

lemma *normal-moment-even*:
has-bochner-integral lborel (*λx. normal-density μ σ x * (x − μ) ^ (2 * k)*) (*fact (2 * k) / ((2 / σ*^{2}*)^k * fact k)*)

proof −
have *eq*: $\bigwedge x$::*real*. $x^2 \hat{} k = (x \hat{} k)^2$
by (*simp add*: *power-mult*[*symmetric*] *ac-simps*)

have *has-bochner-integral lborel* (λx. *exp* $(-x^2) * x \hat{} (2 * k)$)
(*sqrt pi* $*$ (*fact* $(2 * k)$ / $(2 \hat{} (2 * k) *$ *fact k*)))
using *has-bochner-integral-even-function*[*OF gaussian-moment-even-pos*[**where** *k*=*k*]] **by** *simp*
then have *has-bochner-integral lborel* (λx. (*exp* $(-x^2) * x \hat{} (2 * k)) * ((2 * \sigma^2) \hat{} k$ / *sqrt* $(2 * pi * \sigma^2)))$
((*sqrt pi* $*$ (*fact* $(2 * k)$ / $(2 \hat{} (2 * k) *$ *fact k*))) $* ((2 * \sigma^2) \hat{} k$ / *sqrt* $(2 * pi * \sigma^2)))$
by (*rule has-bochner-integral-mult-left*)
also have (λx. (*exp* $(-x^2) * x \hat{} (2 * k)) * ((2 * \sigma^2) \hat{} k$ / *sqrt* $(2 * pi * \sigma^2))) =$
(λx. *exp* $(-$ $((sqrt\ 2 * \sigma) * x)^2$ / $(2 * \sigma^2)) * ((sqrt\ 2 * \sigma) * x) \hat{} (2 * k)$ / *sqrt* $(2 * pi * \sigma^2)$)
by (*auto simp*: *fun-eq-iff field-simps real-sqrt-power*[*symmetric*] *real-sqrt-mult*
real-sqrt-divide power-mult eq)
also have ((*sqrt pi* $*$ (*fact* $(2 * k)$ / $(2 \hat{} (2 * k) *$ *fact k*))) $* ((2 * \sigma^2) \hat{} k$ / *sqrt* $(2 * pi * \sigma^2))) =$
(*inverse* (*sqrt* $2 * \sigma$) $*$ ((*fact* $(2 * k)$)) / $((2/\sigma^2) \hat{} k *$ (*fact k*)))
by (*auto simp*: *fun-eq-iff power-mult field-simps real-sqrt-power*[*symmetric*] *real-sqrt-mult*
power2-eq-square)
finally show *?thesis*
unfolding *normal-density-def*
by (*subst lborel-has-bochner-integral-real-affine-iff*[**where** *c*=*sqrt* $2 * \sigma$ **and** *t*=μ]) *simp-all*
qed

lemma *normal-moment-abs-odd*:
has-bochner-integral lborel (λx. *normal-density* μ σ $x * |x - \mu| \hat{} (2 * k + 1)) (2 \hat{} k * \sigma \hat{} (2 * k + 1) *$ *fact k* $*$ *sqrt* $(2 / pi)$)
proof −
have *has-bochner-integral lborel* (λx::*real*. *indicator* $\{0..\}$ $x *_R$ (*exp* $(-x^2) * |x| \hat{} (2 * k + 1))) (fact\ k / 2)$
by (*rule has-bochner-integral-cong*[*THEN iffD1*, *OF - - - gaussian-moment-odd-pos*[*of k*]]) *auto*
from *has-bochner-integral-even-function*[*OF this*]
have *has-bochner-integral lborel* (λx::*real*. *exp* $(-x^2) * |x| \hat{} (2 * k + 1))$ (*fact k*)
by *simp*
then have *has-bochner-integral lborel* (λx. (*exp* $(-x^2) * |x| \hat{} (2 * k + 1)) * (2 \hat{} k * \sigma \hat{} (2 * k + 1)$ / *sqrt* $(pi * \sigma^2)))$
(*fact k* $* (2 \hat{} k * \sigma \hat{} (2 * k + 1)$ / *sqrt* $(pi * \sigma^2)))$
by (*rule has-bochner-integral-mult-left*)
also have (λx. (*exp* $(-x^2) * |x| \hat{} (2 * k + 1)) * (2 \hat{} k * \sigma \hat{} (2 * k + 1)$ / *sqrt* $(pi * \sigma^2))) =$
(λx. *exp* $(-$ $(((sqrt\ 2 * \sigma) * x)^2$ / $(2 * \sigma^2))) * |sqrt\ 2 * \sigma * x| \hat{} (2 * k + 1)$ / *sqrt* $(2 * pi * \sigma^2)$)

by (*simp add*: *field-simps abs-mult real-sqrt-power*[*symmetric*] *power-mult real-sqrt-mult*)
also have (*fact k* ∗ (2^*k* ∗ *σ*^(2 ∗ *k* + 1) / *sqrt* (*pi* ∗ *σ*²))) =
(*inverse* (*sqrt* 2) ∗ *inverse σ* ∗ (2 ^ *k* ∗ (*σ* ∗ *σ* ^ (2 ∗ *k*)) ∗ (*fact k*) ∗ *sqrt* (2 / *pi*)))
by (*auto simp*: *fun-eq-iff power-mult field-simps real-sqrt-power*[*symmetric*] *real-sqrt-divide*
real-sqrt-mult)
finally show *?thesis*
unfolding *normal-density-def*
by (*subst lborel-has-bochner-integral-real-affine-iff*[**where** *c*=*sqrt* 2 ∗ *σ* **and** *t*=*μ*])
simp-all
qed

lemma *normal-moment-odd*:
has-bochner-integral lborel (λ*x*. *normal-density μ σ x* ∗ (*x* − *μ*)^(2 ∗ *k* + 1)) 0
proof −
have *has-bochner-integral lborel* (λ*x*. *exp* (− *x*²) ∗ *x*^(2 ∗ *k* + 1)::*real*) 0
using *gaussian-moment-odd-pos* **by** (*rule has-bochner-integral-odd-function*) *simp*
then have *has-bochner-integral lborel* (λ*x*. (*exp* (−*x*²)∗*x*^(2 ∗ *k* + 1)) ∗ (2^*k*∗*σ*^(2∗*k*)/*sqrt pi*))
(0 ∗ (2^*k*∗*σ*^(2∗*k*)/*sqrt pi*))
by (*rule has-bochner-integral-mult-left*)
also have (λ*x*. (*exp* (−*x*²)∗*x*^(2 ∗ *k* + 1)) ∗ (2^*k*∗*σ*^(2∗*k*)/*sqrt pi*)) =
(λ*x*. *exp* (− ((*sqrt* 2 ∗ *σ* ∗ *x*)² / (2 ∗ *σ*²))) ∗
(*sqrt* 2 ∗ *σ* ∗ *x* ∗ (*sqrt* 2 ∗ *σ* ∗ *x*) ^ (2 ∗ *k*)) /
sqrt (2 ∗ *pi* ∗ *σ*²))
unfolding *real-sqrt-mult*
by (*simp add*: *field-simps abs-mult real-sqrt-power*[*symmetric*] *power-mult fun-eq-iff*)
finally show *?thesis*
unfolding *normal-density-def*
by (*subst lborel-has-bochner-integral-real-affine-iff*[**where** *c*=*sqrt* 2 ∗ *σ* **and** *t*=*μ*]) *simp-all*
qed

lemma *integral-normal-moment-even*:
integral^L *lborel* (λ*x*. *normal-density μ σ x* ∗ (*x* − *μ*)^(2 ∗ *k*)) = *fact* (2 ∗ *k*) / ((2 / *σ*²)^*k* ∗ *fact k*)
using *normal-moment-even* **by** (*rule has-bochner-integral-integral-eq*)

lemma *integral-normal-moment-abs-odd*:
integral^L *lborel* (λ*x*. *normal-density μ σ x* ∗ |*x* − *μ*|^(2 ∗ *k* + 1)) = 2 ^ *k* ∗ *σ* ^ (2 ∗ *k* + 1) ∗ *fact k* ∗ *sqrt* (2 / *pi*)
using *normal-moment-abs-odd* **by** (*rule has-bochner-integral-integral-eq*)

lemma *integral-normal-moment-odd*:
integral^L *lborel* (λ*x*. *normal-density μ σ x* ∗ (*x* − *μ*)^(2 ∗ *k* + 1)) = 0
using *normal-moment-odd* **by** (*rule has-bochner-integral-integral-eq*)

end

context
fixes σ :: *real*
assumes σ-*pos*[*arith*]: $0 < \sigma$
begin

lemma *normal-moment-nz-1*: *has-bochner-integral lborel* (λx. *normal-density* μ σ $x * x$) μ
proof −
note *normal-moment-even*[*OF* σ-*pos*, *of* μ *0*]
note *normal-moment-odd*[*OF* σ-*pos*, *of* μ *0*] *has-bochner-integral-mult-left*[*of* μ, *OF this*]
note *has-bochner-integral-add*[*OF this*]
then show *?thesis*
by (*simp add*: *power2-eq-square field-simps*)
qed

lemma *integral-normal-moment-nz-1*:
$integral^L$ *lborel* (λx. *normal-density* μ σ $x * x$) = μ
using *normal-moment-nz-1* **by** (*rule has-bochner-integral-integral-eq*)

lemma *integrable-normal-moment-nz-1*: *integrable lborel* (λx. *normal-density* μ σ $x * x$)
using *normal-moment-nz-1* **by** *rule*

lemma *integrable-normal-moment*: *integrable lborel* (λx. *normal-density* μ σ $x *$ $(x - \mu)\hat{}k$)
proof *cases*
assume *even k* **then show** *?thesis*
using *integrable.intros*[*OF normal-moment-even*] **by** (*auto elim*: *evenE*)
next
assume *odd k* **then show** *?thesis*
using *integrable.intros*[*OF normal-moment-odd*] **by** (*auto elim*: *oddE*)
qed

lemma *integrable-normal-moment-abs*: *integrable lborel* (λx. *normal-density* μ σ x $* |x - \mu|\hat{}k$)
proof *cases*
assume *even k* **then show** *?thesis*
using *integrable.intros*[*OF normal-moment-even*] **by** (*auto simp add*: *power-even-abs elim*: *evenE*)
next
assume *odd k* **then show** *?thesis*
using *integrable.intros*[*OF normal-moment-abs-odd*] **by** (*auto elim*: *oddE*)
qed

lemma *integrable-normal-density*[*simp*, *intro*]: *integrable lborel* (*normal-density* μ σ)
using *integrable-normal-moment*[*of* μ *0*] **by** *simp*

lemma *integral-normal-density*[*simp*]: ($\int$ *x. normal-density* μ σ *x* ∂*lborel*) = *1*
using *integral-normal-moment-even*[*of* σ μ *0*] **by** *simp*

lemma *prob-space-normal-density*:
prob-space (*density lborel* (*normal-density* μ σ))
proof qed (*simp add*: *emeasure-density nn-integral-eq-integral normal-density-nonneg*)

end

context
fixes *k* :: *nat*
begin

lemma *std-normal-moment-even*:
has-bochner-integral lborel (λ*x. std-normal-density x* * *x* ^ (*2* * *k*)) (*fact* (*2* * *k*) / (*2*^*k* * *fact k*))
using *normal-moment-even*[*of 1 0 k*] **by** *simp*

lemma *std-normal-moment-abs-odd*:
has-bochner-integral lborel (λ*x. std-normal-density x* * |*x*|^(*2* * *k* + *1*)) (*sqrt* (*2*/*pi*) * *2*^*k* * *fact k*)
using *normal-moment-abs-odd*[*of 1 0 k*] **by** (*simp add*: *ac-simps*)

lemma *std-normal-moment-odd*:
has-bochner-integral lborel (λ*x. std-normal-density x* * *x*^(*2* * *k* + *1*)) *0*
using *normal-moment-odd*[*of 1 0 k*] **by** *simp*

lemma *integral-std-normal-moment-even*:
integral^L *lborel* (λ*x. std-normal-density x* * *x*^(*2**k)) = *fact* (*2* * *k*) / (*2*^*k* * *fact k*)
using *std-normal-moment-even* **by** (*rule has-bochner-integral-integral-eq*)

lemma *integral-std-normal-moment-abs-odd*:
integral^L *lborel* (λ*x. std-normal-density x* * |*x*|^(*2* * *k* + *1*)) = *sqrt* (*2* / *pi*) * *2*^*k* * *fact k*
using *std-normal-moment-abs-odd* **by** (*rule has-bochner-integral-integral-eq*)

lemma *integral-std-normal-moment-odd*:
integral^L *lborel* (λ*x. std-normal-density x* * *x*^(*2* * *k* + *1*)) = *0*
using *std-normal-moment-odd* **by** (*rule has-bochner-integral-integral-eq*)

lemma *integrable-std-normal-moment-abs*: *integrable lborel* (λ*x. std-normal-density x* * |*x*|^*k*)

using *integrable-normal-moment-abs*[*of 1 0 k*] **by** *simp*

lemma *integrable-std-normal-moment*: *integrable lborel* (λ*x*. *std-normal-density x* ∗ *x*^*k*)
using *integrable-normal-moment*[*of 1 0 k*] **by** *simp*

end

lemma (**in** *prob-space*) *normal-density-affine*:
assumes *X*: *distributed M lborel X* (*normal-density* μ σ)
assumes [*simp*, *arith*]: *0* < σ α ≠ *0*
shows *distributed M lborel* (λ*x*. β + α ∗ *X x*) (*normal-density* (β + α ∗ μ) (|α| ∗ σ))
proof −
have *eq*: ⋀*x*. |α| ∗ *normal-density* (β + α ∗ μ) (|α| ∗ σ) (*x* ∗ α + β) = *normal-density* μ σ *x*
by (*simp add*: *normal-density-def real-sqrt-mult field-simps*)
(*simp add*: *power2-eq-square field-simps*)
show *?thesis*
by (*rule distributed-affineI*[*OF* - ‹α ≠ *0*›, **where** *t*=β])
(*simp-all add*: *eq X ennreal-mult'*[*symmetric*])
qed

lemma (**in** *prob-space*) *normal-standard-normal-convert*:
assumes *pos-var*[*simp*, *arith*]: *0* < σ
shows *distributed M lborel X* (*normal-density* μ σ) = *distributed M lborel* (λ*x*. (*X x* − μ) / σ) *std-normal-density*
proof *auto*
assume *distributed M lborel X* (λ*x*. *ennreal* (*normal-density* μ σ *x*))
then have *distributed M lborel* (λ*x*. −μ / σ + (*1*/σ) ∗ *X x*) (λ*x*. *ennreal* (*normal-density* (−μ / σ + (*1*/σ)∗ μ) (|*1*/σ| ∗ σ) *x*))
by(*rule normal-density-affine*) *auto*

then show *distributed M lborel* (λ*x*. (*X x* − μ) / σ) (λ*x*. *ennreal* (*std-normal-density x*))
by (*simp add*: *diff-divide-distrib*[*symmetric*] *field-simps*)
next
assume ∗: *distributed M lborel* (λ*x*. (*X x* − μ) / σ) (λ*x*. *ennreal* (*std-normal-density x*))
have *distributed M lborel* (λ*x*. μ + σ ∗ ((*X x* − μ) / σ)) (λ*x*. *ennreal* (*normal-density* μ |σ| *x*))
using *normal-density-affine*[*OF* ∗, *of* σ μ] **by** *simp*
then show *distributed M lborel X* (λ*x*. *ennreal* (*normal-density* μ σ *x*)) **by** *simp*
qed

lemma *conv-normal-density-zero-mean*:
assumes [*simp*, *arith*]: *0* < σ *0* < τ
shows (λ*x*. ∫⁺*y*. *ennreal* (*normal-density 0* σ (*x* − *y*) ∗ *normal-density 0* τ *y*) ∂*lborel*) =

normal-density 0 (sqrt $(\sigma^2 + \tau^2)$*)* **(is** *?LHS = ?RHS*)
proof −
define σ' τ' **where** $\sigma' = \sigma^2$ **and** $\tau' = \tau^2$
then have [*simp*, *arith*]: $0 < \sigma'$ $0 < \tau'$
by *simp-all*
let *?σ = sqrt* $((\sigma' * \tau') / (\sigma' + \tau'))$
have *sqrt*: $(sqrt\ (2 * pi * (\sigma' + \tau')) * sqrt\ (2 * pi * (\sigma' * \tau') / (\sigma' + \tau'))) =$
$(sqrt\ (2 * pi * \sigma') * sqrt\ (2 * pi * \tau'))$
by (*subst power-eq-iff-eq-base*[*symmetric*, **where** *n=2*])
(*simp-all add*: *real-sqrt-mult*[*symmetric*] *power2-eq-square*)
have *?LHS* =
$(\lambda x.\ \int^+ y.\ ennreal((normal\text{-}density\ 0\ (sqrt\ (\sigma' + \tau'))\ x) * normal\text{-}density\ (\tau' * x / (\sigma' + \tau'))\ ?\sigma\ y)\ \partial lborel)$
apply (*intro ext nn-integral-cong*)
apply (*simp add*: *normal-density-def σ'-def*[*symmetric*] *τ'-def*[*symmetric*] *sqrt mult-exp-exp*)
apply (*simp add*: *divide-simps power2-eq-square*)
apply (*simp add*: *algebra-simps*)
done

also have ... =
$(\lambda x.\ (normal\text{-}density\ 0\ (sqrt\ (\sigma^2 + \tau^2))\ x) * \int^+ y.\ ennreal(\ normal\text{-}density\ (\tau^2 * x / (\sigma^2 + \tau^2))\ ?\sigma\ y)\ \partial lborel)$
by (*subst nn-integral-cmult*[*symmetric*])
(*auto simp*: *σ'-def τ'-def normal-density-def ennreal-mult'*[*symmetric*])

also have ... = $(\lambda x.\ (normal\text{-}density\ 0\ (sqrt\ (\sigma^2 + \tau^2))\ x))$
by (*subst nn-integral-eq-integral*) (*auto simp*: *normal-density-nonneg*)

finally show *?thesis* **by** *fast*
qed

lemma *conv-std-normal-density*:
$(\lambda x.\ \int^+ y.\ ennreal\ (std\text{-}normal\text{-}density\ (x - y) * std\text{-}normal\text{-}density\ y)\ \partial lborel) =$
$(normal\text{-}density\ 0\ (sqrt\ 2))$
by (*subst conv-normal-density-zero-mean*) *simp-all*

lemma (**in** *prob-space*) *add-indep-normal*:
assumes *indep*: *indep-var borel X borel Y*
assumes *pos-var*[*arith*]: $0 < \sigma$ $0 < \tau$
assumes *normalX*[*simp*]: *distributed M lborel X* (*normal-density* μ σ)
assumes *normalY*[*simp*]: *distributed M lborel Y* (*normal-density* ν τ)
shows *distributed M lborel* $(\lambda x.\ X\ x + Y\ x)$ $(normal\text{-}density\ (\mu + \nu)\ (sqrt\ (\sigma^2 + \tau^2)))$
proof −
have *ind*[*simp*]: *indep-var borel* $(\lambda x.\ -\mu + X\ x)$ *borel* $(\lambda x.\ -\nu + Y\ x)$
proof −
have *indep-var borel* $(\ (\lambda x.\ -\mu + x)\ o\ X)$ *borel* $((\lambda x.\ -\nu + x)\ o\ Y)$

by (*auto intro*!: *indep-var-compose assms*)
then show *?thesis* **by** (*simp add*: *o-def*)
qed

have *distributed M lborel* (λ*x*. −*μ* + 1 ∗ *X x*) (*normal-density* (− *μ* + 1 ∗ *μ*) (|1| ∗ *σ*))
by(*rule normal-density-affine*[*OF normalX pos-var*(1), *of* 1 −*μ*]) *simp*
then have *1*[*simp*]: *distributed M lborel* (λ*x*. − *μ* + *X x*) (*normal-density 0 σ*) **by** *simp*

have *distributed M lborel* (λ*x*. −*ν* + 1 ∗ *Y x*) (*normal-density* (− *ν* + 1 ∗ *ν*) (|1| ∗ *τ*))
by(*rule normal-density-affine*[*OF normalY pos-var*(2), *of* 1 −*ν*]) *simp*
then have *2*[*simp*]: *distributed M lborel* (λ*x*. − *ν* + *Y x*) (*normal-density 0 τ*) **by** *simp*

have ∗: *distributed M lborel* (λ*x*. (− *μ* + *X x*) + (− *ν* + *Y x*)) (λ*x*. *ennreal* (*normal-density 0* (*sqrt* (*σ*^{2} + *τ*^{2})) *x*))
using *distributed-convolution*[*OF ind* 1 2] *conv-normal-density-zero-mean*[*OF pos-var*]
by (*simp add*: *ennreal-mult'*[*symmetric*] *normal-density-nonneg*)

have *distributed M lborel* (λ*x*. *μ* + *ν* + 1 ∗ (− *μ* + *X x* + (− *ν* + *Y x*)))
(λ*x*. *ennreal* (*normal-density* (*μ* + *ν* + 1 ∗ 0) (|1| ∗ *sqrt* (*σ*^{2} + *τ*^{2})) *x*))
by (*rule normal-density-affine*[*OF* ∗, *of* 1 *μ* + *ν*]) (*auto simp*: *add-pos-pos*)

then show *?thesis* **by** *auto*
qed

lemma (**in** *prob-space*) *diff-indep-normal*:
assumes *indep*[*simp*]: *indep-var borel X borel Y*
assumes [*simp*, *arith*]: 0 < *σ* 0 < *τ*
assumes *normalX*[*simp*]: *distributed M lborel X* (*normal-density μ σ*)
assumes *normalY*[*simp*]: *distributed M lborel Y* (*normal-density ν τ*)
shows *distributed M lborel* (λ*x*. *X x* − *Y x*) (*normal-density* (*μ* − *ν*) (*sqrt* (*σ*^{2} + *τ*^{2})))
proof −
have *distributed M lborel* (λ*x*. 0 + − 1 ∗ *Y x*) (λ*x*. *ennreal* (*normal-density* (0 + − 1 ∗ *ν*) (|− 1| ∗ *τ*) *x*))
by(*rule normal-density-affine*, *auto*)
then have [*simp*]:*distributed M lborel* (λ*x*. − *Y x*) (λ*x*. *ennreal* (*normal-density* (− *ν*) *τ x*)) **by** *simp*

have *distributed M lborel* (λ*x*. *X x* + (− *Y x*)) (*normal-density* (*μ* + − *ν*) (*sqrt* (*σ*^{2} + *τ*^{2})))
apply (*rule add-indep-normal*)
apply (*rule indep-var-compose*[*unfolded comp-def*, *of borel - borel - λx*. *x - λx*. − *x*])
apply *simp-all*

```
    done
  then show ?thesis by simp
qed

lemma (in prob-space) sum-indep-normal:
  assumes finite I I ≠ {} indep-vars (λi. borel) X I
  assumes ⋀i. i ∈ I ⟹ 0 < σ i
  assumes normal: ⋀i. i ∈ I ⟹ distributed M lborel (X i) (normal-density (μ i)
(σ i))
  shows distributed M lborel (λx. ∑ i∈I. X i x) (normal-density (∑ i∈I. μ i) (sqrt
(∑ i∈I. (σ i)²)))
  using assms
proof (induct I rule: finite-ne-induct)
  case (singleton i) then show ?case by (simp add : abs-of-pos)
next
  case (insert i I)
    then have 1: distributed M lborel (λx. (X i x) + (∑ i∈I. X i x))
              (normal-density (μ i  + sum μ I)  (sqrt ((σ i)² + (sqrt (∑ i∈I. (σ
i)²))²)))
      apply (intro add-indep-normal indep-vars-sum insert real-sqrt-gt-zero)
      apply (auto intro: indep-vars-subset intro!: sum-pos)
      apply fastforce
      done
    have 2: (λx. (X i x)+ (∑ i∈I. X i x)) = (λx. (∑ j∈insert i I. X j x))
        μ i + sum μ I = sum μ (insert i I)
      using insert by auto

     have 3: (sqrt ((σ i)² + (sqrt (∑ i∈I. (σ i)²))²)) = (sqrt (∑ i∈insert i I. (σ
i)²))
      using insert by (simp add: sum-nonneg)

    show ?case using 1 2 3 by simp
qed

lemma (in prob-space) standard-normal-distributed-expectation:
  assumes D: distributed M lborel X std-normal-density
  shows expectation X = 0
  using integral-std-normal-moment-odd[of 0]
    distributed-integral[OF D, of λx. x, symmetric]
  by (auto simp: )

lemma (in prob-space) normal-distributed-expectation:
  assumes σ[arith]: 0 < σ
  assumes D: distributed M lborel X (normal-density μ σ)
  shows expectation X = μ
   using integral-normal-moment-nz-1[OF σ, of μ] distributed-integral[OF D, of
λx. x, symmetric]
  by (auto simp: field-simps)
```

lemma (**in** *prob-space*) *normal-distributed-variance*:
fixes *a b* :: *real*
assumes [*simp*, *arith*]: $0 < \sigma$
assumes *D*: *distributed M lborel X* (*normal-density* μ σ)
shows *variance X* $= \sigma^2$
using *integral-normal-moment-even*[*of* σ μ *1*]
by (*subst distributed-integral*[*OF D*, *symmetric*])
(*simp-all add*: *eval-nat-numeral normal-distributed-expectation*[*OF assms*])

lemma (**in** *prob-space*) *standard-normal-distributed-variance*:
distributed M lborel X std-normal-density $\Longrightarrow$ *variance X* $= 1$
using *normal-distributed-variance*[*of 1 X 0*] **by** *simp*

lemma (**in** *information-space*) *entropy-normal-density*:
assumes [*arith*]: $0 < \sigma$
assumes *D*: *distributed M lborel X* (*normal-density* μ σ)
shows *entropy b lborel X* $=$ *log b* $(2 * pi * exp\ 1 * \sigma^2) / 2$
proof −
have *entropy b lborel X* $= -$ $(\int x.$ *normal-density* μ σ $x *$ *log b* (*normal-density* μ σ x) ∂*lborel*)
using *D* **by** (*rule entropy-distr*) *simp*
also have ... $= -$ $(\int x.$ *normal-density* μ σ $x * (-$ *ln* $(2 * pi * \sigma^2) -$ $(x - \mu)^2 / \sigma^2) /$ $(2 *$ *ln b*) ∂*lborel*)
by (*intro arg-cong*[**where** *f*=*uminus*] *Bochner-Integration.integral-cong*)
(*auto simp*: *normal-density-def field-simps ln-mult log-def ln-div ln-sqrt*)
also have ... $= -$ $(\int x. -$ (*normal-density* μ σ $x *$ (*ln* $(2 * pi * \sigma^2)) +$ (*normal-density* μ σ $x *$ $(x - \mu)^2) / \sigma^2) /$ $(2 *$ *ln b*) ∂*lborel*)
by (*intro arg-cong*[**where** *f*=*uminus*] *Bochner-Integration.integral-cong*) (*auto simp*: *field-split-simps field-simps*)
also have ... $=$ $(\int x.$ *normal-density* μ σ $x *$ (*ln* $(2 * pi * \sigma^2)) +$ (*normal-density* μ σ $x *$ $(x - \mu)^2) / \sigma^2$ ∂*lborel*) / $(2 *$ *ln b*)
by (*simp del*: *minus-add-distrib*)
also have ... $=$ (*ln* $(2 * pi * \sigma^2) + 1) /$ $(2 *$ *ln b*)
using *integral-normal-moment-even*[*of* σ μ *1*] **by** (*simp add*: *integrable-normal-moment fact-numeral*)
also have ... $=$ *log b* $(2 * pi * exp\ 1 * \sigma^2) / 2$
by (*simp add*: *log-def field-simps ln-mult*)
finally show *?thesis* **.**
qed

end

14 Characteristic Functions

theory *Characteristic-Functions*
imports *Weak-Convergence Independent-Family Distributions*
begin

lemma *mult-min-right*: $a \geq 0 \Longrightarrow$ $(a ::$ *real*$) *$ *min b c* $=$ *min* $(a * b)$ $(a * c)$

by (*metis min.absorb-iff2 min-def mult-left-mono*)

lemma *sequentially-even-odd*:
assumes *E*: *eventually* (λ*n*. *P* (*2* ∗ *n*)) *sequentially* **and** *O*: *eventually* (λ*n*. *P* (*2* ∗ *n* + *1*)) *sequentially*
shows *eventually P sequentially*
proof −
from *E* **obtain** *n-e* **where** [*intro*]: ⋀*n*. *n* ≥ *n-e* ⟹ *P* (*2* ∗ *n*)
by (*auto simp*: *eventually-sequentially*)
moreover
from *O* **obtain** *n-o* **where** [*intro*]: ⋀*n*. *n* ≥ *n-o* ⟹ *P* (*Suc* (*2* ∗ *n*))
by (*auto simp*: *eventually-sequentially*)
show *?thesis*
unfolding *eventually-sequentially*
proof (*intro exI allI impI*)
fix *n* **assume** *max* (*2* ∗ *n-e*) (*2* ∗ *n-o* + *1*) ≤ *n* **then show** *P n*
by (*cases even n*) (*auto elim*!: *evenE oddE*)
qed
qed

lemma *limseq-even-odd*:
assumes (λ*n*. *f* (*2* ∗ *n*)) ⟶ (*l* :: *'a* :: *topological-space*)
and (λ*n*. *f* (*2* ∗ *n* + *1*)) ⟶ *l*
shows *f* ⟶ *l*
using *assms* **by** (*auto simp*: *filterlim-iff intro*: *sequentially-even-odd*)

14.1 Application of the FTC: integrating $e^i x$

abbreviation *iexp* :: *real* ⇒ *complex* **where**
iexp ≡ (λ*x*. *exp* (i ∗ *complex-of-real x*))

lemma *isCont-iexp* [*simp*]: *isCont iexp x*
by (*intro continuous-intros*)

lemma *has-vector-derivative-iexp*[*derivative-intros*]:
(*iexp has-vector-derivative* i ∗ *iexp x*) (*at x within s*)
by (*auto intro*!: *derivative-eq-intros simp*: *Re-exp Im-exp has-vector-derivative-complex-iff*)

lemma *interval-integral-iexp*:
fixes *a b* :: *real*
shows (*CLBINT x*=*a*..*b*. *iexp x*) = i ∗ *iexp a* − i ∗ *iexp b*
by (*subst interval-integral-FTC-finite* [**where** *F* = λ*x*. −i ∗ *iexp x*])
(*auto intro*!: *derivative-eq-intros continuous-intros*)

14.2 The Characteristic Function of a Real Measure.

definition
char :: *real measure* ⇒ *real* ⇒ *complex*
where
char M t = *CLINT x*|*M*. *iexp* (*t* ∗ *x*)

lemma (**in** *real-distribution*) *char-zero*: *char M 0 = 1*
unfolding *char-def* **by** (*simp del*: *space-eq-univ add*: *prob-space*)

lemma (**in** *prob-space*) *integrable-iexp*:
assumes *f*: *f* ∈ *borel-measurable M* ⋀*x*. *Im* (*f x*) = *0*
shows *integrable M* (λ*x*. *exp* (i ∗ (*f x*)))
proof (*intro integrable-const-bound* [*of - 1*])
from *f* **have** ⋀*x*. *of-real* (*Re* (*f x*)) = *f x*
by (*simp add*: *complex-eq-iff*)
then show *AE x in M*. *cmod* (*exp* (i ∗ *f x*)) ≤ *1*
using *norm-exp-i-times*[*of Re* (*f x*) **for** *x*] **by** *simp*
qed (*insert f*, *simp*)

lemma (**in** *real-distribution*) *cmod-char-le-1*: *norm* (*char M t*) ≤ *1*
proof −
have *norm* (*char M t*) ≤ (∫ *x*. *norm* (*iexp* (*t* ∗ *x*)) ∂*M*)
unfolding *char-def* **by** (*intro integral-norm-bound*)
also have ... ≤ *1*
by (*simp del*: *of-real-mult*)
finally show *?thesis* **.**
qed

lemma (**in** *real-distribution*) *isCont-char*: *isCont* (*char M*) *t*
unfolding *continuous-at-sequentially*
proof *safe*
fix *X* **assume** *X*: *X* ⟶ *t*
show (*char M* ∘ *X*) ⟶ *char M t*
unfolding *comp-def char-def*
by (*rule integral-dominated-convergence*[**where** *w*=λ*-*. *1*]) (*auto intro*!: *tend-sto-intros X*)
qed

lemma (**in** *real-distribution*) *char-measurable* [*measurable*]: *char M* ∈ *borel-measurable borel*
by (*auto intro*!: *borel-measurable-continuous-onI continuous-at-imp-continuous-on isCont-char*)

14.3 Independence

lemma (**in** *prob-space*) *char-distr-add*:
fixes *X1 X2* :: *'a* ⇒ *real* **and** *t* :: *real*
assumes *indep-var borel X1 borel X2*
shows *char* (*distr M borel* (λω. *X1* ω + *X2* ω)) *t* =
char (*distr M borel X1*) *t* ∗ *char* (*distr M borel X2*) *t*
proof −
from *assms* **have** [*measurable*]: *random-variable borel X1* **by** (*elim indep-var-rv1*)
from *assms* **have** [*measurable*]: *random-variable borel X2* **by** (*elim indep-var-rv2*)

have *char* (*distr M borel* (λω. *X1* ω + *X2* ω)) *t* = (*CLINT x*|*M*. *iexp* (*t* ∗ (*X1* *x* + *X2* *x*)))
by (*simp add*: *char-def integral-distr*)
also have ... = (*CLINT x*|*M*. *iexp* (*t* ∗ (*X1* *x*)) ∗ *iexp* (*t* ∗ (*X2* *x*)))
by (*simp add*: *field-simps exp-add*)
also have ... = (*CLINT x*|*M*. *iexp* (*t* ∗ (*X1* *x*))) ∗ (*CLINT x*|*M*. *iexp* (*t* ∗ (*X2* *x*)))
by (*intro indep-var-lebesgue-integral indep-var-compose*[*unfolded comp-def*, *OF assms*])
(*auto intro*!: *integrable-iexp*)
also have ... = *char* (*distr M borel X1*) *t* ∗ *char* (*distr M borel X2*) *t*
by (*simp add*: *char-def integral-distr*)
finally show *?thesis* **.**
qed

lemma (**in** *prob-space*) *char-distr-sum*:
indep-vars (λ*i*. *borel*) *X A* ⟹
char (*distr M borel* (λω. ∑ *i*∈*A*. *X i* ω)) *t* = (∏ *i*∈*A*. *char* (*distr M borel* (*X i*)) *t*)
proof (*induct A rule*: *infinite-finite-induct*)
case (*insert x F*) **with** *indep-vars-subset*[*of* λ-. *borel X insert x F F*] **show** *?case*
by (*auto simp add*: *char-distr-add indep-vars-sum*)
qed (*simp-all add*: *char-def integral-distr prob-space del*: *distr-const*)

14.4 Approximations to e^{ix}

Proofs from Billingsley, page 343.

lemma *CLBINT-I0c-power-mirror-iexp*:
fixes *x* :: *real* **and** *n* :: *nat*
defines *f s m* ≡ *complex-of-real* ((*x* − *s*) ^ *m*)
shows (*CLBINT s=0..x*. *f s n* ∗ *iexp s*) =
x^*Suc n* / *Suc n* + (i / *Suc n*) ∗ (*CLBINT s=0..x*. *f s* (*Suc n*) ∗ *iexp s*)
proof −
have *1*:
((λ*s*. *complex-of-real*(−((*x* − *s*) ^ (*Suc n*) / (*Suc n*))) ∗ *iexp s*)
has-vector-derivative complex-of-real((*x* − *s*)^*n*) ∗ *iexp s* + (i ∗ *iexp s*) ∗ *complex-of-real*(−((*x* − *s*) ^ (*Suc n*) / (*Suc n*))))
(*at s within A*) **for** *s A*
by (*intro derivative-eq-intros*) *auto*

let *?F* = λ*s*. *complex-of-real*(−((*x* − *s*) ^ (*Suc n*) / (*Suc n*))) ∗ *iexp s*
have *x*^(*Suc n*) / (*Suc n*) = (*CLBINT s=0..x*. (*f s n* ∗ *iexp s* + (i ∗ *iexp s*) ∗ −(*f s* (*Suc n*) / (*Suc n*)))) (**is** *?LHS* = *?RHS*)
proof −
have *?RHS* = (*CLBINT s=0..x*. (*f s n* ∗ *iexp s* + (i ∗ *iexp s*) ∗
complex-of-real(−((*x* − *s*) ^ (*Suc n*) / (*Suc n*)))))
by (*cases* 0 ≤ *x*) (*auto intro*!: *simp*: *f-def*[*abs-def*])
also have ... = *?F x* − *?F* 0
unfolding *zero-ereal-def* **using** *1*

```
    by (intro interval-integral-FTC-finite)
      (auto simp: f-def add-nonneg-eq-0-iff complex-eq-iff
          intro!: continuous-at-imp-continuous-on continuous-intros)
   finally show ?thesis
    by auto
 qed
 show ?thesis
  unfolding ‹?LHS = ?RHS› f-def interval-lebesgue-integral-mult-right [symmetric]
   by (subst interval-lebesgue-integral-add(2) [symmetric])
     (auto intro!: interval-integrable-isCont continuous-intros simp: zero-ereal-def
complex-eq-iff)
qed

lemma iexp-eq1:
 fixes x :: real
 defines f s m ≡ complex-of-real ((x − s) ^ m)
 shows iexp x =
   (∑ k ≤ n. (i * x)^k / (fact k)) + ((i ^ (Suc n)) / (fact n)) * (CLBINT s=0..x.
(f s n) * (iexp s)) (is ?P n)
proof (induction n)
 show ?P 0
   by (auto simp add: field-simps interval-integral-iexp f-def zero-ereal-def)
next
 fix n assume ih: ?P n
 have **: ⋀a b :: real. a = −b ⟷ a + b = 0
   by linarith
 have *: of-nat n * of-nat (fact n) ≠ − (of-nat (fact n)::complex)
   unfolding of-nat-mult[symmetric]
  by (simp add: complex-eq-iff ** of-nat-add[symmetric] del: of-nat-mult of-nat-fact
of-nat-add)
 show ?P (Suc n)
   unfolding sum.atMost-Suc ih f-def CLBINT-I0c-power-mirror-iexp[of - n]
   by (simp add: divide-simps add-eq-0-iff *) (simp add: field-simps)
qed

lemma iexp-eq2:
 fixes x :: real
 defines f s m ≡ complex-of-real ((x − s) ^ m)
  shows iexp x = (∑ k≤Suc n. (i*x)^k/fact k) + i^Suc n/fact n * (CLBINT
s=0..x. f s n*(iexp s − 1))
proof −
 have isCont-f: isCont (λs. f s n) x for n x
   by (auto simp: f-def)
 let ?F = λs. complex-of-real (−((x − s) ^ (Suc n) / real (Suc n)))
 have calc1: (CLBINT s=0..x. f s n * (iexp s − 1)) =
   (CLBINT s=0..x. f s n * iexp s) − (CLBINT s=0..x. f s n)
   unfolding zero-ereal-def
   by (subst interval-lebesgue-integral-diff(2) [symmetric])
     (simp-all add: interval-integrable-isCont isCont-f field-simps)
```

```
  have calc2: (CLBINT s=0..x. f s n) = x^Suc n / Suc n
    unfolding zero-ereal-def
  proof (subst interval-integral-FTC-finite [where a = 0 and b = x and f = λs.
f s n and F = ?F])
    show (?F has-vector-derivative f y n) (at y within {min 0 x..max 0 x}) for y
      unfolding f-def
      by (intro has-vector-derivative-of-real)
       (auto intro!: derivative-eq-intros simp del: power-Suc simp add: add-nonneg-eq-0-iff)
  qed (auto intro: continuous-at-imp-continuous-on isCont-f)

  have calc3: i ^ (Suc (Suc n)) / (fact (Suc n)) = (i ^ (Suc n) / (fact n)) * (i /
(Suc n))
    by (simp add: field-simps)

  show ?thesis
    unfolding iexp-eq1 [where n = Suc n and x=x] calc1 calc2 calc3 unfolding
f-def
    by (subst CLBINT-I0c-power-mirror-iexp [where n = n]) auto
qed

lemma abs-LBINT-I0c-abs-power-diff:
  |LBINT s=0..x. |(x - s)^n|| = |x ^ (Suc n) / (Suc n)|
proof -
 have |LBINT s=0..x. |(x - s)^n|| = |LBINT s=0..x. (x - s)^n|
  proof cases
    assume 0 ≤ x ∨ even n
    then have (LBINT s=0..x. |(x - s)^n|) = LBINT s=0..x. (x - s)^n
        by (auto simp add: zero-ereal-def power-even-abs power-abs min-absorb1
max-absorb2
              intro!: interval-integral-cong)
    then show ?thesis by simp
  next
    assume ¬ (0 ≤ x ∨ even n)
    then have (LBINT s=0..x. |(x - s)^n|) = LBINT s=0..x. -((x - s)^n)
      by (auto simp add: zero-ereal-def power-abs min-absorb1 max-absorb2
                ereal-min[symmetric] ereal-max[symmetric] power-minus-odd[symmetric]
              simp del: ereal-min ereal-max intro!: interval-integral-cong)
    also have ... = - (LBINT s=0..x. (x - s)^n)
      by (subst interval-lebesgue-integral-uminus, rule refl)
    finally show ?thesis by simp
  qed
  also have LBINT s=0..x. (x - s)^n = x^Suc n / Suc n
  proof -
    let ?F = λt. - ((x - t)^(Suc n) / Suc n)
    have LBINT s=0..x. (x - s)^n = ?F x - ?F 0
      unfolding zero-ereal-def
      by (intro interval-integral-FTC-finite continuous-at-imp-continuous-on
            has-field-derivative-iff-has-vector-derivative[THEN iffD1])
```

```
    (auto simp del: power-Suc intro!: derivative-eq-intros simp add: add-nonneg-eq-0-iff)
  also have ... = x ^ (Suc n) / (Suc n) by simp
  finally show ?thesis .
 qed
 finally show ?thesis .
qed

lemma iexp-approx1: cmod (iexp x - (∑ k ≤ n. (i * x) ^ k / fact k)) ≤ |x| ^ (Suc
n) / fact (Suc n)
proof -
 have iexp x - (∑ k ≤ n. (i * x) ^ k / fact k) =
    ((i ^ (Suc n)) / (fact n)) * (CLBINT s=0..x. (x - s) ^ n * (iexp s)) (is ?t1 =
?t2)
  by (subst iexp-eq1 [of - n], simp add: field-simps)
 then have cmod (?t1) = cmod (?t2)
  by simp
 also have ... =  (1 / of-nat (fact n)) * cmod (CLBINT s=0..x. (x - s) ^ n *
(iexp s))
  by (simp add: norm-mult norm-divide norm-power)
 also have ... ≤ (1 / of-nat (fact n)) * |LBINT s=0..x. cmod ((x - s) ^ n * (iexp
s))|
  by (intro mult-left-mono interval-integral-norm2)
    (auto simp: zero-ereal-def intro: interval-integrable-isCont)
 also have ... ≤ (1 / of-nat (fact n)) * |LBINT s=0..x. |(x - s) ^ n||
  by (simp add: norm-mult del: of-real-diff of-real-power)
 also have ... ≤ (1 / of-nat (fact n)) * |x ^ (Suc n) / (Suc n)|
  by (simp add: abs-LBINT-I0c-abs-power-diff)
 also have 1 / real-of-nat (fact n::nat) * |x ^ Suc n / real (Suc n)| =
    |x| ^ Suc n / fact (Suc n)
  by (simp add: abs-mult power-abs)
 finally show ?thesis .
qed

lemma iexp-approx2: cmod (iexp x - (∑ k ≤ n. (i * x) ^ k / fact k)) ≤ 2 * |x| ^ n
/ fact n
proof (induction n) — really cases
 case (Suc n)
 have *: ⋀a b. interval-lebesgue-integrable lborel a b f ⟹ interval-lebesgue-integrable
lborel a b g ⟹
    |LBINT s=a..b. f s| ≤ |LBINT s=a..b. g s|
  if f: ⋀s. 0 ≤ f s and g: ⋀s. f s ≤ g s for f g :: - ⇒ real
  using order-trans[OF f g] f g
  unfolding interval-lebesgue-integral-def interval-lebesgue-integrable-def set-lebesgue-integral-def
set-integrable-def
  by (auto simp: integral-nonneg-AE[OF AE-I2] intro!: integral-mono mult-mono)

 have iexp x - (∑ k ≤ Suc n. (i * x) ^ k / fact k) =
    ((i ^ (Suc n)) / (fact n)) * (CLBINT s=0..x. (x - s) ^ n * (iexp s - 1)) (is
?t1 = ?t2)
```

unfolding *iexp-eq2* [*of x n*] **by** (*simp add*: *field-simps*)
then have *cmod* (*?t1*) = *cmod* (*?t2*)
by *simp*
also have ... = (*1* / (*fact n*)) * *cmod* (*CLBINT s=0..x.* (*x* − *s*)^*n* * (*iexp s* − *1*))
by (*simp add*: *norm-mult norm-divide norm-power*)
also have ... ≤ (*1* / (*fact n*)) * |*LBINT s=0..x. cmod* ((*x* − *s*)^*n* * (*iexp s* − *1*))|
by (*intro mult-left-mono interval-integral-norm2*)
(*auto intro*!: *interval-integrable-isCont simp*: *zero-ereal-def*)
also have ... = (*1* / (*fact n*)) * |*LBINT s=0..x. abs* ((*x* − *s*)^*n*) * *cmod*((*iexp s* − *1*))|
by (*simp add*: *norm-mult del*: *of-real-diff of-real-power*)
also have ... ≤ (*1* / (*fact n*)) * |*LBINT s=0..x. abs* ((*x* − *s*)^*n*) * *2*|
by (*intro mult-left-mono* * *order-trans* [*OF norm-triangle-ineq4*])
(*auto simp*: *mult-ac zero-ereal-def intro*!: *interval-integrable-isCont*)
also have ... = (*2* / (*fact n*)) * |*x* ^ (*Suc n*) / (*Suc n*)|
by (*simp add*: *abs-LBINT-I0c-abs-power-diff abs-mult*)
also have *2* / *fact n* * |*x* ^ *Suc n* / *real* (*Suc n*)| = *2* * |*x*| ^ *Suc n* / (*fact* (*Suc n*))
by (*simp add*: *abs-mult power-abs*)
finally show *?case* .
qed (*insert norm-triangle-ineq4*[*of iexp x 1*], *simp*)

lemma (**in** *real-distribution*) *char-approx1*:
assumes *integrable-moments*: ⋀*k*. *k* ≤ *n* ⟹ *integrable M* (λ*x*. *x*^*k*)
shows *cmod* (*char M t* − (∑ *k* ≤ *n*. ((i * *t*)^*k* / *fact k*) * *expectation* (λ*x*. *x*^*k*))) ≤
(*2**|*t*|^*n* / *fact n*) * *expectation* (λ*x*. |*x*|^*n*) (**is** *cmod* (*char M t* − *?t1*) ≤ -)
proof −
have *integ-iexp*: *integrable M* (λ*x*. *iexp* (*t* * *x*))
by (*intro integrable-const-bound*) *auto*

define *c* **where** [*abs-def*]: *c k x* = (i * *t*)^*k* / *fact k* * *complex-of-real* (*x*^*k*) **for** *k x*
have *integ-c*: ⋀*k*. *k* ≤ *n* ⟹ *integrable M* (λ*x*. *c k x*)
unfolding *c-def* **by** (*intro integrable-mult-right integrable-of-real integrable-moments*)

have *k* ≤ *n* ⟹ *expectation* (*c k*) = (i**t*) ^ *k* * (*expectation* (λ*x*. *x* ^ *k*)) / *fact k* **for** *k*
unfolding *c-def integral-mult-right-zero integral-complex-of-real* **by** *simp*
then have *norm* (*char M t* − *?t1*) = *norm* (*char M t* − (*CLINT x* | *M*. (∑ *k* ≤ *n*. *c k x*)))
by (*simp add*: *integ-c*)
also have ... = *norm* ((*CLINT x* | *M*. *iexp* (*t* * *x*) − (∑ *k* ≤ *n*. *c k x*)))
unfolding *char-def* **by** (*subst Bochner-Integration.integral-diff*[*OF integ-iexp*]) (*auto intro*!: *integ-c*)
also have ... ≤ *expectation* (λ*x*. *cmod* (*iexp* (*t* * *x*) − (∑ *k* ≤ *n*. *c k x*)))
by (*intro integral-norm-bound*)

also have ... ≤ *expectation* (λ*x*. 2 * |*t*| ^ *n* / *fact n* * |*x*| ^ *n*)
proof (*rule integral-mono*)
show *integrable M* (λ*x*. *cmod* (*iexp* (*t* * *x*) − (∑ *k*≤*n*. *c k x*)))
by (*intro integrable-norm Bochner-Integration.integrable-diff integ-iexp Bochner-Integration.integrable-sum integ-c*) *simp*
show *integrable M* (λ*x*. 2 * |*t*| ^ *n* / *fact n* * |*x*| ^ *n*)
unfolding *power-abs*[*symmetric*]
by (*intro integrable-mult-right integrable-abs integrable-moments*) *simp*
show *cmod* (*iexp* (*t* * *x*) − (∑ *k*≤*n*. *c k x*)) ≤ 2 * |*t*| ^ *n* / *fact n* * |*x*| ^ *n* **for** *x*
using *iexp-approx2*[*of t* * *x n*] **by** (*auto simp add*: *abs-mult field-simps c-def*)
qed
finally show *?thesis*
unfolding *integral-mult-right-zero* **.**
qed

lemma (**in** *real-distribution*) *char-approx2*:
assumes *integrable-moments*: ⋀*k*. *k* ≤ *n* ⟹ *integrable M* (λ*x*. *x* ^ *k*)
shows *cmod* (*char M t* − (∑ *k* ≤ *n*. ((i * *t*)^*k* / *fact k*) * *expectation* (λ*x*. *x*^*k*)))
≤
(|*t*|^*n* / *fact* (*Suc n*)) * *expectation* (λ*x*. *min* (2 * |*x*|^*n* * *Suc n*) (|*t*| * |*x*|^*Suc n*))
(**is** *cmod* (*char M t* − *?t1*) ≤ -)
proof −
have *integ-iexp*: *integrable M* (λ*x*. *iexp* (*t* * *x*))
by (*intro integrable-const-bound*) *auto*

define *c* **where** [*abs-def*]: *c k x* = (i * *t*)^*k* / *fact k* * *complex-of-real* (*x*^*k*) **for** *k x*
have *integ-c*: ⋀*k*. *k* ≤ *n* ⟹ *integrable M* (λ*x*. *c k x*)
unfolding *c-def* **by** (*intro integrable-mult-right integrable-of-real integrable-moments*)

have *: *min* (2 * |*t* * *x*|^*n* / *fact n*) (|*t* * *x*|^*Suc n* / *fact* (*Suc n*)) =
|*t*|^*n* / *fact* (*Suc n*) * *min* (2 * |*x*|^*n* * *real* (*Suc n*)) (|*t*| * |*x*|^(*Suc n*)) **for** *x*
apply (*subst mult-min-right*)
apply *simp*
apply (*rule arg-cong2*[**where** *f*=*min*])
apply (*simp-all add*: *field-simps abs-mult del*: *fact-Suc*)
apply (*simp-all add*: *field-simps*)
done

have *k* ≤ *n* ⟹ *expectation* (*c k*) = (i*t) ^ *k* * (*expectation* (λ*x*. *x* ^ *k*)) / *fact k* **for** *k*
unfolding *c-def integral-mult-right-zero integral-complex-of-real* **by** *simp*
then have *norm* (*char M t* − *?t1*) = *norm* (*char M t* − (*CLINT x* | *M*. (∑ *k* ≤ *n*. *c k x*)))
by (*simp add*: *integ-c*)
also have ... = *norm* ((*CLINT x* | *M*. *iexp* (*t* * *x*) − (∑ *k* ≤ *n*. *c k x*)))
unfolding *char-def* **by** (*subst Bochner-Integration.integral-diff*[*OF integ-iexp*])

(*auto intro*!: *integ-c*)
also have ... ≤ *expectation* (λ*x*. *cmod* (*iexp* (*t* ∗ *x*) − (∑ *k* ≤ *n*. *c k x*)))
by (*rule integral-norm-bound*)
also have ... ≤ *expectation* (λ*x*. *min* (2 ∗ |*t* ∗ *x*|^*n* / *fact n*) (|*t* ∗ *x*|^(*Suc n*) / *fact* (*Suc n*)))
(**is** - ≤ *expectation ?f*)
proof (*rule integral-mono*)
show *integrable M* (λ*x*. *cmod* (*iexp* (*t* ∗ *x*) − (∑ *k*≤*n*. *c k x*)))
by (*intro integrable-norm Bochner-Integration.integrable-diff integ-iexp Bochner-Integration.integrable-sum integ-c*) *simp*
show *integrable M ?f*
by (*rule Bochner-Integration.integrable-bound*[**where** *f*=λ*x*. 2 ∗ |*t* ∗ *x*| ^ *n* / *fact n*])
(*auto simp*: *integrable-moments power-abs*[*symmetric*] *power-mult-distrib*)
show *cmod* (*iexp* (*t* ∗ *x*) − (∑ *k*≤*n*. *c k x*)) ≤ *?f x* **for** *x*
using *iexp-approx1*[*of t* ∗ *x n*] *iexp-approx2*[*of t* ∗ *x n*]
by (*auto simp add*: *abs-mult field-simps c-def intro*!: *min.boundedI*)
qed
also have ... = (|*t*|^*n* / *fact* (*Suc n*)) ∗ *expectation* (λ*x*. *min* (2 ∗ |*x*|^*n* ∗ *Suc n*) (|*t*| ∗ |*x*|^*Suc n*))
unfolding ∗
proof (*rule Bochner-Integration.integral-mult-right*)
show *integrable M* (λ*x*. *min* (2 ∗ |*x*| ^ *n* ∗ *real* (*Suc n*)) (|*t*| ∗ |*x*| ^ *Suc n*))
by (*rule Bochner-Integration.integrable-bound*[**where** *f*=λ*x*. 2 ∗ |*x*| ^ *n* ∗ *real* (*Suc n*)])
(*auto simp*: *integrable-moments power-abs*[*symmetric*] *power-mult-distrib*)
qed
finally show *?thesis*
unfolding *integral-mult-right-zero* **.**
qed

lemma (**in** *real-distribution*) *char-approx3*:
fixes *t*
assumes
integrable-1: *integrable M* (λ*x*. *x*) **and**
integral-1: *expectation* (λ*x*. *x*) = *0* **and**
integrable-2: *integrable M* (λ*x*. *x*^*2*) **and**
integral-2: *variance* (λ*x*. *x*) = σ*2*
shows *cmod* (*char M t* − (*1* − *t*^*2* ∗ σ*2* / *2*)) ≤
(*t*^*2* / *6*) ∗ *expectation* (λ*x*. *min* (*6* ∗ *x*^*2*) (*abs t* ∗ (*abs x*)^*3*))
proof −
note *real-distribution.char-approx2* [*of M 2 t*, *simplified*]
have [*simp*]: *prob UNIV* = *1* **by** (*metis prob-space space-eq-univ*)
from *integral-2* **have** [*simp*]: *expectation* (λ*x*. *x* ∗ *x*) = σ*2*
by (*simp add*: *integral-1 numeral-eq-Suc*)
have *1*: *k* ≤ *2* ⟹ *integrable M* (λ*x*. *x*^*k*) **for** *k*
using *assms* **by** (*auto simp*: *eval-nat-numeral le-Suc-eq*)
note *char-approx1*
note *2* = *char-approx1* [*of 2 t*, *OF 1*, *simplified*]

have *cmod* (*char M t* − (∑ *k*≤*2*. (i ∗ *t*) ^ *k* ∗ (*expectation* (λ*x*. *x* ^ *k*)) / (*fact* *k*))) ≤
*t*² ∗ *expectation* (λ*x*. *min* (*6* ∗ *x*²) (|*t*| ∗ |*x*| ^ *3*)) / *fact* (*3*::*nat*)
using *char-approx2* [*of 2 t*, *OF 1*] **by** *simp*
also have (∑ *k*≤*2*. (i ∗ *t*) ^ *k* ∗ *expectation* (λ*x*. *x* ^ *k*) / (*fact k*)) = *1* − *t*^*2* ∗ *σ2* / *2*
by (*simp add*: *complex-eq-iff numeral-eq-Suc integral-1 Re-divide Im-divide*)
also have *fact 3* = *6* **by** (*simp add*: *eval-nat-numeral*)
also have *t*² ∗ *expectation* (λ*x*. *min* (*6* ∗ *x*²) (|*t*| ∗ |*x*| ^ *3*)) / *6* =
*t*² / *6* ∗ *expectation* (λ*x*. *min* (*6* ∗ *x*²) (|*t*| ∗ |*x*| ^ *3*)) **by** (*simp add*: *field-simps*)
finally show *?thesis* **.**
qed

This is a more familiar textbook formulation in terms of random variables, but we will use the previous version for the CLT.

lemma (**in** *prob-space*) *char-approx3′*:
fixes μ :: *real measure* **and** *X*
assumes *rv-X* [*simp*]: *random-variable borel X*
and [*simp*]: *integrable M X integrable M* (λ*x*. (*X x*)^*2*) *expectation X* = *0*
and *var-X*: *variance X* = *σ2*
and *μ-def*: μ = *distr M borel X*
shows *cmod* (*char* μ *t* − (*1* − *t*^*2* ∗ *σ2* / *2*)) ≤
(*t*^*2* / *6*) ∗ *expectation* (λ*x*. *min* (*6* ∗ (*X x*)^*2*) (|*t*| ∗ |*X x*|^*3*))
using *var-X* **unfolding** *μ-def*
apply (*subst integral-distr* [*symmetric*, *OF rv-X*], *simp*)
apply (*intro real-distribution.char-approx3*)
apply (*auto simp add*: *integrable-distr-eq integral-distr*)
done

this is the formulation in the book – in terms of a random variable *with* the distribution, rather the distribution itself. I don't know which is more useful, though in principal we can go back and forth between them.

lemma (**in** *prob-space*) *char-approx1′*:
fixes μ :: *real measure* **and** *X*
assumes *integrable-moments* : ⋀*k*. *k* ≤ *n* ⟹ *integrable M* (λ*x*. *X x* ^ *k*)
and *rv-X*[*measurable*]: *random-variable borel X*
and *μ-distr* : *distr M borel X* = μ
shows *cmod* (*char* μ *t* − (∑ *k* ≤ *n*. ((i ∗ *t*)^*k* / *fact k*) ∗ *expectation* (λ*x*. (*X x*)^*k*))) ≤
(*2* ∗ |*t*|^*n* / *fact n*) ∗ *expectation* (λ*x*. |*X x*|^*n*)
unfolding *μ-distr*[*symmetric*]
apply (*subst* (*1 2*) *integral-distr* [*symmetric*, *OF rv-X*], *simp*, *simp*)
apply (*intro real-distribution.char-approx1*[*of distr M borel X n t*] *real-distribution-distr rv-X*)
apply (*auto simp*: *integrable-distr-eq integrable-moments*)
done

14.5 Calculation of the Characteristic Function of the Standard Distribution

abbreviation
std-normal-distribution ≡ *density lborel std-normal-density*

lemma *real-dist-normal-dist*: *real-distribution std-normal-distribution*
using *prob-space-normal-density* **by** (*auto simp*: *real-distribution-def real-distribution-axioms-def*)

lemma *std-normal-distribution-even-moments*:
fixes *k* :: *nat*
shows (*LINT x*|*std-normal-distribution*. *x*^(2 ∗ *k*)) = *fact* (2 ∗ *k*) / (2^*k* ∗ *fact k*)
and *integrable std-normal-distribution* (λ*x*. *x*^(2 ∗ *k*))
using *integral-std-normal-moment-even*[*of k*]
by (*subst integral-density*)
(*auto simp*: *normal-density-nonneg integrable-density*
intro: *integrable.intros std-normal-moment-even*)

lemma *integrable-std-normal-distribution-moment*: *integrable std-normal-distribution* (λ*x*. *x*^*k*)
by (*auto simp*: *normal-density-nonneg integrable-std-normal-moment integrable-density*)

lemma *integral-std-normal-distribution-moment-odd*:
odd k ⟹ *integral*^{L} *std-normal-distribution* (λ*x*. *x*^*k*) = 0
using *integral-std-normal-moment-odd*[*of* (*k* − 1) *div* 2]
by (*auto simp*: *integral-density normal-density-nonneg elim*: *oddE*)

lemma *std-normal-distribution-even-moments-abs*:
fixes *k* :: *nat*
shows (*LINT x*|*std-normal-distribution*. |*x*|^(2 ∗ *k*)) = *fact* (2 ∗ *k*) / (2^*k* ∗ *fact k*)
using *integral-std-normal-moment-even*[*of k*]
by (*subst integral-density*) (*auto simp*: *normal-density-nonneg power-even-abs*)

lemma *std-normal-distribution-odd-moments-abs*:
fixes *k* :: *nat*
shows (*LINT x*|*std-normal-distribution*. |*x*|^(2 ∗ *k* + 1)) = *sqrt* (2 / *pi*) ∗ 2 ^ *k* ∗ *fact k*
using *integral-std-normal-moment-abs-odd*[*of k*]
by (*subst integral-density*) (*auto simp*: *normal-density-nonneg*)

theorem *char-std-normal-distribution*:
char std-normal-distribution = (λ*t*. *complex-of-real* (*exp* (− (*t*^2) / 2)))
proof (*intro ext LIMSEQ-unique*)
fix *t* :: *real*
let *?f'* = λ*k*. (i ∗ *t*)^*k* / *fact k* ∗ (*LINT x* | *std-normal-distribution*. *x*^*k*)
let *?f* = λ*n*. (∑ *k* ≤ *n*. *?f' k*)
show *?f* ⟶ *exp* (−(*t*^2) / 2)

proof (*rule limseq-even-odd*)
have (i ∗ *complex-of-real t*) ^ (*2* ∗ *a*) / (*2* ^ *a* ∗ *fact a*) = (− ((*complex-of-real t*)² / *2*)) ^ *a* / *fact a* **for** *a*
by (*subst power-mult*) (*simp add*: *field-simps uminus-power-if power-mult*)
then have ∗: *?f* (*2* ∗ *n*) = *complex-of-real* (∑ *k* < *Suc n*. (*1* / *fact k*) ∗ (− (*t*^*2*) / *2*)^*k*) **for** *n* :: *nat*
unfolding *of-real-sum*
by (*intro sum.reindex-bij-witness-not-neutral*[*symmetric*, **where**
i=λ*n*. *n div 2* **and** *j*=λ*n*. *2* ∗ *n* **and** *T'*={*i*. *i* ≤ *2* ∗ *n* ∧ *odd i*} **and** *S'*={}])
(*auto simp*: *integral-std-normal-distribution-moment-odd std-normal-distribution-even-moments*)
show (λ*n*. *?f* (*2* ∗ *n*)) ⟶ *exp* (−(*t*^*2*) / *2*)
unfolding ∗ **using** *exp-converges*[**where** *'a*=*real*]
by (*intro tendsto-of-real LIMSEQ-Suc*) (*auto simp*: *inverse-eq-divide sums-def* [*symmetric*])
have ∗∗: *?f* (*2* ∗ *n* + *1*) = *?f* (*2* ∗ *n*) **for** *n*
proof −
have *?f* (*2* ∗ *n* + *1*) = *?f* (*2* ∗ *n*) + *?f'* (*2* ∗ *n* + *1*)
by *simp*
also have *?f'* (*2* ∗ *n* + *1*) = *0*
by (*subst integral-std-normal-distribution-moment-odd*) *simp-all*
finally show *?f* (*2* ∗ *n* + *1*) = *?f* (*2* ∗ *n*)
by *simp*
qed
show (λ*n*. *?f* (*2* ∗ *n* + *1*)) ⟶ *exp* (−(*t*^*2*) / *2*)
unfolding ∗∗ **by** *fact*
qed

have ∗∗: (λ*n*. *x* ^ *n* / *fact n*) ⟶ *0* **for** *x* :: *real*
using *summable-LIMSEQ-zero* [*OF summable-exp*] **by** (*auto simp add*: *inverse-eq-divide*)

let *?F* = λ*n*. *2* ∗ |*t*| ^ *n* / *fact n* ∗ (*LINT x*|*std-normal-distribution*. |*x*| ^ *n*)

show *?f* ⟶ *char std-normal-distribution t*
proof (*rule metric-tendsto-imp-tendsto*[*OF limseq-even-odd*])
show (λ*n*. *?F* (*2* ∗ *n*)) ⟶ *0*
proof (*rule Lim-transform-eventually*)
show ∀ *F n in sequentially*. *2* ∗ ((*t*^*2* / *2*)^*n* / *fact n*) = *?F* (*2* ∗ *n*)
unfolding *std-normal-distribution-even-moments-abs* **by** (*simp add*: *power-mult power-divide*)
qed (*intro tendsto-mult-right-zero* ∗∗)

have ∗: *?F* (*2* ∗ *n* + *1*) = (*2* ∗ |*t*| ∗ *sqrt* (*2* / *pi*)) ∗ ((*2* ∗ *t*^*2*)^*n* ∗ *fact n* / *fact* (*2* ∗ *n* + *1*)) **for** *n*
unfolding *std-normal-distribution-odd-moments-abs*
by (*simp add*: *field-simps power-mult*[*symmetric*] *power-even-abs*)
have *norm* ((*2* ∗ *t*²) ^ *n* ∗ *fact n* / *fact* (*2* ∗ *n* + *1*)) ≤ (*2* ∗ *t*²) ^ *n* / *fact n* **for** *n*

using *mult-mono*[*OF - square-fact-le-2-fact*, *of 1 1 + 2 * real n n*]
by (*auto simp add*: *divide-simps intro*!: *mult-left-mono*)
then show (λ*n*. *?F* (*2 * n + 1*)) ⟶ *0*
unfolding * **by** (*intro tendsto-mult-right-zero Lim-null-comparison* [*OF - ** [*of 2 * t*^{2}]]) *auto*

show ∀$_F$ *n in sequentially*. *dist* (*?f n*) (*char std-normal-distribution t*) ≤ *dist* (*?F n*) *0*
using *real-distribution.char-approx1*[*OF real-dist-normal-dist integrable-std-normal-distribution-moment*]
by (*auto simp*: *dist-norm integral-nonneg-AE norm-minus-commute*)
qed
qed

end

15 Helly's selection theorem

The set of bounded, monotone, right continuous functions is sequentially compact

theory *Helly-Selection*
imports *HOL−Library.Diagonal-Subsequence Weak-Convergence*
begin

lemma *minus-one-less*: *x − 1 < (x::real)*
by *simp*

theorem *Helly-selection*:
fixes *f* :: *nat* ⇒ *real* ⇒ *real*
assumes *rcont*: ⋀*n x*. *continuous* (*at-right x*) (*f n*)
assumes *mono*: ⋀*n*. *mono* (*f n*)
assumes *bdd*: ⋀*n x*. |*f n x*| ≤ *M*
shows ∃ *s*. *strict-mono* (*s::nat* ⇒ *nat*) ∧ (∃ *F*. (∀ *x*. *continuous* (*at-right x*) *F*) ∧ *mono F* ∧ (∀ *x*. |*F x*| ≤ *M*) ∧
(∀ *x*. *continuous* (*at x*) *F* ⟶ (λ*n*. *f* (*s n*) *x*) ⟶ *F x*))
proof −
obtain *m* :: *real* ⇒ *nat* **where** *bij-betw m* ℚ *UNIV*
using *countable-rat Rats-infinite* **by** (*erule countableE-infinite*)
then obtain *r* :: *nat* ⇒ *real* **where** *bij*: *bij-betw r UNIV* ℚ
using *bij-betw-inv* **by** *blast*

have *dense-r*: ⋀*x y*. *x < y* ⟹ ∃ *n*. *x < r n* ∧ *r n < y*
by (*metis Rats-dense-in-real bij f-the-inv-into-f bij-betw-def*)

let *?P* = λ*n*. λ*s*. *convergent* (λ*k*. *f* (*s k*) (*r n*))
interpret *nat*: *subseqs ?P*
proof (*unfold convergent-def*, *unfold subseqs-def*, *auto*)
fix *n* :: *nat* **and** *s* :: *nat* ⇒ *nat* **assume** *s*: *strict-mono s*
have *bounded* {−*M*..*M*}

using *bounded-closed-interval* **by** *auto*
moreover have $\bigwedge k.\ f\ (s\ k)\ (r\ n) \in \{-M..M\}$
using *bdd* **by** (*simp add*: *abs-le-iff minus-le-iff*)
ultimately have $\exists\, l\ s'.$ *strict-mono* $s' \wedge ((\lambda k.\ f\ (s\ k)\ (r\ n)) \circ s') \longrightarrow l$
using *compact-Icc compact-imp-seq-compact seq-compactE* **by** *metis*
thus $\exists\, s'.$ *strict-mono* $(s'::nat \Rightarrow nat) \wedge (\exists\, l.\ (\lambda k.\ f\ (s\ (s'\ k))\ (r\ n)) \longrightarrow l)$
by (*auto simp*: *comp-def*)
qed
define d **where** $d =$ *nat.diagseq*
have *subseq*: *strict-mono d*
unfolding *d-def* **using** *nat.subseq-diagseq* **by** *auto*
have *rat-cnv*: *?P n d* **for** n
proof −
have *Pn-seqseq*: *?P n* (*nat.seqseq* (*Suc n*))
by (*rule nat.seqseq-holds*)
have *1*: $(\lambda k.\ f\ ((nat.seqseq\ (Suc\ n) \circ (\lambda k.\ nat.fold\text{-}reduce\ (Suc\ n)\ k$
$(Suc\ n + k)))\ k)\ (r\ n)) = (\lambda k.\ f\ (nat.seqseq\ (Suc\ n)\ k)\ (r\ n)) \circ$
$(\lambda k.\ nat.fold\text{-}reduce\ (Suc\ n)\ k\ (Suc\ n + k))$
by *auto*
have *2*: *?P n* $(d \circ ((+)\ (Suc\ n)))$
unfolding *d-def nat.diagseq-seqseq 1*
by (*intro convergent-subseq-convergent Pn-seqseq nat.subseq-diagonal-rest*)
then obtain L **where** *3*: $(\lambda na.\ f\ (d\ (na + Suc\ n))\ (r\ n)) \longrightarrow L$
by (*auto simp*: *add.commute dest*: *convergentD*)
then have $(\lambda k.\ f\ (d\ k)\ (r\ n)) \longrightarrow L$
by (*rule LIMSEQ-offset*)
then show *?thesis*
by (*auto simp*: *convergent-def*)
qed
let *?f* $= \lambda n.\ \lambda k.\ f\ (d\ k)\ (r\ n)$
have *lim-f*: *?f n* $\longrightarrow$ *lim* (*?f n*) **for** n
using *rat-cnv convergent-LIMSEQ-iff* **by** *auto*
have *lim-bdd*: *lim* (*?f n*) $\in \{-M..M\}$ **for** n
proof −
have *closed* $\{-M..M\}$ **using** *closed-real-atLeastAtMost* **by** *auto*
hence $(\forall\, i.$ *?f n i* $\in \{-M..M\}) \wedge$ *?f n* $\longrightarrow$ *lim* (*?f n*) $\longrightarrow$ *lim* (*?f n*) $\in \{-M..M\}$
unfolding *closed-sequential-limits* **by** (*drule-tac* $x = \lambda k.\ f\ (d\ k)\ (r\ n)$ **in** *spec*) *blast*
moreover have $\forall\, i.$ *?f n i* $\in \{-M..M\}$
using *bdd* **by** (*simp add*: *abs-le-iff minus-le-iff*)
ultimately show *lim* (*?f n*) $\in \{-M..M\}$
using *lim-f* **by** *auto*
qed
then have *limset-bdd*: $\bigwedge x.\ \{lim\ (?f\ n)\ |n.\ x < r\ n\} \subseteq \{-M..M\}$
by *auto*
then have *bdd-below*: *bdd-below* $\{lim\ (?f\ n)\ |n.\ x < r\ n\}$ **for** x
by (*metis* (*mono-tags*) *bdd-below-Icc bdd-below-mono*)
have *r-unbdd*: $\exists\, n.\ x < r\ n$ **for** x

```
    using dense-r[OF less-add-one, of x] by auto
  then have nonempty: {lim (?f n) |n. x < r n} ≠ {} for x
    by auto

  define F where F x = Inf {lim (?f n) |n. x < r n} for x
  have F-eq: ereal (F x) = (INF n∈{n. x < r n}. ereal (lim (?f n))) for x
     unfolding F-def by (subst ereal-Inf'[OF bdd-below nonempty]) (simp add:
setcompr-eq-image image-comp)
  have mono-F: mono F
     using nonempty by (auto intro!: cInf-superset-mono simp: F-def bdd-below
mono-def)
  moreover have ⋀x. continuous (at-right x) F
   unfolding continuous-within order-tendsto-iff eventually-at-right[OF less-add-one]
  proof safe
    show F x < u ⟹ ∃ b>x. ∀ y>x. y < b ⟶ F y < u for x u
      unfolding F-def cInf-less-iff[OF nonempty bdd-below] by auto
  next
    show ∃ b>x. ∀ y>x. y < b ⟶ l < F y if l: l < F x for x l
    using less-le-trans[OF l mono-F[THEN monoD, of x]] by (auto intro: less-add-one)
  qed
  moreover
  { fix x
    have F x ∈ {−M..M}
    unfolding F-def using limset-bdd bdd-below r-unbdd by (intro closed-subset-contains-Inf)
auto
    then have |F x| ≤ M by auto }
  moreover have (λn. f (d n) x) ⟶⟶ F x if cts: continuous (at x) F for x
  proof (rule limsup-le-liminf-real)
    show limsup (λn. f (d n) x) ≤ F x
    proof (rule tendsto-lowerbound)
      show (F ⟶⟶ ereal (F x)) (at-right x)
        using cts unfolding continuous-at-split by (auto simp: continuous-within)
      show ∀ F i in at-right x. limsup (λn. f (d n) x) ≤ F i
        unfolding eventually-at-right[OF less-add-one]
      proof (rule, rule, rule less-add-one, safe)
        fix y assume y: x < y
        with dense-r obtain N where x < r N r N < y by auto
        have ∗: y < r n' ⟹ lim (?f N) ≤ lim (?f n') for n'
            using ‹r N < y› by (intro LIMSEQ-le[OF lim-f lim-f]) (auto intro!:
mono[THEN monoD])
        have limsup (λn. f (d n) x) ≤ limsup (?f N)
              using ‹x < r N› by (auto intro!: Limsup-mono always-eventually
mono[THEN monoD])
        also have ... = lim (λn. ereal (?f N n))
          using rat-cnv[of N] by (force intro!: convergent-limsup-cl simp: conver-
gent-def)
        also have ... ≤ F y
          by (auto intro!: INF-greatest ∗ simp: convergent-real-imp-convergent-ereal
rat-cnv F-eq)
```

finally show *limsup* $(\lambda n.\ f\ (d\ n)\ x) \leq F\ y$ **.**
qed
qed *simp*
show $F\ x \leq$ *liminf* $(\lambda n.\ f\ (d\ n)\ x)$
proof (*rule tendsto-upperbound*)
show $(F \longrightarrow$ *ereal* $(F\ x))$ (*at-left x*)
using *cts* **unfolding** *continuous-at-split* **by** (*auto simp*: *continuous-within*)
show $\forall_F\ i$ *in at-left x*. $F\ i \leq$ *liminf* $(\lambda n.\ f\ (d\ n)\ x)$
unfolding *eventually-at-left*[*OF minus-one-less*]
proof (*rule*, *rule*, *rule minus-one-less*, *safe*)
fix y **assume** y: $y < x$
with *dense-r* **obtain** N **where** $y < r\ N\ r\ N < x$ **by** *auto*
have $F\ y \leq$ *liminf* (*?f N*)
using ‹$y < r\ N$› **by** (*auto simp*: *F-eq convergent-real-imp-convergent-ereal rat-cnv convergent-liminf-cl intro*!: *INF-lower2*)
also have $\ldots \leq$ *liminf* $(\lambda n.\ f\ (d\ n)\ x)$
using ‹$r\ N < x$› **by** (*auto intro*!: *Liminf-mono monoD*[*OF mono*] *always-eventually*)
finally show $F\ y \leq$ *liminf* $(\lambda n.\ f\ (d\ n)\ x)$ **.**
qed
qed *simp*
qed
ultimately show *?thesis* **using** *subseq* **by** *auto*
qed

definition
tight :: (*nat* ⇒ *real measure*) ⇒ *bool*
where
tight $\mu \equiv (\forall n.$ *real-distribution* $(\mu\ n)) \wedge (\forall (\varepsilon$::*real*$)>0.\ \exists a\ b$::*real*. $a < b \wedge (\forall n.$ *measure* $(\mu\ n)\ \{a<..b\} > 1 - \varepsilon))$

theorem *tight-imp-convergent-subsubsequence*:
assumes μ: *tight* μ *strict-mono s*
shows $\exists r\ M.$ *strict-mono* $(r :: nat \Rightarrow nat) \wedge$ *real-distribution M* $\wedge$ *weak-conv-m* $(\mu \circ s \circ r)\ M$
proof −
define f **where** $f\ k =$ *cdf* $(\mu\ (s\ k))$ **for** k
interpret μ: *real-distribution* $\mu\ k$ **for** k
using μ **unfolding** *tight-def* **by** *auto*

have *rcont*: $\bigwedge x.$ *continuous* (*at-right x*) (*f k*)
and *mono*: *mono* (*f k*)
and *top*: $(f\ k \longrightarrow 1)$ *at-top*
and *bot*: $(f\ k \longrightarrow 0)$ *at-bot* **for** k
unfolding *f-def mono-def*
using μ*.cdf-nondecreasing* μ*.cdf-is-right-cont* μ*.cdf-lim-at-top-prob* μ*.cdf-lim-at-bot*

by *auto*
have *bdd*: $|f\ k\ x| \leq 1$ **for** $k\ x$
by (*auto simp add*: *abs-le-iff minus-le-iff f-def* μ*.cdf-nonneg* μ*.cdf-bounded-prob*)

from *Helly-selection*[*OF rcont mono bdd*, *of* $\lambda x.\ x$] **obtain** $r\ F$
where F: *strict-mono* r $\bigwedge x.$ *continuous* (*at-right* x) F *mono* F $\bigwedge x.\ |F\ x| \leq 1$
and *lim-F*: $\bigwedge x.$ *continuous* (*at* x) $F \Longrightarrow (\lambda n.\ f\ (r\ n)\ x) \longrightarrow F\ x$
by *blast*

have $0 \leq f\ n\ x$ **for** $n\ x$
unfolding *f-def* **by** (*rule* μ*.cdf-nonneg*)
have *F-nonneg*: $0 \leq F\ x$ **for** x
proof −
obtain y **where** $y < x$ *isCont* $F\ y$
using *open-minus-countable*[*OF mono-ctble-discont*[*OF* ‹*mono* F›], *of* {..< x}] **by** *auto*
then have $0 \leq F\ y$
by (*intro LIMSEQ-le-const*[*OF lim-F*]) (*auto simp*: *f-def* μ*.cdf-nonneg*)
also have $\ldots \leq F\ x$
using ‹$y < x$› **by** (*auto intro*!: *monoD*[*OF* ‹*mono* F›])
finally show $0 \leq F\ x$ **.**
qed

have *Fab*: $\exists\, a\ b.\ (\forall\, x{\geq}b.\ F\ x \geq 1 - \varepsilon) \wedge (\forall\, x{\leq}a.\ F\ x \leq \varepsilon)$ **if** ε: $0 < \varepsilon$ **for** ε
proof *auto*
obtain $a'\ b'$ **where** $a'b'$: $a' < b'$ $\bigwedge k.$ *measure* $(\mu\ k)$ {a'<..b'} $> 1 - \varepsilon$
using ε μ **by** (*auto simp*: *tight-def*)
obtain a **where** a: $a < a'$ *isCont* $F\ a$
using *open-minus-countable*[*OF mono-ctble-discont*[*OF* ‹*mono* F›], *of* {..< a'}] **by** *auto*
obtain b **where** b: $b' < b$ *isCont* $F\ b$
using *open-minus-countable*[*OF mono-ctble-discont*[*OF* ‹*mono* F›], *of* {b' <..}] **by** *auto*
have $a < b$
using $a\ b\ a'b'$ **by** *simp*

let $?\mu = \lambda k.$ *measure* $(\mu\ (s\ (r\ k)))$
have *ab*: $?\mu\ k$ {a<..b} $> 1 - \varepsilon$ **for** k
proof −
have $?\mu\ k$ {a'<..b'} $\leq ?\mu\ k$ {a<..b}
using $a\ b$ **by** (*intro* μ*.finite-measure-mono*) *auto*
then show *?thesis*
using $a'b'(2)$ **by** (*metis less-eq-real-def less-trans*)
qed

have $(\lambda k.\ ?\mu\ k$ {..b}) $\longrightarrow F\ b$
using $b(2)$ *lim-F* **unfolding** *f-def cdf-def o-def* **by** *auto*
then have $1 - \varepsilon \leq F\ b$
proof (*rule tendsto-lowerbound*, *intro always-eventually allI*)

fix k
have $1 - \varepsilon < ?\mu\ k\ \{a<..b\}$
using *ab* **by** *auto*
also have $\ldots \leq ?\mu\ k\ \{..b\}$
by (*auto intro*!: *μ.finite-measure-mono*)
finally show $1 - \varepsilon \leq ?\mu\ k\ \{..b\}$
by (*rule less-imp-le*)
qed (*rule sequentially-bot*)
then show $\exists\, b.\ \forall\, x{\geq}b.\ 1 - \varepsilon \leq F\ x$
using F **unfolding** *mono-def* **by** (*metis order.trans*)

have $(\lambda k.\ ?\mu\ k\ \{..a\}) \longrightarrow F\ a$
using *a*(*2*) *lim-F* **unfolding** *f-def cdf-def o-def* **by** *auto*
then have *Fa*: $F\ a \leq \varepsilon$
proof (*rule tendsto-upperbound*, *intro always-eventually allI*)
fix k
have $?\mu\ k\ \{..a\} + ?\mu\ k\ \{a<..b\} \leq 1$
by (*subst μ.finite-measure-Union*[*symmetric*]) *auto*
then show $?\mu\ k\ \{..a\} \leq \varepsilon$
using *ab*[*of k*] **by** *simp*
qed (*rule sequentially-bot*)
then show $\exists\, a.\ \forall\, x{\leq}a.\ F\ x \leq \varepsilon$
using F **unfolding** *mono-def* **by** (*metis order.trans*)
qed

have $(F \longrightarrow 1)$ *at-top*
proof (*rule order-tendstoI*)
show $1 < y \Longrightarrow \forall_F\ x$ *in at-top*. $F\ x < y$ **for** y
using ‹$\bigwedge x.\ |F\ x| \leq 1$› ‹$\bigwedge x.\ 0 \leq F\ x$› **by** (*auto intro*: *le-less-trans always-eventually*)
fix y :: *real* **assume** $y < 1$
then obtain z **where** $y < z\ z < 1$
using *dense*[*of y 1*] **by** *auto*
with *Fab*[*of 1 − z*] **show** $\forall_F\ x$ *in at-top*. $y < F\ x$
by (*auto simp*: *eventually-at-top-linorder intro*: *less-le-trans*)
qed
moreover
have $(F \longrightarrow 0)$ *at-bot*
proof (*rule order-tendstoI*)
show $y < 0 \Longrightarrow \forall_F\ x$ *in at-bot*. $y < F\ x$ **for** y
using ‹$\bigwedge x.\ 0 \leq F\ x$› **by** (*auto intro*: *less-le-trans always-eventually*)
fix y :: *real* **assume** $0 < y$
then obtain z **where** $0 < z\ z < y$
using *dense*[*of 0 y*] **by** *auto*
with *Fab*[*of z*] **show** $\forall_F\ x$ *in at-bot*. $F\ x < y$
by (*auto simp*: *eventually-at-bot-linorder intro*: *le-less-trans*)
qed
ultimately have M: *real-distribution* (*interval-measure F*) *cdf* (*interval-measure F*) $= F$

using *F* **by** (*auto intro*!: *real-distribution-interval-measure cdf-interval-measure simp*: *mono-def*)
with *lim-F LIMSEQ-subseq-LIMSEQ M* **have** *weak-conv-m* (*μ* ∘ *s* ∘ *r*) (*interval-measure F*)
by (*auto simp*: *weak-conv-def weak-conv-m-def f-def comp-def*)
then show ∃ *r M*. *strict-mono* (*r* :: *nat* ⇒ *nat*) ∧ (*real-distribution M* ∧ *weak-conv-m* (*μ* ∘ *s* ∘ *r*) *M*)
using *F M* **by** *auto*
qed

corollary *tight-subseq-weak-converge*:
fixes *μ* :: *nat* ⇒ *real measure* **and** *M* :: *real measure*
assumes ⋀*n*. *real-distribution* (*μ n*) *real-distribution M* **and** *tight*: *tight μ* **and**
subseq: ⋀*s ν*. *strict-mono s* ⟹ *real-distribution ν* ⟹ *weak-conv-m* (*μ* ∘ *s*) *ν* ⟹ *weak-conv-m* (*μ* ∘ *s*) *M*
shows *weak-conv-m μ M*
proof (*rule ccontr*)
define *f* **where** *f n* = *cdf* (*μ n*) **for** *n*
define *F* **where** *F* = *cdf M*

assume ¬ *weak-conv-m μ M*
then obtain *x* **where** *x*: *isCont F x* ¬ (*λn*. *f n x*) ⟶ *F x*
by (*auto simp*: *weak-conv-m-def weak-conv-def f-def F-def*)
then obtain *ε* **where** *ε* > *0* **and** *infinite* {*n*. ¬ *dist* (*f n x*) (*F x*) < *ε*}
by (*auto simp*: *tendsto-iff not-eventually INFM-iff-infinite cofinite-eq-sequentially*[*symmetric*])
then obtain *s* :: *nat* ⇒ *nat* **where** *s*: ⋀*n*. ¬ *dist* (*f* (*s n*) *x*) (*F x*) < *ε* **and** *strict-mono s*
using *enumerate-in-set enumerate-mono* **by** (*fastforce simp*: *strict-mono-def*)
then obtain *r ν* **where** *r*: *strict-mono r real-distribution ν weak-conv-m* (*μ* ∘ *s* ∘ *r*) *ν*
using *tight-imp-convergent-subsubsequence*[*OF tight*] **by** *blast*
then have *weak-conv-m* (*μ* ∘ (*s* ∘ *r*)) *M*
using ‹*strict-mono s*› *r* **by** (*intro subseq strict-mono-o*) (*auto simp*: *comp-assoc*)
then have (*λn*. *f* (*s* (*r n*)) *x*) ⟶ *F x*
using *x* **by** (*auto simp*: *weak-conv-m-def weak-conv-def F-def f-def*)
then show *False*
using *s* ‹*ε* > *0*› **by** (*auto dest*: *tendstoD*)
qed

end

16 Integral of sinc

theory *Sinc-Integral*
imports *Distributions*
begin

16.1 Various preparatory integrals

Naming convention The theorem name consists of the following parts:

- Kind of integral: *has-bochner-integral / integrable / LBINT*
- Interval: Interval (0 / infinity / open / closed) (infinity / open / closed)
- Name of the occurring constants: power, exp, m (for minus), scale, sin, ...

lemma *has-bochner-integral-I0i-power-exp-m'*:
has-bochner-integral lborel (*λx. x^k * exp* (*−x*) *∗ indicator* {*0 ..*} *x*::*real*) (*fact k*)
using *nn-intergal-power-times-exp-Ici*[*of k*]
by (*intro has-bochner-integral-nn-integral*)
(*auto simp*: *nn-integral-set-ennreal split*: *split-indicator*)

lemma *has-bochner-integral-I0i-power-exp-m*:
has-bochner-integral lborel (*λx. x^k * exp* (*−x*) *∗ indicator* {*0 <..*} *x*::*real*) (*fact k*)
using *AE-lborel-singleton*[*of 0*]
by (*intro has-bochner-integral-cong-AE*[*THEN iffD1*, *OF - - - has-bochner-integral-I0i-power-exp-m'*])
(*auto split*: *split-indicator*)

lemma *integrable-I0i-exp-mscale*: *0 < u*::*real* *⟹ set-integrable lborel* {*0 <..*} (*λx. exp* (*−*(*x ∗ u*)))
using *lborel-integrable-real-affine-iff*[*of u λx. indicator* {*0 <..*} *x ∗R exp* (*− x*) *0*]
has-bochner-integral-I0i-power-exp-m[*of 0*]
by (*simp add*: *indicator-def zero-less-mult-iff mult-ac integrable.intros set-integrable-def*)

lemma *LBINT-I0i-exp-mscale*: *0 < u*::*real* *⟹ LBINT x*=*0..∞. exp* (*−*(*x ∗ u*)) = *1 / u*
using *lborel-integrable-real-affine*[*of u λx. indicator* {*0<..*} *x ∗R exp* (*− x*) *0*]
has-bochner-integral-I0i-power-exp-m[*of 0*]
by (*auto simp*: *indicator-def zero-less-mult-iff interval-lebesgue-integral-0-infty set-lebesgue-integral-def field-simps*
dest!: *has-bochner-integral-integral-eq*)

lemma *LBINT-I0c-exp-mscale-sin*:
LBINT x=*0..t. exp* (*−*(*u ∗ x*)) *∗ sin x* =
(*1 / *(*1 + u^2*)) *∗* (*1 − exp* (*−*(*u ∗ t*)) *∗* (*u ∗ sin t* + *cos t*)) (**is** *- = ?F t*)
unfolding *zero-ereal-def*
proof (*subst interval-integral-FTC-finite*)
show (*?F has-vector-derivative exp* (*−* (*u ∗ x*)) *∗ sin x*) (*at x within* {*min 0 t..max 0 t*}) **for** *x*
by (*auto intro*!: *derivative-eq-intros*
simp: *has-field-derivative-iff-has-vector-derivative*[*symmetric*] *power2-eq-square*)
(*simp-all add*: *field-simps add-nonneg-eq-0-iff*)

qed (*auto intro*: *continuous-at-imp-continuous-on*)

lemma *LBINT-I0i-exp-mscale-sin*:
assumes $0 < x$
shows *LBINT* $u{=}0..\infty.\ |exp\ (-u * x) * sin\ x| = |sin\ x|\ /\ x$
proof (*subst interval-integral-FTC-nonneg*)
let *?F* $= \lambda u.\ 1\ /\ x * (1 - exp\ (-\ u * x)) * |sin\ x|$
show $\bigwedge t.$ (*?F has-real-derivative* $|exp\ (-\ t * x) * sin\ x|$) (*at t*)
using ‹$0 < x$› **by** (*auto intro*!: *derivative-eq-intros simp*: *abs-mult*)
show ((*?F* ∘ *real-of-ereal*) ⟶ 0) (*at-right 0*)
using ‹$0 < x$› **by** (*auto simp*: *zero-ereal-def ereal-tendsto-simps intro*!: *tendsto-eq-intros*)
have ∗: ((λ*t*. *exp* (− *t* ∗ *x*)) ⟶ 0) *at-top*
using ‹$0 < x$›
by (*auto intro*!: *exp-at-bot*[*THEN filterlim-compose*] *filterlim-tendsto-pos-mult-at-top filterlim-ident*
simp: *filterlim-uminus-at-bot mult.commute*[*of* - *x*])
show ((*?F* ∘ *real-of-ereal*) ⟶ $|sin\ x|\ /\ x$) (*at-left* ∞)
using ‹$0 < x$› **unfolding** *ereal-tendsto-simps*
by (*intro filterlim-compose*[*OF* - ∗]) (*auto intro*!: *tendsto-eq-intros filterlim-ident*)
qed *auto*

lemma
shows *integrable-inverse-1-plus-square*:
set-integrable lborel (*einterval* (−∞) ∞) (λ*x*. *inverse* (*1* + *x*^*2*))
and *LBINT-inverse-1-plus-square*:
LBINT $x{=}-\infty..\infty.$ *inverse* (*1* + *x*^*2*) = *pi*
proof −
have *1*: − (*pi* / *2*) < *x* ⟹ *x* ∗ *2* < *pi* ⟹ (*tan has-real-derivative 1* + (*tan* $x)^2$) (*at x*) **for** *x*
using *cos-gt-zero-pi*[*of x*] **by** (*subst tan-sec*) (*auto intro*!: *DERIV-tan simp*: *power-inverse*)
have *2*: − (*pi* / *2*) < *x* ⟹ *x* ∗ *2* < *pi* ⟹ *isCont* (λ*x*. *1* + (*tan x*)^2) *x* **for** *x*
using *cos-gt-zero-pi*[*of x*] **by** *auto*
have *3*: ⟦− (*pi* / *2*) < *x*; *x* ∗ *2* < *pi*⟧ ⟹ *isCont* (λ*x*. *inverse* (*1* + *x*^{2})) (*tan x*) **for** *x*
by (*rule continuous-intros* | *simp add*: *add-nonneg-eq-0-iff*)+
show *LBINT* $x{=}-\infty..\infty.$ *inverse* (*1* + *x*^*2*) = *pi*
by (*subst interval-integral-substitution-nonneg*[*of* −*pi*/*2 pi*/*2 tan* λ*x*. *1* + (*tan x*)^*2*])
(*auto intro*: *derivative-eq-intros 1 2 3 filterlim-tan-at-right*
simp add: *ereal-tendsto-simps filterlim-tan-at-left add-nonneg-eq-0-iff set-integrable-def*)
show *set-integrable lborel* (*einterval* (−∞) ∞) (λ*x*. *inverse* (*1* + *x*^*2*))
by (*subst interval-integral-substitution-nonneg*[*of* −*pi*/*2 pi*/*2 tan* λ*x*. *1* + (*tan x*)^*2*])
(*auto intro*: *derivative-eq-intros 1 2 3 filterlim-tan-at-right*
simp add: *ereal-tendsto-simps filterlim-tan-at-left add-nonneg-eq-0-iff set-integrable-def*)

qed

lemma
shows *integrable-I0i-1-div-plus-square*:
interval-lebesgue-integrable lborel 0 ∞ (λx. 1 / (1 + x^2))
and *LBINT-I0i-1-div-plus-square*:
LBINT x=0..∞. 1 / (1 + x^2) = pi / 2
proof −
have *1*: *0 < x ⟹ x * 2 < pi ⟹ (tan has-real-derivative 1 + (tan x)^2) (at x)*
for *x*
using *cos-gt-zero-pi*[*of x*] **by** (*subst tan-sec*) (*auto intro*!: *DERIV-tan simp*: *power-inverse*)
have *2*: *0 < x ⟹ x * 2 < pi ⟹ isCont (λx. 1 + (tan x)^2) x* **for** *x*
using *cos-gt-zero-pi*[*of x*] **by** *auto*
show *LBINT x=0..∞. 1 / (1 + x^2) = pi / 2*
by (*subst interval-integral-substitution-nonneg*[*of 0 pi/2 tan λx. 1 + (tan x)^2*])
(*auto intro*: *derivative-eq-intros 1 2 tendsto-eq-intros*
simp add: *ereal-tendsto-simps filterlim-tan-at-left zero-ereal-def add-nonneg-eq-0-iff set-integrable-def*)
show *interval-lebesgue-integrable lborel 0 ∞ (λx. 1 / (1 + x^2))*
unfolding *interval-lebesgue-integrable-def*
by (*subst interval-integral-substitution-nonneg*[*of 0 pi/2 tan λx. 1 + (tan x)^2*])
(*auto intro*: *derivative-eq-intros 1 2 tendsto-eq-intros*
simp add: *ereal-tendsto-simps filterlim-tan-at-left zero-ereal-def add-nonneg-eq-0-iff set-integrable-def*)
qed

17 The sinc function, and the sine integral (Si)

abbreviation *sinc* :: *real ⇒ real* **where**
sinc ≡ (λx. if x = 0 then 1 else sin x / x)

lemma *sinc-at-0*: *((λx. sin x / x::real) ⟶ 1) (at 0)*
using *DERIV-sin* [*of 0*] **by** (*auto simp add*: *has-field-derivative-def field-has-derivative-at*)

lemma *isCont-sinc*: *isCont sinc x*
proof *cases*
assume *x = 0* **then show** *?thesis*
using *LIM-equal* [**where** *g = λx. sin x / x* **and** *a=0* **and** *f=sinc* **and** *l=1*]
by (*auto simp*: *isCont-def sinc-at-0*)
next
assume *x ≠ 0* **show** *?thesis*
by (*rule continuous-transform-within* [**where** *d = abs x* **and** *f = λx. sin x / x*])
(*auto simp add*: *dist-real-def ‹x ≠ 0›*)
qed

lemma *continuous-on-sinc*[*continuous-intros*]:
continuous-on S f ⟹ continuous-on S (λx. sinc (f x))

using *continuous-on-compose*[*of S f sinc*, *OF - continuous-at-imp-continuous-on*]
by (*auto simp*: *isCont-sinc*)

lemma *borel-measurable-sinc*[*measurable*]: *sinc* ∈ *borel-measurable borel*
by (*intro borel-measurable-continuous-onI continuous-at-imp-continuous-on ballI isCont-sinc*)

lemma *sinc-AE*: *AE x in lborel. sin x / x = sinc x*
by (*rule AE-I* [**where** *N* = {*0*}], *auto*)

definition *Si* :: *real* ⇒ *real* **where** *Si t* ≡ *LBINT x=0..t. sin x / x*

lemma *sinc-neg* [*simp*]: *sinc* (− *x*) = *sinc x*
by *auto*

lemma *Si-alt-def* : *Si t = LBINT x=0..t. sinc x*
proof *cases*
assume *0* ≤ *t* **then show** *?thesis*
using *AE-lborel-singleton*[*of 0*]
by (*auto simp*: *Si-def intro*!: *interval-lebesgue-integral-cong-AE*)
next
assume ¬ *0* ≤ *t* **then show** *?thesis*
unfolding *Si-def* **using** *AE-lborel-singleton*[*of 0*]
by (*subst* (*1 2*) *interval-integral-endpoints-reverse*)
(*auto simp*: *Si-def intro*!: *interval-lebesgue-integral-cong-AE*)
qed

lemma *Si-neg*:
assumes *T* ≥ *0* **shows** *Si* (− *T*) = − *Si T*
proof −
have *LBINT x=ereal 0..T.* −*1* ∗*R sinc* (− *x*) = *LBINT y= ereal* (− *0*)..*ereal* (− *T*). *sinc y*
by (*rule interval-integral-substitution-finite* [*OF assms*])
(*auto intro*: *derivative-intros continuous-at-imp-continuous-on isCont-sinc*)
also have (*LBINT x=ereal 0..T.* −*1* ∗*R sinc* (− *x*)) = −(*LBINT x=ereal 0..T. sinc x*)
by (*subst sinc-neg*) (*simp-all add*: *interval-lebesgue-integral-uminus*)
finally have ∗: −(*LBINT x=ereal 0..T. sinc x*) = *LBINT y= ereal 0..ereal* (− *T*). *sinc y*
by *simp*
show *?thesis*
using *assms* **unfolding** *Si-alt-def*
by (*subst zero-ereal-def*)+ (*auto simp add*: ∗ [*symmetric*])
qed

lemma *integrable-sinc′*:
interval-lebesgue-integrable lborel (*ereal 0*) (*ereal T*) (λ*t. sin* (*t* ∗ *ϑ*) / *t*)
proof −

have $*$: *interval-lebesgue-integrable lborel* (*ereal 0*) (*ereal T*) ($\lambda t.\ \vartheta * sinc\ (t * \vartheta)$)
by (*intro interval-lebesgue-integrable-mult-right interval-integrable-isCont continuous-within-compose3* [*OF isCont-sinc*])
auto
show *?thesis*
by (*rule interval-lebesgue-integrable-cong-AE*[*THEN iffD1*, *OF - - - **]) (*insert AE-lborel-singleton*[*of 0*], *auto*)
qed

lemma *DERIV-Si*: (*Si has-real-derivative sinc x*) (*at x*)
proof −
have (*at x within* {*min 0* (*x* − *1*)*..max 0* (*x* + *1*)}) = *at x*
by (*intro at-within-interior*) *auto*
moreover have *min 0* (*x* − *1*) ≤ *x x* ≤ *max 0* (*x* + *1*)
by *auto*
ultimately show *?thesis*
using *interval-integral-FTC2*[*of min 0* (*x* − *1*) *0 max 0* (*x* + *1*) *sinc x*]
by (*auto simp*: *continuous-at-imp-continuous-on isCont-sinc Si-alt-def*[*abs-def*] *zero-ereal-def*
has-field-derivative-iff-has-vector-derivative
split del: *if-split*)
qed

lemma *isCont-Si*: *isCont Si x*
using *DERIV-Si* **by** (*rule DERIV-isCont*)

lemma *borel-measurable-Si*[*measurable*]: *Si* ∈ *borel-measurable borel*
by (*auto intro*: *isCont-Si continuous-at-imp-continuous-on borel-measurable-continuous-onI*)

lemma *Si-at-top-LBINT*:
((λ*t*. (*LBINT x*=*0*..∞. *exp* (−(*x* ∗ *t*)) ∗ (*x* ∗ *sin t* + *cos t*) / (*1* + *x*^*2*))) ⟶ *0*) *at-top*
proof −
let *?F* = λ*x t*. *exp* (− (*x* ∗ *t*)) ∗ (*x* ∗ *sin t* + *cos t*) / (*1* + x^2) :: *real*
have *int*: *set-integrable lborel* {*0*<..} (λ*x*. *exp* (− *x*) ∗ (*x* + *1*) :: *real*)
unfolding *distrib-left*
using *has-bochner-integral-I0i-power-exp-m*[*of 0*] *has-bochner-integral-I0i-power-exp-m*[*of 1*]
by (*intro set-integral-add*) (*auto dest*!: *integrable.intros simp*: *ac-simps set-integrable-def*)

have ((λ*t*::*real*. *LBINT x*:{*0*<..}. *?F x t*) ⟶ *LBINT x*::*real*:{*0*<..}. *0*) *at-top*
unfolding *set-lebesgue-integral-def*
proof (*rule integral-dominated-convergence-at-top*[*OF - - int* [*unfolded set-integrable-def*]],
simp-all del: *abs-divide split*: *split-indicator*)
have $*$: *0* < *x* ⟹ |*x* ∗ *sin t* + *cos t*| / (*1* + x^2) ≤ (*x* ∗ *1* + *1*) / *1* **for** *x t* :: *real*

by (*intro frac-le abs-triangle-ineq*[*THEN order-trans*] *add-mono*)
(*auto simp add*: *abs-mult simp del*: *mult-1-right intro*!: *mult-mono*)
then have $1 \leq t \Longrightarrow 0 < x \Longrightarrow |?F\ x\ t| \leq exp\ (-\ x) * (x + 1)$ **for** $x\ t$ :: *real*
by (*auto simp add*: *abs-mult times-divide-eq-right*[*symmetric*] *simp del*: *times-divide-eq-right*
intro!: *mult-mono*)
then show $\forall_F\ i$ *in at-top. AE x in lborel.* $0 < x \longrightarrow |?F\ x\ i| \leq exp\ (-\ x) * (x + 1)$
by (*auto intro*: *eventually-mono eventually-ge-at-top*[*of 1*::*real*])
show *AE x in lborel.* $0 < x \longrightarrow (?F\ x \longrightarrow 0)$ *at-top*
proof (*intro always-eventually impI allI*)
fix x :: *real* **assume** $0 < x$
show $((\lambda t.\ exp\ (-\ (x * t)) * (x * sin\ t + cos\ t) / (1 + x^2)) \longrightarrow 0)$ *at-top*
proof (*rule Lim-null-comparison*)
show $\forall_F\ t$ *in at-top. norm* $(?F\ x\ t) \leq exp\ (-\ (x * t)) * (x + 1)$
using *eventually-ge-at-top*[*of 1*::*real*] ‹$0 < x$›
by (*auto simp add*: *abs-mult times-divide-eq-right*[*symmetric*] *simp del*: *times-divide-eq-right*
intro!: *mult-mono elim*: *eventually-mono*)
show $((\lambda t.\ exp\ (-\ (x * t)) * (x + 1)) \longrightarrow 0)$ *at-top*
by (*auto simp*: *filterlim-uminus-at-top* [*symmetric*]
intro!: *filterlim-tendsto-pos-mult-at-top*[*OF tendsto-const*] ‹$0<x$›
filterlim-ident
tendsto-mult-left-zero filterlim-compose[*OF exp-at-bot*])
qed
qed
qed
then show $((\lambda t.\ (LBINT\ x{=}0..\infty.\ exp\ (-(x * t)) * (x * sin\ t + cos\ t) / (1 + x\hat{}2))) \longrightarrow 0)$ *at-top*
by (*simp add*: *interval-lebesgue-integral-0-infty set-lebesgue-integral-def*)
qed

lemma *Si-at-top-integrable*:
assumes $t \geq 0$
shows *interval-lebesgue-integrable lborel* $0\ \infty$ $(\lambda x.\ exp\ (-\ (x * t)) * (x * sin\ t + cos\ t) / (1 + x^2))$
using ‹$0 \leq t$› **unfolding** *le-less*
proof
assume $0 = t$ **then show** *?thesis*
using *integrable-I0i-1-div-plus-square* **by** *simp*
next
assume [*arith*]: $0 < t$
have $*$: $0 \leq a \Longrightarrow 0 < x \Longrightarrow a / (1 + x^2) \leq a$ **for** $a\ x$ :: *real*
using *zero-le-power2*[*of x, arith*] **using** *divide-left-mono*[*of 1 1+x*^{2} *a*] **by** *auto*
have *set-integrable lborel* $\{0<..\}$ $(\lambda x.\ (exp\ (-\ x) * x) * (sin\ t/t) + exp\ (-\ x) * cos\ t)$
using *has-bochner-integral-I0i-power-exp-m*[*of 0*] *has-bochner-integral-I0i-power-exp-m*[*of 1*]
by (*intro set-integral-add set-integrable-mult-left*)

```
      (auto dest!: integrable.intros simp: ac-simps set-integrable-def)
  from lborel-integrable-real-affine[OF this [unfolded set-integrable-def], of t 0]
  show ?thesis
    unfolding interval-lebesgue-integral-0-infty set-integrable-def
    by (rule Bochner-Integration.integrable-bound) (auto simp: field-simps * split:
split-indicator)
qed

lemma Si-at-top: (Si ⟶ pi / 2) at-top
proof −
  have †: ∀F t in at-top. pi / 2 − (LBINT u=0..∞. exp (−(u * t)) * (u * sin t
+ cos t) / (1 + u^2)) = Si t
    using eventually-ge-at-top[of 0]
  proof eventually-elim
    fix t :: real assume t ≥ 0
    have Si t = LBINT x=0..t. sin x * (LBINT u=0..∞. exp (−(u * x)))
      unfolding Si-def using ‹0 ≤ t›
      by (intro interval-integral-discrete-difference[where X={0}]) (auto simp:
LBINT-I0i-exp-mscale)
    also have ... = LBINT x. (LBINT u=ereal 0..∞. indicator {0 <..< t} x *R
sin x * exp (−(u * x)))
      using ‹0≤t› by (simp add: zero-ereal-def interval-lebesgue-integral-le-eq
mult-ac set-lebesgue-integral-def)
    also have ... = LBINT x. (LBINT u. indicator ({0<..} × {0 <..< t}) (u, x)
*R (sin x * exp (−(u * x))))
      apply (subst interval-integral-Ioi)
      unfolding set-lebesgue-integral-def by(simp-all add: indicator-times ac-simps
)
    also have ... = LBINT u. (LBINT x. indicator ({0<..} × {0 <..< t}) (u, x)
*R (sin x * exp (−(u * x))))
    proof (intro lborel-pair.Fubini-integral[symmetric] lborel-pair.Fubini-integrable)
      show (λ(x, y). indicator ({0<..} × {0<..<t}) (y, x) *R (sin x * exp (− (y
* x))))
        ∈ borel-measurable (lborel ⨂M lborel) (is ?f ∈ borel-measurable -)
        by measurable

      have AE x in lborel. indicator {0..t} x *R abs (sinc x) = LBINT y. norm (?f
(x, y))
        using AE-lborel-singleton[of 0] AE-lborel-singleton[of t]
      proof eventually-elim
        fix x :: real assume x: x ≠ 0 x ≠ t
        have LBINT y. |indicator ({0<..} × {0<..<t}) (y, x) *R (sin x * exp (−
(y * x)))| =
          LBINT y. |sin x| * exp (− (y * x)) * indicator {0<..} y * indicator
{0<..<t} x
          by (intro Bochner-Integration.integral-cong) (auto split: split-indicator
simp: abs-mult)
        also have ... = |sin x| * indicator {0<..<t} x * (LBINT y=0..∞. exp (−
(y * x)))
```

by (*cases* $x > 0$)
(*auto simp*: *field-simps interval-lebesgue-integral-0-infty set-lebesgue-integral-def split*: *split-indicator*)
also have $\ldots = |\sin x| * \mathit{indicator}\ \{0<..<t\}\ x * (1\ /\ x)$
by (*cases* $x > 0$) (*auto simp add*: *LBINT-I0i-exp-mscale*)
also have $\ldots = \mathit{indicator}\ \{0..t\}\ x *_R |\mathit{sinc}\ x|$
using x **by** (*simp add*: *field-simps split*: *split-indicator*)
finally show *indicator* $\{0..t\}\ x *_R \mathit{abs}\ (\mathit{sinc}\ x) = \mathit{LBINT}\ y.\ \mathit{norm}\ (?f\ (x, y))$
by *simp*
qed
moreover have *integrable lborel* $(\lambda x.\ \mathit{indicat\text{-}real}\ \{0..t\}\ x *_R |\mathit{sinc}\ x|)$
by (*auto intro*!: *borel-integrable-compact continuous-intros simp del*: *real-scaleR-def*)
ultimately show *integrable lborel* $(\lambda x.\ \mathit{LBINT}\ y.\ \mathit{norm}\ (?f\ (x, y)))$
by (*rule integrable-cong-AE-imp*[*rotated 2*]) *simp*

have $0 < x \Longrightarrow \mathit{set\text{-}integrable\ lborel}\ \{0<..\}\ (\lambda y.\ \sin x * \exp\ (-\ (y * x)))$ **for** $x :: \mathit{real}$
by (*intro set-integrable-mult-right integrable-I0i-exp-mscale*)
then show *AE x in lborel. integrable lborel* $(\lambda y.\ ?f\ (x, y))$
by (*intro AE-I2*) (*auto simp*: *indicator-times set-integrable-def split*: *split-indicator*)
qed
also have ... $= \mathit{LBINT}\ u{=}0..\infty.\ (\mathit{LBINT}\ x{=}0..t.\ \exp\ (-(u * x)) * \sin x)$
using ‹$0 \le t$›
by (*auto simp*: *interval-lebesgue-integral-def set-lebesgue-integral-def zero-ereal-def ac-simps*
split: *split-indicator intro*!: *Bochner-Integration.integral-cong*)
also have $\ldots = \mathit{LBINT}\ u{=}0..\infty.\ 1\ /\ (1 + u^2) - 1\ /\ (1 + u^2) * (\exp\ (-\ (u * t)) * (u * \sin t + \cos t))$
by (*auto simp*: *divide-simps LBINT-I0c-exp-mscale-sin intro*!: *interval-integral-cong*)
also have ... $= \mathit{pi}\ /\ 2 - (\mathit{LBINT}\ u{=}0..\infty.\ \exp\ (-\ (u * t)) * (u * \sin t + \cos t)\ /\ (1 + u\hat{}2))$
using *Si-at-top-integrable*[*OF* ‹$0 \le t$›] **by** (*simp add*: *integrable-I0i-1-div-plus-square LBINT-I0i-1-div-plus-square*)
finally show $\mathit{pi}\ /\ 2 - (\mathit{LBINT}\ u{=}0..\infty.\ \exp\ (-(u * t)) * (u * \sin t + \cos t)\ /\ (1 + u\hat{}2)) = \mathit{Si}\ t$ **..**
qed
show *?thesis*
by (*rule Lim-transform-eventually* [*OF* - †])
(*auto intro*!: *tendsto-eq-intros Si-at-top-LBINT*)
qed

17.1 The final theorems: boundedness and scalability

lemma *bounded-Si*: $\exists B.\ \forall T.\ |\mathit{Si}\ T| \le B$
proof −
have $*$: $0 \le y \Longrightarrow \mathit{dist}\ x\ y < z \Longrightarrow \mathit{abs}\ x \le y + z$ **for** $x\ y\ z :: \mathit{real}$
by (*simp add*: *dist-real-def*)

have *eventually* (λ*T*. *dist* (*Si T*) (*pi / 2*) < *1*) *at-top*
using *Si-at-top* **by** (*elim tendstoD*) *simp*
then have *eventually* (λ*T*. *0* ≤ *T* ∧ |*Si T*| ≤ *pi / 2 + 1*) *at-top*
using *eventually-ge-at-top*[*of 0*] **by** *eventually-elim* (*insert* *[*of pi/2 Si - 1*], *auto*)
then have ∃ *T*. *0* ≤ *T* ∧ (∀ *t* ≥ *T*. |*Si t*| ≤ *pi / 2 + 1*)
by (*auto simp add*: *eventually-at-top-linorder*)
then obtain *T* **where** *T*: *0* ≤ *T* ⋀*t*. *t* ≥ *T* ⟹ |*Si t*| ≤ *pi / 2 + 1*
by *auto*
moreover have *t* ≤ − *T* ⟹ |*Si t*| ≤ *pi / 2 + 1* **for** *t*
using *T*(*1*) *T*(*2*)[*of* −*t*] *Si-neg*[*of* − *t*] **by** *simp*
moreover have ∃ *M*. ∀ *t*. −*T* ≤ *t* ∧ *t* ≤ *T* ⟶ |*Si t*| ≤ *M*
by (*rule isCont-bounded*) (*auto intro*!: *isCont-Si continuous-intros* ‹*0* ≤ *T*›)
then obtain *M* **where** *M*: ⋀*t*. −*T* ≤ *t* ∧ *t* ≤ *T* ⟹ |*Si t*| ≤ *M*
by *auto*
ultimately show *?thesis*
by (*intro exI*[*of - max M* (*pi/2 + 1*)]) (*meson le-max-iff-disj linorder-not-le order-le-less*)
qed

lemma *LBINT-I0c-sin-scale-divide*:
assumes *T* ≥ *0*
shows *LBINT t*=*0..T*. *sin* (*t* ∗ ϑ) / *t* = *sgn* ϑ ∗ *Si* (*T* ∗ |ϑ|)
proof −
{ assume *0* < ϑ
have (*LBINT t*=*ereal 0..T*. *sin* (*t* ∗ ϑ) / *t*) = (*LBINT t*=*ereal 0..T*. ϑ ∗$_R$ *sinc* (*t* ∗ ϑ))
by (*rule interval-integral-discrete-difference*[*of* {*0*}]) *auto*
also have ... = (*LBINT t*=*ereal* (*0* ∗ ϑ)*..T* ∗ ϑ. *sinc t*)
apply (*rule interval-integral-substitution-finite* [*OF assms*])
apply (*subst mult.commute*, *rule DERIV-subset*)
by (*auto intro*!: *derivative-intros continuous-at-imp-continuous-on isCont-sinc*)
also have ... = (*LBINT t*=*ereal* (*0* ∗ ϑ)*..T* ∗ ϑ. *sin t* / *t*)
by (*rule interval-integral-discrete-difference*[*of* {*0*}]) *auto*
finally have (*LBINT t*=*ereal 0..T*. *sin* (*t* ∗ ϑ) / *t*) = (*LBINT t*=*ereal 0..T* ∗ ϑ. *sin t* / *t*)
by *simp*
hence *LBINT x*. *indicator* {*0*<..<*T*} *x* ∗ *sin* (*x* ∗ ϑ) / *x* =
LBINT x. *indicator* {*0*<..<*T* ∗ ϑ} *x* ∗ *sin x* / *x*
using *assms* ‹*0* < ϑ› **unfolding** *interval-lebesgue-integral-def einterval-eq zero-ereal-def*
by (*auto simp*: *ac-simps set-lebesgue-integral-def*)
} note *aux1* = *this*
{ assume *0* > ϑ
have [*simp*]: *integrable lborel* (λ*x*. *sin* (*x* ∗ ϑ) ∗ *indicator* {*0*<..<*T*} *x* / *x*)
using *integrable-sinc*′ [*of T* ϑ] *assms*
by (*simp add*: *interval-lebesgue-integrable-def set-integrable-def ac-simps*)
have (*LBINT t*=*ereal 0..T*. *sin* (*t* ∗ −ϑ) / *t*) = (*LBINT t*=*ereal 0..T*. −ϑ ∗$_R$ *sinc* (*t* ∗ −ϑ))

by (*rule interval-integral-discrete-difference*[*of* {*0*}]) *auto*
also have ... = (*LBINT t*=*ereal* (*0* ∗ −*ϑ*)..*T* ∗ −*ϑ*. *sinc t*)
apply (*rule interval-integral-substitution-finite* [*OF assms*])
apply (*subst mult.commute*, *rule DERIV-subset*)
by (*auto intro*!: *derivative-intros continuous-at-imp-continuous-on isCont-sinc*)
also have ... = (*LBINT t*=*ereal* (*0* ∗ −*ϑ*)..*T* ∗ −*ϑ*. *sin t* / *t*)
by (*rule interval-integral-discrete-difference*[*of* {*0*}]) *auto*
finally have (*LBINT t*=*ereal 0*..*T*. *sin* (*t* ∗ −*ϑ*) / *t*) = (*LBINT t*=*ereal 0*..*T* ∗ −*ϑ*. *sin t* / *t*)
by *simp*
hence *LBINT x*. *indicator* {*0*<..<*T*} *x* ∗ *sin* (*x* ∗ *ϑ*) / *x* =
− (*LBINT x*. *indicator* {*0*<..<− (*T* ∗ *ϑ*)} *x* ∗ *sin x* / *x*)
using *assms* ‹*0* > *ϑ*› **unfolding** *interval-lebesgue-integral-def einterval-eq zero-ereal-def*
by (*auto simp add*: *field-simps mult-le-0-iff set-lebesgue-integral-def split*: *if-split-asm*)
} **note** *aux2* = *this*
show *?thesis*
using *assms* **unfolding** *Si-def interval-lebesgue-integral-def set-lebesgue-integral-def sgn-real-def einterval-eq zero-ereal-def*
by (*auto simp*: *aux1 aux2*)
qed

end

18 The Levy inversion theorem, and the Levy continuity theorem.

theory *Levy*
imports *Characteristic-Functions Helly-Selection Sinc-Integral*
begin

18.1 The Levy inversion theorem

lemma *Levy-Inversion-aux1*:
fixes *a b* :: *real*
assumes *a* ≤ *b*
shows ((λ*t*. (*iexp* (−(*t* ∗ *a*)) − *iexp* (−(*t* ∗ *b*))) / (i ∗ *t*)) ⟶ *b* − *a*) (*at 0*)
(**is** (*?F* ⟶ -) (*at* -))
proof −
have *1*: *cmod* (*?F t* − (*b* − *a*)) ≤ *a*^*2* / *2* ∗ *abs t* + *b*^*2* / *2* ∗ *abs t* **if** *t* ≠ *0* **for** *t*
proof −
have *cmod* (*?F t* − (*b* − *a*)) = *cmod* (
(*iexp* (−(*t* ∗ *a*)) − (*1* + i ∗ −(*t* ∗ *a*))) / (i ∗ *t*) −
(*iexp* (−(*t* ∗ *b*)) − (*1* + i ∗ −(*t* ∗ *b*))) / (i ∗ *t*))
(**is** - = *cmod* (*?one* / (i ∗ *t*) − *?two* / (i ∗ *t*)))
using ‹*t* ≠ *0*› **by** (*intro arg-cong*[**where** *f*=*norm*]) (*simp add*: *field-simps*)

```
    also have ... ≤ cmod (?one / (i * t)) + cmod (?two / (i * t))
      by (rule norm-triangle-ineq4)
    also have cmod (?one / (i * t)) = cmod ?one / abs t
      by (simp add: norm-divide norm-mult)
    also have cmod (?two / (i * t)) = cmod ?two / abs t
      by (simp add: norm-divide norm-mult)
    also have cmod ?one / abs t + cmod ?two / abs t ≤
        ((- (a * t))^2 / 2) / abs t + ((- (b * t))^2 / 2) / abs t
      apply (rule add-mono)
      apply (rule divide-right-mono)
     using iexp-approx1 [of -(t * a) 1] apply (simp add: field-simps eval-nat-numeral)
      apply force
      apply (rule divide-right-mono)
     using iexp-approx1 [of -(t * b) 1] apply (simp add: field-simps eval-nat-numeral)
      by force
    also have ... = a^2 / 2 * abs t + b^2 / 2 * abs t
     using ‹t ≠ 0› apply (case-tac t ≥ 0, simp add: field-simps power2-eq-square)
        using ‹t ≠ 0› by (subst (1 2) abs-of-neg, auto simp add: field-simps
power2-eq-square)
    finally show cmod (?F t - (b - a)) ≤ a^2 / 2 * abs t + b^2 / 2 * abs t .
  qed
  show ?thesis
    apply (rule LIM-zero-cancel)
    apply (rule tendsto-norm-zero-cancel)
    apply (rule real-LIM-sandwich-zero [OF - - 1])
    apply (auto intro!: tendsto-eq-intros)
    done
qed

lemma Levy-Inversion-aux2:
  fixes a b t :: real
  assumes a ≤ b and t ≠ 0
  shows cmod ((iexp (t * b) - iexp (t * a)) / (i * t)) ≤ b - a (is ?F ≤ -)
proof -
  have ?F = cmod (iexp (t * a) * (iexp (t * (b - a)) - 1) / (i * t))
     using ‹t ≠ 0› by (intro arg-cong[where f=norm]) (simp add: field-simps
exp-diff exp-minus)
  also have ... = cmod (iexp (t * (b - a)) - 1) / abs t
    unfolding norm-divide norm-mult norm-exp-i-times using ‹t ≠ 0›
    by (simp add: complex-eq-iff norm-mult)
  also have ... ≤ abs (t * (b - a)) / abs t
    using iexp-approx1 [of t * (b - a) 0]
    by (intro divide-right-mono) (auto simp add: field-simps eval-nat-numeral)
  also have ... = b - a
    using assms by (auto simp add: abs-mult)
  finally show ?thesis .
qed
```

theorem (**in** *real-distribution*) *Levy-Inversion*:
fixes $a\ b :: real$
assumes $a \leq b$
defines $\mu \equiv measure\ M$ **and** $\varphi \equiv char\ M$
assumes $\mu\ \{a\} = 0$ **and** $\mu\ \{b\} = 0$
shows $(\lambda T.\ 1\ /\ (2 * pi) * (CLBINT\ t{=}{-}T..T.\ (iexp\ (-(t * a)) - iexp\ (-(t * b))) / (\mathrm{i} * t) * \varphi\ t))$
$\longrightarrow \mu\ \{a<..b\}$
(**is** $(\lambda T.\ 1\ /\ (2 * pi) * (CLBINT\ t{=}{-}T..T.\ ?F\ t * \varphi\ t)) \longrightarrow of\text{-}real\ (\mu\ \{a<..b\}))$
proof −
interpret *P*: *pair-sigma-finite lborel M* **..**
from *bounded-Si* **obtain** *B* **where** *Bprop*: $\bigwedge T.\ abs\ (Si\ T) \leq B$ **by** *auto*
from *Bprop* [*of 0*] **have** [*simp*]: $B \geq 0$ **by** *auto*
let $?f = \lambda t\ x :: real.\ (iexp\ (t * (x - a)) - iexp(t * (x - b))) / (\mathrm{i} * t)$
{ fix $T :: real$
assume $T \geq 0$
let $?f' = \lambda(t,\ x).\ indicator\ \{-T<..<T\}\ t *_R\ ?f\ t\ x$
{ fix x
have *1*: *complex-interval-lebesgue-integrable lborel u v* $(\lambda t.\ ?f\ t\ x)$ **for** $u\ v ::$ *real*
using *Levy-Inversion-aux2*[*of x − b x − a*]
apply (*simp add*: *interval-lebesgue-integrable-def set-integrable-def del*: *times-divide-eq-left*)
apply (*intro integrableI-bounded-set-indicator*[**where** $B{=}b - a$] *conjI impI*)
apply (*auto intro*!: *AE-I* [*of - - {0}*] *simp*: *assms*)
done
have $(CLBINT\ t.\ ?f'\ (t,\ x)) = (CLBINT\ t{=}{-}T..T.\ ?f\ t\ x)$
using ‹$T \geq 0$› **by** (*simp add*: *interval-lebesgue-integral-def set-lebesgue-integral-def*)
also have $\ldots = (CLBINT\ t{=}{-}T..(0 :: real).\ ?f\ t\ x) + (CLBINT\ t{=}(0 :: real)..T.\ ?f\ t\ x)$
(**is** - = - + *?t*)
using *1* **by** (*intro interval-integral-sum*[*symmetric*]) (*simp add*: *min-absorb1 max-absorb2* ‹$T \geq 0$›)
also have $(CLBINT\ t{=}{-}T..(0 :: real).\ ?f\ t\ x) = (CLBINT\ t{=}(0{::}real)..T.\ ?f\ (-t)\ x)$
by (*subst interval-integral-reflect*) *auto*
also have $\ldots + ?t = (CLBINT\ t{=}(0{::}real)..T.\ ?f\ (-t)\ x + ?f\ t\ x)$
using *1*
by (*intro interval-lebesgue-integral-add*(*2*) [*symmetric*] *interval-integrable-mirror*[*THEN iffD2*]) *simp-all*
also have $\ldots = (CLBINT\ t{=}(0{::}real)..T.\ ((iexp(t * (x - a)) - iexp\ (-(t * (x - a)))) - (iexp(t * (x - b)) - iexp\ (-(t * (x - b))))) / (\mathrm{i} * t))$
using ‹$T \geq 0$› **by** (*intro interval-integral-cong*) (*auto simp add*: *field-split-simps*)
also have $\ldots = (CLBINT\ t{=}(0{::}real)..T.\ complex\text{-}of\text{-}real($
$2 * (sin\ (t * (x - a)) / t) - 2 * (sin\ (t * (x - b)) / t)))$
using ‹$T \geq 0$›
apply (*intro interval-integral-cong*)

apply (*simp add*: *field-simps cis.ctr Im-divide Re-divide Im-exp Re-exp complex-eq-iff*)
unfolding *minus-diff-eq*[*symmetric*, *of* $y * x$ $y * a$ **for** y a] *sin-minus cos-minus*
apply (*simp add*: *field-simps power2-eq-square*)
done
also have ... = *complex-of-real* (*LBINT* t=(0::*real*)..T. $2 * (sin\ (t * (x - a))\ /\ t) - 2 * (sin\ (t * (x - b))\ /\ t)$)
by (*rule interval-lebesgue-integral-of-real*)
also have ... = *complex-of-real* ($2 * (sgn\ (x - a) * Si\ (T * abs\ (x - a)) - sgn\ (x - b) * Si\ (T * abs\ (x - b))))$
apply (*subst interval-lebesgue-integral-diff*)
apply (*rule interval-lebesgue-integrable-mult-right*, *rule integrable-sinc'*)+
apply (*subst interval-lebesgue-integral-mult-right*)+
apply (*simp add*: *zero-ereal-def*[*symmetric*] *LBINT-I0c-sin-scale-divide*[*OF* ‹$T \geq 0$›])
done
finally have (*CLBINT t*. *?f'* (*t*, *x*)) = $2 * (sgn\ (x - a) * Si\ (T * abs\ (x - a)) - sgn\ (x - b) * Si\ (T * abs\ (x - b)))$ **.**
} note *main-eq* = *this*
have (*CLBINT* t=$-T$..T. *?F* $t * \varphi\ t$) =
(*CLBINT t*. (*CLINT x* | *M*. *?F* $t * iexp\ (t * x) * indicator\ \{-T<..<T\}\ t$))
using ‹$T \geq 0$› **unfolding** φ-*def char-def interval-lebesgue-integral-def set-lebesgue-integral-def*
by (*auto split*: *split-indicator intro*!: *Bochner-Integration.integral-cong*)
also have ... = (*CLBINT t*. (*CLINT x* | *M*. *?f'* (*t*, *x*)))
by (*auto intro*!: *Bochner-Integration.integral-cong simp*: *field-simps exp-diff exp-minus split*: *split-indicator*)
also have ... = (*CLINT x* | *M*. (*CLBINT t*. *?f'* (*t*, *x*)))
proof (*intro P.Fubini-integral* [*symmetric*] *integrableI-bounded-set* [**where** B=$b - a$])
show *emeasure* (*lborel* $\bigotimes_M$ *M*) ($\{-\ T<..<T\} \times$ *space M*) $< \infty$
using ‹$T \geq 0$›
by (*subst emeasure-pair-measure-Times*)
(*auto simp*: *ennreal-mult-less-top not-less top-unique*)
show *AE* x∈$\{-\ T<..<T\} \times$ *space M in lborel* $\bigotimes_M$ *M*. *cmod* (*case x of* (*t*, *x*) ⇒ *?f'* (*t*, *x*)) $\leq b - a$
using *Levy-Inversion-aux2*[*of* $x - b$ $x - a$ **for** x] ‹$a \leq b$›
by (*intro AE-I* [*of* - - $\{0\} \times$ *UNIV*]) (*force simp*: *emeasure-pair-measure-Times*)+
qed (*auto split*: *split-indicator split-indicator-asm*)
also have ... = (*CLINT x* | *M*. (*complex-of-real* ($2 * (sgn\ (x - a) * Si\ (T * abs\ (x - a)) - sgn\ (x - b) * Si\ (T * abs\ (x - b))))))$
using *main-eq* **by** (*intro Bochner-Integration.integral-cong*, *auto*)
also have ... = *complex-of-real* (*LINT x* | *M*. ($2 * (sgn\ (x - a) * Si\ (T * abs\ (x - a)) - sgn\ (x - b) * Si\ (T * abs\ (x - b)))))$
by (*rule integral-complex-of-real*)
finally have (*CLBINT* t=$-T$..T. *?F* $t * \varphi\ t$) =
complex-of-real (*LINT x* | *M*. ($2 * (sgn\ (x - a) * Si\ (T * abs\ (x - a)) - sgn\ (x - b) * Si\ (T * abs\ (x - b)))))$ **.**

} note *main-eq2 = this*

have (λT :: *nat*. *LINT x | M*. (*2* ∗ (*sgn* (*x* − *a*) ∗
Si (*T* ∗ *abs* (*x* − *a*)) − *sgn* (*x* − *b*) ∗ *Si* (*T* ∗ *abs* (*x* − *b*))))) ⟶
(*LINT x | M*. *2* ∗ *pi* ∗ *indicator* {*a*<..*b*} *x*)
proof (*rule integral-dominated-convergence* [**where** *w*=λ*x*. *4* ∗ *B*])
show *integrable M* (λ*x*. *4* ∗ *B*)
by (*rule integrable-const-bound* [*of* - *4* ∗ *B*]) *auto*
next
let *?S* = λ*n*::*nat*. λ*x*. *sgn* (*x* − *a*) ∗ *Si* (*n* ∗ |*x* − *a*|) − *sgn* (*x* − *b*) ∗ *Si* (*n* ∗ |*x* − *b*|)
{ fix *n x*
have *norm* (*?S n x*) ≤ *norm* (*sgn* (*x* − *a*) ∗ *Si* (*n* ∗ |*x* − *a*|)) + *norm* (*sgn* (*x* − *b*) ∗ *Si* (*n* ∗ |*x* − *b*|))
by (*rule norm-triangle-ineq4*)
also have ... ≤ *B* + *B*
using *Bprop* **by** (*intro add-mono*) (*auto simp*: *abs-mult abs-sgn-eq*)
finally have *norm* (*2* ∗ *?S n x*) ≤ *4* ∗ *B*
by *simp* **}**
then show ⋀*n*. *AE x in M*. *norm* (*2* ∗ *?S n x*) ≤ *4* ∗ *B*
by *auto*
have *AE x in M*. *x* ≠ *a AE x in M*. *x* ≠ *b*
using *prob-eq-0*[*of* {*a*}] *prob-eq-0*[*of* {*b*}] ‹μ {*a*} = *0*› ‹μ {*b*} = *0*› **by** (*auto simp*: μ-*def*)
then show *AE x in M*. (λ*n*. *2* ∗ *?S n x*) ⟶ *2* ∗ *pi* ∗ *indicator* {*a*<..*b*} *x*
proof *eventually-elim*
fix *x* **assume** *x*: *x* ≠ *a x* ≠ *b*
then have (λ*n*. *2* ∗ (*sgn* (*x* − *a*) ∗ *Si* (|*x* − *a*| ∗ *n*) − *sgn* (*x* − *b*) ∗ *Si* (|*x* − *b*| ∗ *n*)))
⟶ *2* ∗ (*sgn* (*x* − *a*) ∗ (*pi* / *2*) − *sgn* (*x* − *b*) ∗ (*pi* / *2*))
by (*intro tendsto-intros filterlim-compose*[*OF Si-at-top*]
filterlim-tendsto-pos-mult-at-top[*OF tendsto-const*] *filterlim-real-sequentially*)
auto
also have (λ*n*. *2* ∗ (*sgn* (*x* − *a*) ∗ *Si* (|*x* − *a*| ∗ *n*) − *sgn* (*x* − *b*) ∗ *Si* (|*x* − *b*| ∗ *n*))) = (λ*n*. *2* ∗ *?S n x*)
by (*auto simp*: *ac-simps*)
also have *2* ∗ (*sgn* (*x* − *a*) ∗ (*pi* / *2*) − *sgn* (*x* − *b*) ∗ (*pi* / *2*)) = *2* ∗ *pi* ∗ *indicator* {*a*<..*b*} *x*
using *x* ‹*a* ≤ *b*› **by** (*auto split*: *split-indicator*)
finally show (λ*n*. *2* ∗ *?S n x*) ⟶ *2* ∗ *pi* ∗ *indicator* {*a*<..*b*} *x* **.**
qed
qed *simp-all*
also have (*LINT x | M*. *2* ∗ *pi* ∗ *indicator* {*a*<..*b*} *x*) = *2* ∗ *pi* ∗ μ {*a*<..*b*}
by (*simp add*: μ-*def*)
finally have (λT. *LINT x | M*. (*2* ∗ (*sgn* (*x* − *a*) ∗
Si (*T* ∗ *abs* (*x* − *a*)) − *sgn* (*x* − *b*) ∗ *Si* (*T* ∗ *abs* (*x* − *b*))))) ⟶
2 ∗ *pi* ∗ μ {*a*<..*b*} **.**
with *main-eq2* **show** *?thesis*
by (*auto intro*!: *tendsto-eq-intros*)

qed

theorem *Levy-uniqueness*:
fixes *M1 M2* :: *real measure*
assumes *real-distribution M1 real-distribution M2* **and**
char M1 = *char M2*
shows *M1* = *M2*
proof −
interpret *M1*: *real-distribution M1* **by** (*rule assms*)
interpret *M2*: *real-distribution M2* **by** (*rule assms*)
have *countable* ({*x*. *measure M1* {*x*} ≠ *0*} ∪ {*x*. *measure M2* {*x*} ≠ *0*})
by (*intro countable-Un M2.countable-support M1.countable-support*)
then have *count*: *countable* {*x*. *measure M1* {*x*} ≠ *0* ∨ *measure M2* {*x*} ≠ *0*}
by (*simp add*: *Un-def*)

have *cdf M1* = *cdf M2*
proof (*rule ext*)
fix *x*
let *?D* = λ*x*. |*cdf M1 x* − *cdf M2 x*|

{ fix ε :: *real*
assume ε > *0*
have (*?D* ⟶ *0*) *at-bot*
using *M1.cdf-lim-at-bot M2.cdf-lim-at-bot* **by** (*intro tendsto-eq-intros*) *auto*
then have *eventually* (λ*y*. *?D y* < ε / *2* ∧ *y* ≤ *x*) *at-bot*
using ‹ε > *0*› **by** (*simp only*: *tendsto-iff dist-real-def diff-0-right eventually-conj eventually-le-at-bot abs-idempotent*)
then obtain *M* **where** ⋀*y*. *y* ≤ *M* ⟹ *?D y* < ε / *2 M* ≤ *x*
unfolding *eventually-at-bot-linorder* **by** *auto*
with *open-minus-countable*[*OF count*, *of* {..< *M*}] **obtain** *a* **where**
measure M1 {*a*} = *0 measure M2* {*a*} = *0 a* < *M a* ≤ *x* **and** *1*: *?D a* < ε / *2*
by *auto*

have (*?D* ⟶ *?D x*) (*at-right x*)
using *M1.cdf-is-right-cont* [*of x*] *M2.cdf-is-right-cont* [*of x*]
by (*intro tendsto-intros*) (*auto simp add*: *continuous-within*)
then have *eventually* (λ*y*. |*?D y* − *?D x*| < ε / *2*) (*at-right x*)
using ‹ε > *0*› **by** (*simp only*: *tendsto-iff dist-real-def eventually-conj eventually-at-right-less*)
then obtain *N* **where** *N* > *x* ⋀*y*. *x* < *y* ⟹ *y* < *N* ⟹ |*?D y* − *?D x*| < ε / *2*
by (*auto simp add*: *eventually-at-right*[*OF less-add-one*])
with *open-minus-countable*[*OF count*, *of* {*x* <..< *N*}] **obtain** *b* **where** *x* < *b b* < *N*
measure M1 {*b*} = *0 measure M2* {*b*} = *0* **and** *2*: |*?D b* − *?D x*| < ε / *2*
by (*auto simp*: *abs-minus-commute*)
from ‹*a* ≤ *x*› ‹*x* < *b*› **have** *a* < *b a* ≤ *b* **by** *auto*

from ‹*char M1 = char M2*›
M1.Levy-Inversion [*OF* ‹$a \leq b$› ‹*measure M1* $\{a\} = 0$› ‹*measure M1* $\{b\} = 0$›]
M2.Levy-Inversion [*OF* ‹$a \leq b$› ‹*measure M2* $\{a\} = 0$› ‹*measure M2* $\{b\} = 0$›]
have *complex-of-real* (*measure M1* $\{a<..b\}$) = *complex-of-real* (*measure M2* $\{a<..b\}$)
by (*intro LIMSEQ-unique*) *auto*
then have *?D a = ?D b*
unfolding *of-real-eq-iff M1.cdf-diff-eq* [*OF* ‹$a < b$›, *symmetric*] *M2.cdf-diff-eq* [*OF* ‹$a < b$›, *symmetric*] **by** *simp*
then have *?D x* = $|$(*?D b* − *?D x*) − *?D a*$|$
by *simp*
also have ... ≤ $|$*?D b* − *?D x*$|$ + $|$*?D a*$|$
by (*rule abs-triangle-ineq4*)
also have ... ≤ ε / 2 + ε / 2
using *1 2* **by** (*intro add-mono*) *auto*
finally have *?D x* ≤ ε **by** *simp* **}**
then show *cdf M1 x = cdf M2 x*
by (*metis abs-le-zero-iff dense-ge eq-iff-diff-eq-0*)
qed
thus *?thesis*
by (*rule cdf-unique* [*OF* ‹*real-distribution M1*› ‹*real-distribution M2*›])
qed

18.2 The Levy continuity theorem

theorem *levy-continuity1*:
fixes *M* :: *nat* ⇒ *real measure* **and** *M′* :: *real measure*
assumes ⋀*n*. *real-distribution* (*M n*) *real-distribution M′ weak-conv-m M M′*
shows (λ*n*. *char* (*M n*) *t*) ⟶ *char M′ t*
unfolding *char-def* **using** *assms* **by** (*rule weak-conv-imp-integral-bdd-continuous-conv*) *auto*

theorem *levy-continuity*:
fixes *M* :: *nat* ⇒ *real measure* **and** *M′* :: *real measure*
assumes *real-distr-M* : ⋀*n*. *real-distribution* (*M n*)
and *real-distr-M′*: *real-distribution M′*
and *char-conv*: ⋀*t*. (λ*n*. *char* (*M n*) *t*) ⟶ *char M′ t*
shows *weak-conv-m M M′*
proof −
interpret *Mn*: *real-distribution M n* **for** *n* **by** *fact*
interpret *M′*: *real-distribution M′* **by** *fact*

have ∗: ⋀*u x*. *u* > *0* ⟹ *x* ≠ *0* ⟹ (*CLBINT t*:{−*u*..*u*}. *1* − *iexp* (*t* ∗ *x*)) = *2* ∗ (*u* − *sin* (*u* ∗ *x*) / *x*)
proof −
fix *u* :: *real* **and** *x* :: *real*
assume *u* > *0* **and** *x* ≠ *0*

hence (*CLBINT t:{−u..u}. 1 − iexp (t ∗ x)*) = (*CLBINT t=−u..u. 1 − iexp (t ∗ x)*)
by (*subst interval-integral-Icc*, *auto*)
also have ... = (*CLBINT t=−u..0. 1 − iexp (t ∗ x)*) + (*CLBINT t=0..u. 1 − iexp (t ∗ x)*)
using ‹*u > 0*›
apply (*subst interval-integral-sum*)
apply (*simp add*: *min-absorb1 min-absorb2 max-absorb1 max-absorb2*)
apply (*rule interval-integrable-isCont*)
apply *auto*
done
also have ... = (*CLBINT t=ereal 0..u. 1 − iexp (t ∗ −x)*) + (*CLBINT t=ereal 0..u. 1 − iexp (t ∗ x)*)
apply (*subgoal-tac 0 = ereal 0*, *erule ssubst*)
by (*subst interval-integral-reflect*, *auto*)
also have ... = (*LBINT t=ereal 0..u. 2 − 2 ∗ cos (t ∗ x)*)
apply (*subst interval-lebesgue-integral-add* (*2*) [*symmetric*])
apply ((*rule interval-integrable-isCont*, *auto*)+) [*2*]
unfolding *exp-Euler cos-of-real*
apply (*simp add*: *of-real-mult interval-lebesgue-integral-of-real*[*symmetric*])
done
also have ... = *2 ∗ u − 2 ∗ sin (u ∗ x) / x*
by (*subst interval-lebesgue-integral-diff*)
(*auto intro*!: *interval-integrable-isCont*
simp: *interval-lebesgue-integral-of-real integral-cos* [*OF* ‹*x ≠ 0*›] *mult.commute*[*of - x*])
finally show (*CLBINT t:{−u..u}. 1 − iexp (t ∗ x)*) = *2 ∗* (*u − sin (u ∗ x) / x*)
by (*simp add*: *field-simps*)
qed
have *main-bound*: ⋀*u n. u > 0 ⟹ Re* (*CLBINT t:{−u..u}. 1 − char* (*M n*) *t*) ≥
u ∗ measure (*M n*) {*x. abs x ≥ 2 / u*}
proof −
fix *u* :: *real* **and** *n*
assume *u > 0*
interpret *P*: *pair-sigma-finite M n lborel* **..**

have *Mn1* [*simp*]: *measure* (*M n*) *UNIV = 1* **by** (*metis Mn.prob-space Mn.space-eq-univ*)

have *Mn2* [*simp*]: ⋀*x. complex-integrable* (*M n*) (λ*t. exp* (i ∗ *complex-of-real* (*x ∗ t*)))
by (*rule Mn.integrable-const-bound* [**where** *B = 1*], *auto*)
have *Mn3*: *set-integrable* (*M n* ⨂$_M$ *lborel*) (*UNIV* × {− *u..u*}) (λ*a. 1 − exp* (i ∗ *complex-of-real* (*snd a* ∗ *fst a*)))
using ‹*0 < u*›
unfolding *set-integrable-def*
by (*intro integrableI-bounded-set-indicator* [**where** *B=2*])
(*auto simp*: *lborel.emeasure-pair-measure-Times ennreal-mult-less-top not-less*

top-unique
split: *split-indicator*
intro!: *order-trans* [*OF norm-triangle-ineq4*])
have (*CLBINT t*:{−*u*..*u*}. *1* − *char* (*M n*) *t*) =
(*CLBINT t*:{−*u*..*u*}. (*CLINT x* | *M n*. *1* − *iexp* (*t* ∗ *x*)))
unfolding *char-def* **by** (*rule set-lebesgue-integral-cong*, *auto simp del*: *of-real-mult*)
also have ... = (*CLBINT t*. (*CLINT x* | *M n*. *indicator* {−*u*..*u*} *t* ∗$_R$ (*1* − *iexp* (*t* ∗ *x*))))
unfolding *set-lebesgue-integral-def*
by (*rule Bochner-Integration.integral-cong*) (*auto split*: *split-indicator*)
also have ... = (*CLINT x* | *M n*. (*CLBINT t*:{−*u*..*u*}. *1* − *iexp* (*t* ∗ *x*)))
using *Mn3* **by** (*subst P.Fubini-integral*) (*auto simp*: *indicator-times split-beta′ set-integrable-def set-lebesgue-integral-def*)
also have ... = (*CLINT x* | *M n*. (*if x* = *0 then 0 else 2* ∗ (*u* − *sin* (*u* ∗ *x*) / *x*)))
using ‹*u* > *0*› **by** (*intro Bochner-Integration.integral-cong*, *auto simp add*: ∗ *simp del*: *of-real-mult*)
also have ... = (*LINT x* | *M n*. (*if x* = *0 then 0 else 2* ∗ (*u* − *sin* (*u* ∗ *x*) / *x*)))
by (*rule integral-complex-of-real*)
finally have *Re* (*CLBINT t*:{−*u*..*u*}. *1* − *char* (*M n*) *t*) =
(*LINT x* | *M n*. (*if x* = *0 then 0 else 2* ∗ (*u* − *sin* (*u* ∗ *x*) / *x*))) **by** *simp*
also have ... ≥ (*LINT x* : {*x*. *abs x* ≥ *2* / *u*} | *M n*. *u*)
proof −
have *complex-integrable* (*M n*) (λ*x*. *CLBINT t*:{−*u*..*u*}. *1* − *iexp* (*snd* (*x*, *t*) ∗ *fst* (*x*, *t*)))
using *Mn3* **unfolding** *set-integrable-def set-lebesgue-integral-def*
by (*intro P.integrable-fst*) (*simp add*: *indicator-times split-beta′*)
hence *complex-integrable* (*M n*) (λ*x*. *if x* = *0 then 0 else 2* ∗ (*u* − *sin* (*u* ∗ *x*) / *x*))
using ‹*u* > *0*›
unfolding *set-integrable-def*
by (*subst integrable-cong*) (*auto simp add*: ∗ *simp del*: *of-real-mult*)
hence ∗∗: *integrable* (*M n*) (λ*x*. *if x* = *0 then 0 else 2* ∗ (*u* − *sin* (*u* ∗ *x*) / *x*))
unfolding *complex-of-real-integrable-eq* **.**
have *2* ∗ *sin x* ≤ *x* **if** *2* ≤ *x* **for** *x* :: *real*
by (*rule order-trans*[*OF* - ‹*2* ≤ *x*›]) *auto*
moreover have *x* ≤ *2* ∗ *sin x* **if** *x* ≤ − *2* **for** *x* :: *real*
by (*rule order-trans*[*OF* ‹*x* ≤ − *2*›]) *auto*
moreover have *x* < *0* ⟹ *x* ≤ *sin x* **for** *x* :: *real*
using *sin-x-le-x*[*of* −*x*] **by** *simp*
ultimately show *?thesis*
using ‹*u* > *0*› **unfolding** *set-lebesgue-integral-def*
by (*intro integral-mono* [*OF* - ∗∗])
(*auto simp*: *divide-simps sin-x-le-x mult.commute*[*of u*] *mult-neg-pos top-unique less-top*[*symmetric*]
split: *split-indicator*)
qed

```
  also (xtrans) have (LINT x : {x. abs x ≥ 2 / u} | M n. u) = u * measure (M
n) {x. abs x ≥ 2 / u}
    unfolding set-lebesgue-integral-def
    by (simp add: Mn.emeasure-eq-measure)
  finally show Re (CLBINT t:{−u..u}. 1 − char (M n) t) ≥ u * measure (M
n) {x. abs x ≥ 2 / u} .
 qed

 have tight-aux: ⋀ε. ε > 0 ⟹ ∃ a b. a < b ∧ (∀ n. 1 − ε < measure (M n)
{a<..b})
 proof −
  fix ε :: real
  assume ε > 0
  note M'.isCont-char [of 0]
  hence ∃ d>0. ∀ t. abs t < d ⟶ cmod (char M' t − 1) < ε / 4
   apply (subst (asm) continuous-at-eps-delta)
   apply (drule-tac x = ε / 4 in spec)
   using ‹ε > 0› by (auto simp add: dist-real-def dist-complex-def M'.char-zero)
  then obtain d where d > 0 ∧ (∀ t. (abs t < d ⟶ cmod (char M' t − 1) <
ε / 4)) ..
  hence d0: d > 0 and d1: ⋀t. abs t < d ⟹ cmod (char M' t − 1) < ε / 4
by auto
  have 1: ⋀x. cmod (1 − char M' x) ≤ 2
  by (rule order-trans [OF norm-triangle-ineq4], auto simp add: M'.cmod-char-le-1)
  then have 2: ⋀u v. complex-set-integrable lborel {u..v} (λx. 1 − char M' x)
   unfolding set-integrable-def
   by (intro integrableI-bounded-set-indicator[where B=2]) (auto simp: emea-
sure-lborel-Icc-eq)
  have 3: ⋀u v. integrable lborel (λx. indicat-real {u..v} x *R cmod (1 − char
M' x))
  by (intro borel-integrable-compact[OF compact-Icc] continuous-at-imp-continuous-on
        continuous-intros ballI M'.isCont-char continuous-intros)
 have cmod (CLBINT t:{−d/2..d/2}. 1 − char M' t) ≤ LBINT t:{−d/2..d/2}.
cmod (1 − char M' t)
   unfolding set-lebesgue-integral-def
   using integral-norm-bound[of - λx. indicator {u..v} x *R (1 − char M' x)
for u v] by simp
  also have 4: ... ≤ LBINT t:{−d/2..d/2}. ε / 4
   unfolding set-lebesgue-integral-def
   apply (rule integral-mono [OF 3])
    apply (simp add: emeasure-lborel-Icc-eq)
   apply (case-tac x ∈ {−d/2..d/2})
    apply auto
   apply (subst norm-minus-commute)
   apply (rule less-imp-le)
   apply (rule d1 [simplified])
   using d0 apply auto
   done
  also from d0 4 have ... = d * ε / 4
```

unfolding *set-lebesgue-integral-def* **by** *simp*
finally have *bound*: *cmod* (*CLBINT t*:{−*d*/2..*d*/2}. 1 − *char M' t*) ≤ *d* ∗ *ε* / 4 .
have *cmod* (1 − *char* (*M n*) *x*) ≤ 2 **for** *n x*
by (*rule order-trans* [*OF norm-triangle-ineq4*], *auto simp add*: *Mn.cmod-char-le-1*)
then have (λ*n*. *CLBINT t*:{−*d*/2..*d*/2}. 1 − *char* (*M n*) *t*) ⟶ (*CLBINT t*:{−*d*/2..*d*/2}. 1 − *char M' t*)
unfolding *set-lebesgue-integral-def*
apply (*intro integral-dominated-convergence*[**where** *w*=λ*x*. *indicator* {−*d*/2..*d*/2} *x* ∗$_R$ 2])
apply (*auto intro*!: *char-conv tendsto-intros*
simp: *emeasure-lborel-Icc-eq*
split: *split-indicator*)
done
hence *eventually* (λ*n*. *cmod* ((*CLBINT t*:{−*d*/2..*d*/2}. 1 − *char* (*M n*) *t*) − (*CLBINT t*:{−*d*/2..*d*/2}. 1 − *char M' t*)) < *d* ∗ *ε* / 4) *sequentially*
using *d0* ‹*ε* > 0› **apply** (*subst* (*asm*) *tendsto-iff*)
by (*subst* (*asm*) *dist-complex-def*, *drule spec*, *erule mp*, *auto*)
hence ∃ *N*. ∀ *n* ≥ *N*. *cmod* ((*CLBINT t*:{−*d*/2..*d*/2}. 1 − *char* (*M n*) *t*) − (*CLBINT t*:{−*d*/2..*d*/2}. 1 − *char M' t*)) < *d* ∗ *ε* / 4 **by** (*simp add*: *eventually-sequentially*)
then obtain *N*
where ∀ *n*≥*N*. *cmod* ((*CLBINT t*:{− *d* / 2..*d* / 2}. 1 − *char* (*M n*) *t*) − (*CLBINT t*:{− *d* / 2..*d* / 2}. 1 − *char M' t*)) < *d* ∗ *ε* / 4 ..
hence *N*: ⋀*n*. *n* ≥ *N* ⟹ *cmod* ((*CLBINT t*:{−*d*/2..*d*/2}. 1 − *char* (*M n*) *t*) − (*CLBINT t*:{−*d*/2..*d*/2}. 1 − *char M' t*)) < *d* ∗ *ε* / 4 **by** *auto*
{ fix *n*
assume *n* ≥ *N*
have *cmod* (*CLBINT t*:{−*d*/2..*d*/2}. 1 − *char* (*M n*) *t*) = *cmod* ((*CLBINT t*:{−*d*/2..*d*/2}. 1 − *char* (*M n*) *t*) − (*CLBINT t*:{−*d*/2..*d*/2}. 1 − *char M' t*) + (*CLBINT t*:{−*d*/2..*d*/2}. 1 − *char M' t*)) **by** *simp*
also have ... ≤ *cmod* ((*CLBINT t*:{−*d*/2..*d*/2}. 1 − *char* (*M n*) *t*) − (*CLBINT t*:{−*d*/2..*d*/2}. 1 − *char M' t*)) + *cmod*(*CLBINT t*:{−*d*/2..*d*/2}. 1 − *char M' t*)
by (*rule norm-triangle-ineq*)
also have ... < *d* ∗ *ε* / 4 + *d* ∗ *ε* / 4
by (*rule add-less-le-mono* [*OF N* [*OF* ‹*n* ≥ *N*›] *bound*])
also have ... = *d* ∗ *ε* / 2 **by** *auto*
finally have *cmod* (*CLBINT t*:{−*d*/2..*d*/2}. 1 − *char* (*M n*) *t*) < *d* ∗ *ε* / 2 .
hence *d* ∗ *ε* / 2 > *Re* (*CLBINT t*:{−*d*/2..*d*/2}. 1 − *char* (*M n*) *t*)
by (*rule order-le-less-trans* [*OF complex-Re-le-cmod*])
hence *d* ∗ *ε* / 2 > *Re* (*CLBINT t*:{−(*d*/2)..*d*/2}. 1 − *char* (*M n*) *t*) (**is** - > *?lhs*) **by** *simp*
also have *?lhs* ≥ (*d* / 2) ∗ *measure* (*M n*) {*x*. *abs x* ≥ 2 / (*d* / 2)}
using *d0* **by** (*intro main-bound*, *simp*)
finally (*xtrans*) **have** *d* ∗ *ε* / 2 > (*d* / 2) ∗ *measure* (*M n*) {*x*. *abs x* ≥ 2 /

```
(d / 2)} .
    with d0 ‹ε > 0› have ε > measure (M n) {x. abs x ≥ 2 / (d / 2)} by (simp
add: field-simps)
    hence ε > 1 − measure (M n) (UNIV − {x. abs x ≥ 2 / (d / 2)})
      apply (subst Mn.borel-UNIV [symmetric])
      by (subst Mn.prob-compl, auto)
    also have UNIV − {x. abs x ≥ 2 / (d / 2)} = {x. −(4 / d) < x ∧ x < (4
/ d)}
      using d0 apply (auto simp add: field-simps)

      apply (case-tac x ≥ 0, auto simp add: field-simps)
      apply (subgoal-tac 0 ≤ x * d, arith, rule mult-nonneg-nonneg, auto)
      apply (case-tac x ≥ 0, auto simp add: field-simps)
      apply (subgoal-tac x * d ≤ 0, arith)
      apply (rule mult-nonpos-nonneg, auto)
      by (case-tac x ≥ 0, auto simp add: field-simps)
    finally have measure (M n) {x. −(4 / d) < x ∧ x < (4 / d)} > 1 − ε
      by auto
  } note 6 = this
  { fix n :: nat
    have ∗: (UN (k :: nat). {− real k<..real k}) = UNIV
     by (auto, metis leI le-less-trans less-imp-le minus-less-iff reals-Archimedean2)
    have (λk. measure (M n) {− real k<..real k}) ⟶
       measure (M n) (UN (k :: nat). {− real k<..real k})
      by (rule Mn.finite-Lim-measure-incseq, auto simp add: incseq-def)
    hence (λk. measure (M n) {− real k<..real k}) ⟶ 1
      using Mn.prob-space unfolding ∗ Mn.borel-UNIV by simp
   hence eventually (λk. measure (M n) {− real k<..real k} > 1 − ε) sequentially
      apply (elim order-tendstoD (1))
      using ‹ε > 0› by auto
  } note 7 = this
  { fix n :: nat
    have eventually (λk. ∀ m < n. measure (M m) {− real k<..real k} > 1 − ε)
sequentially
      (is ?P n)
    proof (induct n)
      case (Suc n) with 7[of n] show ?case
        by eventually-elim (auto simp add: less-Suc-eq)
    qed simp
  } note 8 = this
  from 8 [of N] have ∃ K :: nat. ∀ k ≥ K. ∀ m<N. 1 − ε <
      Sigma-Algebra.measure (M m) {− real k<..real k}
    by (auto simp add: eventually-sequentially)
   hence ∃ K :: nat. ∀ m<N. 1 − ε < Sigma-Algebra.measure (M m) {− real
K<..real K} by auto
  then obtain K :: nat where
    ∀ m<N. 1 − ε < Sigma-Algebra.measure (M m) {− real K<..real K} ..
  hence K: ⋀m. m < N ⟹ 1 − ε < Sigma-Algebra.measure (M m) {− real
K<..real K}
```

```
    by auto
  let ?K' = max K (4 / d)
  have −?K' < ?K' ∧ (∀ n. 1 − ε < measure (M n) {−?K'<..?K'})
    using d0 apply auto
    apply (rule max.strict-coboundedI2, auto)
  proof −
    fix n
    show  1 − ε < measure (M n) {− max (real K) (4 / d)<..max (real K) (4
/ d)}
      apply (case-tac n < N)
      apply (rule order-less-le-trans)
      apply (erule K)
      apply (rule Mn.finite-measure-mono, auto)
      apply (rule order-less-le-trans)
      apply (rule 6, erule leI)
      by (rule Mn.finite-measure-mono, auto)
  qed
  thus ∃ a b. a < b ∧ (∀ n. 1 − ε < measure (M n) {a<..b}) by (intro exI)
 qed
 have tight: tight M
  by (auto simp: tight-def intro: assms tight-aux)
 show ?thesis
 proof (rule tight-subseq-weak-converge [OF real-distr-M real-distr-M' tight])
  fix s ν
  assume s: strict-mono (s :: nat ⇒ nat)
  assume nu: weak-conv-m (M ∘ s) ν
  assume ∗: real-distribution ν
  have 2: ⋀n. real-distribution ((M ∘ s) n) unfolding comp-def by (rule assms)
  have 3: ⋀t. (λn. char ((M ∘ s) n) t) ⟶ char ν t by (intro levy-continuity1
[OF 2 ∗ nu])
  have 4: ⋀t. (λn. char ((M ∘ s) n) t) = ((λn. char (M n) t) ∘ s) by (rule ext,
simp)
  have 5: ⋀t. (λn. char ((M ∘ s) n) t) ⟶ char M' t
   by (subst 4, rule LIMSEQ-subseq-LIMSEQ [OF - s], rule assms)
  hence char ν = char M' by (intro ext, intro LIMSEQ-unique [OF 3 5])
  hence ν = M' by (rule Levy-uniqueness [OF ∗ ‹real-distribution M'›])
  thus weak-conv-m (M ∘ s) M'
   by (elim subst) (rule nu)
 qed
qed

end
```

19 The Central Limit Theorem

```
theory Central-Limit-Theorem
 imports Levy
begin
```

theorem (**in** *prob-space*) *central-limit-theorem-zero-mean*:
fixes $X :: nat \Rightarrow 'a \Rightarrow real$
and $\mu :: real\ measure$
and $\sigma :: real$
and $S :: nat \Rightarrow 'a \Rightarrow real$
assumes *X-indep*: *indep-vars* $(\lambda i.\ borel)\ X\ UNIV$
and *X-mean-0*: $\bigwedge n.\ expectation\ (X\ n) = 0$
and σ*-pos*: $\sigma > 0$
and *X-square-integrable*: $\bigwedge n.\ integrable\ M\ (\lambda x.\ (X\ n\ x)^2)$
and *X-variance*: $\bigwedge n.\ variance\ (X\ n) = \sigma^2$
and *X-distrib*: $\bigwedge n.\ distr\ M\ borel\ (X\ n) = \mu$
defines $S\ n \equiv \lambda x.\ \sum i{<}n.\ X\ i\ x$
shows *weak-conv-m* $(\lambda n.\ distr\ M\ borel\ (\lambda x.\ S\ n\ x\ /\ sqrt\ (n * \sigma^2)))\ std\text{-}normal\text{-}distribution$
proof −
let $?S' = \lambda n\ x.\ S\ n\ x\ /\ sqrt\ (real\ n * \sigma^2)$
define φ **where** $\varphi\ n = char\ (distr\ M\ borel\ (?S'\ n))$ **for** n
define ψ **where** $\psi\ n\ t = char\ \mu\ (t\ /\ sqrt\ (\sigma^2 * n))$ **for** $n\ t$

have *X-rv* [*simp*, *measurable*]: *random-variable borel* $(X\ n)$ **for** n
using *X-indep* **unfolding** *indep-vars-def2* **by** *simp*
have *X-integrable* [*simp*, *intro*]: *integrable M* $(X\ n)$ **for** n
by (*rule square-integrable-imp-integrable*[*OF - X-square-integrable*]) *simp-all*
interpret μ: *real-distribution* μ
by (*subst X-distrib* [*symmetric*, *of 0*], *rule real-distribution-distr*, *simp*)

have μ*-integrable* [*simp*]: *integrable* μ $(\lambda x.\ x)$
and μ*-mean-integrable* [*simp*]: $\mu.expectation\ (\lambda x.\ x) = 0$
and μ*-square-integrable* [*simp*]: *integrable* μ $(\lambda x.\ x\hat{}2)$
and μ*-variance* [*simp*]: $\mu.expectation\ (\lambda x.\ x\hat{}2) = \sigma^2$
using *assms* **by** (*simp-all add*: *X-distrib* [*symmetric*, *of 0*] *integrable-distr-eq integral-distr*)

have *main*: $\forall_F\ n\ in\ sequentially.$
$cmod\ (\varphi\ n\ t - (1 + (-(t\hat{}2)\ /\ 2)\ /\ n)\hat{}n) \leq$
$t^2\ /\ (6 * \sigma^2) * (LINT\ x|\mu.\ min\ (6 * x^2)\ (|t\ /\ sqrt\ (\sigma^2 * n)| * |x|\ \hat{}\ 3))$ **for** t
proof (*rule eventually-sequentiallyI*)
fix $n :: nat$
assume $n \geq nat\ (ceiling\ (t\hat{}2\ /\ 4))$
hence n: $n \geq t\hat{}2\ /\ 4$ **by** (*subst nat-ceiling-le-eq* [*symmetric*])
let $?t = t\ /\ sqrt\ (\sigma^2 * n)$

define ψ' **where** $\psi'\ n\ i = char\ (distr\ M\ borel\ (\lambda x.\ X\ i\ x\ /\ sqrt\ (\sigma^2 * n)))$ **for** $n\ i$
have $*$: $\bigwedge n\ i\ t.\ \psi'\ n\ i\ t = \psi\ n\ t$
unfolding ψ*-def* ψ'*-def char-def*
by (*subst X-distrib* [*symmetric*]) (*auto simp*: *integral-distr*)

have $\varphi\ n\ t = char\ (distr\ M\ borel\ (\lambda x.\ \sum i{<}n.\ X\ i\ x\ /\ sqrt\ (\sigma^2 * real\ n)))\ t$

```
    by (auto simp: φ-def S-def sum-divide-distrib ac-simps)
  also have ... = (∏ i < n. ψ' n i t)
    unfolding ψ'-def
    apply (rule char-distr-sum)
    apply (rule indep-vars-compose2[where X=X])
    apply (rule indep-vars-subset)
    apply (rule X-indep)
    apply auto
    done
  also have ... = (ψ n t)^n
    by (auto simp add: * prod-constant)
  finally have φ-eq: φ n t =(ψ n t)^n .

  have norm (ψ n t - (1 - ?t^2 * σ² / 2)) ≤ ?t² / 6 * (LINT x|μ. min (6 *
x²) (|?t| * |x| ^ 3))
    unfolding ψ-def by (rule μ.char-approx3, auto)
  also have ?t^2 * σ² = t^2 / n
    using σ-pos by (simp add: power-divide)
  also have t^2 / n / 2 = (t^2 / 2) / n
    by simp
  finally have **: norm (ψ n t - (1 + (-(t^2) / 2) / n)) ≤
    ?t² / 6 * (LINT x|μ. min (6 * x²) (|?t| * |x| ^ 3)) by simp

  have norm (φ n t - (complex-of-real (1 + (-(t^2) / 2) / n))^n) ≤
      n * norm (ψ n t - (complex-of-real (1 + (-(t^2) / 2) / n)))
    using n
    by (auto intro!: norm-power-diff μ.cmod-char-le-1 abs-leI
          simp del: of-real-diff simp: of-real-diff[symmetric] divide-le-eq φ-eq
ψ-def)
  also have ... ≤ n * (?t² / 6 * (LINT x|μ. min (6 * x²) (|?t| * |x| ^ 3)))
    by (rule mult-left-mono [OF **], simp)
  also have ... = (t² / (6 * σ²) * (LINT x|μ. min (6 * x²) (|?t| * |x| ^ 3)))
    using σ-pos by (simp add: field-simps min-absorb2)
  finally show norm (φ n t - (1 + (-(t^2) / 2) / n)^n) ≤
     (t² / (6 * σ²) * (LINT x|μ. min (6 * x²) (|?t| * |x| ^ 3)))
    by simp
qed

show ?thesis
proof (rule levy-continuity)
  fix t
  let ?t = λn. t / sqrt (σ² * n)
  have ⋀x. (λn. min (6 * x²) (|t| * |x| ^ 3 / |sqrt (σ² * real n)|)) ⟶ 0
    using σ-pos
    by (auto simp: real-sqrt-mult min-absorb2
          intro!: tendsto-min[THEN tendsto-eq-rhs] sqrt-at-top[THEN filter-
lim-compose]
        filterlim-tendsto-pos-mult-at-top filterlim-at-top-imp-at-infinity
        tendsto-divide-0 filterlim-real-sequentially)
```

then have (λn. *LINT* $x|\mu$. *min* $(6 * x^2)$ $(|?t\ n| * |x|\ \hat{}\ 3)) \longrightarrow$ (*LINT* $x|\mu$. *0*)
by (*intro integral-dominated-convergence* [**where** $w = \lambda x.\ 6 * x\hat{}2$]) *auto*
then have $*$: ($\lambda n.\ t^2$ / $(6 * \sigma^2) *$ (*LINT* $x|\mu$. *min* $(6 * x^2)$ $(|?t\ n| * |x|\ \hat{}\ 3))) \longrightarrow$ *0*
by (*simp only*: *integral-zero tendsto-mult-right-zero*)

have (λn. *complex-of-real* $((1 + (-(t\hat{}2)\ /\ 2)\ /\ n)\hat{}n)) \longrightarrow$ *complex-of-real* (*exp* $(-(t\hat{}2)\ /\ 2)$)
by (*rule isCont-tendsto-compose* [*OF - tendsto-exp-limit-sequentially*]) *auto*
then have ($\lambda n.\ \varphi\ n\ t) \longrightarrow$ *complex-of-real* (*exp* $(-(t\hat{}2)\ /\ 2)$)
by (*rule Lim-transform*) (*rule Lim-null-comparison* [*OF main* $*$])
then show (λn. *char* (*distr M borel* (*?S′ n*)) *t*) $\longrightarrow$ *char std-normal-distribution t*
by (*simp add*: φ*-def char-std-normal-distribution*)
qed (*auto intro*!: *real-dist-normal-dist simp*: *S-def*)
qed

theorem (**in** *prob-space*) *central-limit-theorem*:
fixes *X* :: *nat* $\Rightarrow$ *'a* $\Rightarrow$ *real*
and μ :: *real measure*
and σ :: *real*
and *S* :: *nat* $\Rightarrow$ *'a* $\Rightarrow$ *real*
assumes *X-indep*: *indep-vars* (λi. *borel*) *X UNIV*
and *X-mean*: $\bigwedge n$. *expectation* $(X\ n) = m$
and σ*-pos*: $\sigma > 0$
and *X-square-integrable*: $\bigwedge n$. *integrable M* $(\lambda x.\ (X\ n\ x)^2)$
and *X-variance*: $\bigwedge n$. *variance* $(X\ n) = \sigma^2$
and *X-distrib*: $\bigwedge n$. *distr M borel* $(X\ n) = \mu$
defines $X'\ i\ x \equiv X\ i\ x - m$
shows *weak-conv-m* (λn. *distr M borel* ($\lambda x.\ (\sum i{<}n.\ X'\ i\ x)$ / *sqrt* $(n{*}\sigma^2)$))) *std-normal-distribution*
proof (*intro central-limit-theorem-zero-mean*)
show *indep-vars* (λi. *borel*) *X′ UNIV*
unfolding *X′-def*[*abs-def*] **using** *X-indep* **by** (*rule indep-vars-compose2*) *auto*
have *X-rv* [*simp*, *measurable*]: *random-variable borel* $(X\ n)$ **for** *n*
using *X-indep* **unfolding** *indep-vars-def2* **by** *simp*
have *X-integrable* [*simp*, *intro*]: *integrable M* $(X\ n)$ **for** *n*
by (*rule square-integrable-imp-integrable*[*OF - X-square-integrable*]) *simp-all*
show *expectation* $(X'\ n) = 0$ **for** *n*
using *X-mean* **by** (*auto simp*: *X′-def*[*abs-def*] *prob-space*)
show $\sigma > 0$ *integrable M* $(\lambda x.\ (X'\ n\ x)^2)$ *variance* $(X'\ n) = \sigma^2$ **for** *n*
using ‹$0 < \sigma$› *X-integrable X-mean X-square-integrable X-variance* **unfolding** *X′-def*
by (*auto simp*: *prob-space power2-diff*)
show *distr M borel* $(X'\ n)$ = *distr* μ *borel* ($\lambda x.\ x - m$) **for** *n*
unfolding *X-distrib*[*of n*, *symmetric*] **using** *X-integrable*
by (*subst distr-distr*) (*auto simp*: *X′-def*[*abs-def*] *comp-def*)
qed

end

theory *Discrete-Topology*
imports *HOL−Analysis.Analysis*
begin

Copy of discrete types with discrete topology. This space is polish.

typedef *'a discrete = UNIV::'a set*
morphisms *of-discrete discrete*
..

instantiation *discrete* :: (*type*) *metric-space*
begin

definition *dist-discrete* :: *'a discrete ⇒ 'a discrete ⇒ real*
where *dist-discrete n m = (if n = m then 0 else 1)*

definition *uniformity-discrete* :: (*'a discrete × 'a discrete*) *filter* **where**
(*uniformity*::(*'a discrete × 'a discrete*) *filter*) = (*INF e*∈{*0 <..*}. *principal* {(*x*, *y*). *dist x y < e*})

definition *open-discrete* :: *'a discrete set ⇒ bool* **where**
(*open*::*'a discrete set ⇒ bool*) *U* ⟷ (∀ *x*∈*U*. *eventually* (λ(*x'*, *y*). *x' = x* ⟶ *y* ∈ *U*) *uniformity*)

instance proof qed (*auto simp*: *uniformity-discrete-def open-discrete-def dist-discrete-def intro*: *exI*[**where** *x=1*])
end

lemma *open-discrete*: *open* (*S* :: *'a discrete set*)
unfolding *open-dist dist-discrete-def* **by** (*auto intro*!: *exI*[*of - 1 / 2*])

instance *discrete* :: (*type*) *complete-space*
proof
fix *X*::*nat*⇒*'a discrete*
assume *Cauchy X*
then obtain *n* **where** ∀ *m*≥*n*. *X n = X m*
by (*force simp*: *dist-discrete-def Cauchy-def split*: *if-split-asm dest*:*spec*[**where** *x=1*])
thus *convergent X*
by (*intro convergentI*[**where** *L=X n*] *tendstoI eventually-sequentiallyI*[*of n*])
(*simp add*: *dist-discrete-def*)
qed

instance *discrete* :: (*countable*) *countable*
proof
have *inj* (λ*c*::*'a discrete*. *to-nat* (*of-discrete c*))

by (*simp add*: *inj-on-def of-discrete-inject*)
thus ∃ *f*::'*a discrete* ⇒ *nat*. *inj f* **by** *blast*
qed

instance *discrete* :: (*countable*) *second-countable-topology*
proof
let *?B* = *range* (λ*n*::'*a discrete*. {*n*})
have ⋀*S*. *generate-topology ?B* (⋃ *x*∈*S*. {*x*})
by (*intro generate-topology-Union*) (*auto intro*: *generate-topology.intros*)
then have *open* = *generate-topology ?B*
by (*auto intro*!: *ext simp*: *open-discrete*)
moreover have *countable ?B* **by** *simp*
ultimately show ∃ *B*::'*a discrete set set*. *countable B* ∧ *open* = *generate-topology B* **by** *blast*
qed

instance *discrete* :: (*countable*) *polish-space* **..**

end

20 Probability mass function

theory *Probability-Mass-Function*
imports
Giry-Monad
HOL−Library.Multiset
begin

lemma *AE-emeasure-singleton*:
assumes *x*: *emeasure M* {*x*} ≠ *0* **and** *ae*: *AE x in M*. *P x* **shows** *P x*
proof −
from *x* **have** *x-M*: {*x*} ∈ *sets M*
by (*auto intro*: *emeasure-notin-sets*)
from *ae* **obtain** *N* **where** *N*: {*x*∈*space M*. ¬ *P x*} ⊆ *N emeasure M N* = *0 N* ∈ *sets M*
by (*auto elim*: *AE-E*)
{ assume ¬ *P x*
with *x-M*[*THEN sets.sets-into-space*] *N* **have** *emeasure M* {*x*} ≤ *emeasure M N*
by (*intro emeasure-mono*) *auto*
with *x N* **have** *False*
by (*auto simp*:) **}**
then show *P x* **by** *auto*
qed

lemma *AE-measure-singleton*: *measure M* {*x*} ≠ *0* ⟹ *AE x in M*. *P x* ⟹ *P x*
by (*metis AE-emeasure-singleton measure-def emeasure-empty measure-empty*)

lemma (**in** *finite-measure*) *AE-support-countable*:

assumes [*simp*]: *sets M = UNIV*
shows (*AE x in M. measure M {x} ≠ 0*) ⟷ (∃ *S. countable S* ∧ (*AE x in M. x ∈ S*))
proof
assume ∃ *S. countable S* ∧ (*AE x in M. x ∈ S*)
then obtain *S* **where** *S*[*intro*]: *countable S* **and** *ae*: *AE x in M. x ∈ S*
by *auto*
then have *emeasure M* (⋃*x*∈{*x*∈*S*. *emeasure M* {*x*} ≠ *0*}. {*x*}) =
(∫⁺ *x*. *emeasure M* {*x*} ∗ *indicator* {*x*∈*S*. *emeasure M* {*x*} ≠ *0*} *x* ∂*count-space UNIV*)
by (*subst emeasure-UN-countable*)
(*auto simp*: *disjoint-family-on-def nn-integral-restrict-space*[*symmetric*] *restrict-count-space*)
also have ... = (∫⁺ *x*. *emeasure M* {*x*} ∗ *indicator S x* ∂*count-space UNIV*)
by (*auto intro*!: *nn-integral-cong split*: *split-indicator*)
also have ... = *emeasure M* (⋃*x*∈*S*. {*x*})
by (*subst emeasure-UN-countable*)
(*auto simp*: *disjoint-family-on-def nn-integral-restrict-space*[*symmetric*] *restrict-count-space*)
also have ... = *emeasure M* (*space M*)
using *ae* **by** (*intro emeasure-eq-AE*) *auto*
finally have *emeasure M* {*x* ∈ *space M*. *x*∈*S* ∧ *emeasure M* {*x*} ≠ *0*} = *emeasure M* (*space M*)
by (*simp add*: *emeasure-single-in-space cong*: *rev-conj-cong*)
with *finite-measure-compl*[*of* {*x* ∈ *space M*. *x*∈*S* ∧ *emeasure M* {*x*} ≠ *0*}]
have *AE x in M. x ∈ S* ∧ *emeasure M* {*x*} ≠ *0*
by (*intro AE-I*[*OF order-refl*]) (*auto simp*: *emeasure-eq-measure measure-nonneg set-diff-eq cong*: *conj-cong*)
then show *AE x in M. measure M* {*x*} ≠ *0*
by (*auto simp*: *emeasure-eq-measure*)
qed (*auto intro*!: *exI*[*of* - {*x*. *measure M* {*x*} ≠ *0*}] *countable-support*)

20.1 PMF as measure

typedef *'a pmf* = {*M* :: *'a measure. prob-space M* ∧ *sets M = UNIV* ∧ (*AE x in M. measure M* {*x*} ≠ *0*)}
morphisms *measure-pmf Abs-pmf*
by (*intro exI*[*of* - *uniform-measure* (*count-space UNIV*) {*undefined*}])
(*auto intro*!: *prob-space-uniform-measure AE-uniform-measureI*)

declare [[*coercion measure-pmf*]]

lemma *prob-space-measure-pmf*: *prob-space* (*measure-pmf p*)
using *pmf.measure-pmf*[*of p*] **by** *auto*

interpretation *measure-pmf*: *prob-space measure-pmf M* **for** *M*
by (*rule prob-space-measure-pmf*)

interpretation *measure-pmf*: *subprob-space measure-pmf M* **for** *M*

by (*rule prob-space-imp-subprob-space*) *unfold-locales*

lemma *subprob-space-measure-pmf*: *subprob-space* (*measure-pmf x*)
by *unfold-locales*

locale *pmf-as-measure*
begin

setup-lifting *type-definition-pmf*

end

context
begin

interpretation *pmf-as-measure* **.**

lemma *sets-measure-pmf*[*simp*]: *sets* (*measure-pmf p*) = *UNIV*
by *transfer blast*

lemma *sets-measure-pmf-count-space*[*measurable-cong*]:
sets (*measure-pmf M*) = *sets* (*count-space UNIV*)
by *simp*

lemma *space-measure-pmf*[*simp*]: *space* (*measure-pmf p*) = *UNIV*
using *sets-eq-imp-space-eq*[*of measure-pmf p count-space UNIV*] **by** *simp*

lemma *measure-pmf-UNIV* [*simp*]: *measure* (*measure-pmf p*) *UNIV* = *1*
using *measure-pmf.prob-space*[*of p*] **by** *simp*

lemma *measure-pmf-in-subprob-algebra*[*measurable* (*raw*)]: *measure-pmf x* ∈ *space* (*subprob-algebra* (*count-space UNIV*))
by (*simp add*: *space-subprob-algebra subprob-space-measure-pmf*)

lemma *measurable-pmf-measure1*[*simp*]: *measurable* (*M* :: *'a pmf*) *N* = *UNIV* → *space N*
by (*auto simp*: *measurable-def*)

lemma *measurable-pmf-measure2*[*simp*]: *measurable N* (*M* :: *'a pmf*) = *measurable N* (*count-space UNIV*)
by (*intro measurable-cong-sets*) *simp-all*

lemma *measurable-pair-restrict-pmf2*:
assumes *countable A*
assumes [*measurable*]: ⋀*y*. *y* ∈ *A* ⟹ (λ*x*. *f* (*x*, *y*)) ∈ *measurable M L*
shows *f* ∈ *measurable* (*M* ⨂$_M$ *restrict-space* (*measure-pmf N*) *A*) *L* (**is** *f* ∈ *measurable ?M* -)
proof −
have [*measurable-cong*]: *sets* (*restrict-space* (*count-space UNIV*) *A*) = *sets* (*count-space*

A)
by (*simp add*: *restrict-count-space*)

show *?thesis*
by (*intro measurable-compose-countable'*[**where** *f*=λ*a b*. *f* (*fst b*, *a*) **and** *g*=*snd* **and** *I*=*A*,
unfolded prod.collapse] *assms*)
measurable
qed

lemma *measurable-pair-restrict-pmf1*:
assumes *countable A*
assumes [*measurable*]: ⋀*x*. *x* ∈ *A* ⟹ (λ*y*. *f* (*x*, *y*)) ∈ *measurable N L*
shows *f* ∈ *measurable* (*restrict-space* (*measure-pmf M*) *A* ⨂$_M$ *N*) *L*
proof −
have [*measurable-cong*]: *sets* (*restrict-space* (*count-space UNIV*) *A*) = *sets* (*count-space A*)
by (*simp add*: *restrict-count-space*)

show *?thesis*
by (*intro measurable-compose-countable'*[**where** *f*=λ*a b*. *f* (*a*, *snd b*) **and** *g*=*fst* **and** *I*=*A*,
unfolded prod.collapse] *assms*)
measurable
qed

lift-definition *pmf* :: *'a pmf* ⇒ *'a* ⇒ *real* **is** λ*M x*. *measure M* {*x*} .

lift-definition *set-pmf* :: *'a pmf* ⇒ *'a set* **is** λ*M*. {*x*. *measure M* {*x*} ≠ *0*} .
declare [[*coercion set-pmf*]]

lemma *AE-measure-pmf*: *AE x in* (*M*::*'a pmf*). *x* ∈ *M*
by *transfer simp*

lemma *emeasure-pmf-single-eq-zero-iff*:
fixes *M* :: *'a pmf*
shows *emeasure M* {*y*} = *0* ⟷ *y* ∉ *M*
unfolding *set-pmf.rep-eq* **by** (*simp add*: *measure-pmf.emeasure-eq-measure*)

lemma *AE-measure-pmf-iff*: (*AE x in measure-pmf M*. *P x*) ⟷ (∀ *y*∈*M*. *P y*)
using *AE-measure-singleton*[*of M*] *AE-measure-pmf*[*of M*]
by (*auto simp*: *set-pmf.rep-eq*)

lemma *AE-pmfI*: (⋀*y*. *y* ∈ *set-pmf M* ⟹ *P y*) ⟹ *almost-everywhere* (*measure-pmf M*) *P*
by(*simp add*: *AE-measure-pmf-iff*)

lemma *countable-set-pmf* [*simp*]: *countable* (*set-pmf p*)
by *transfer* (*metis prob-space.finite-measure finite-measure.countable-support*)

lemma *pmf-positive*: $x \in$ *set-pmf* $p \Longrightarrow 0 <$ *pmf* $p\ x$
by *transfer* (*simp add*: *less-le*)

lemma *pmf-nonneg*[*simp*]: $0 \leq$ *pmf* $p\ x$
by *transfer simp*

lemma *pmf-not-neg* [*simp*]: $\neg$*pmf* $p\ x < 0$
by (*simp add*: *not-less pmf-nonneg*)

lemma *pmf-pos* [*simp*]: *pmf* $p\ x \neq 0 \Longrightarrow$ *pmf* $p\ x > 0$
using *pmf-nonneg*[*of* $p\ x$] **by** *linarith*

lemma *pmf-le-1*: *pmf* $p\ x \leq 1$
by (*simp add*: *pmf.rep-eq*)

lemma *set-pmf-not-empty*: *set-pmf* $M \neq \{\}$
using *AE-measure-pmf*[*of* M] **by** (*intro notI*) *simp*

lemma *set-pmf-iff*: $x \in$ *set-pmf* $M \longleftrightarrow$ *pmf* $M\ x \neq 0$
by *transfer simp*

lemma *pmf-positive-iff*: $0 <$ *pmf* $p\ x \longleftrightarrow x \in$ *set-pmf* p
unfolding *less-le* **by** (*simp add*: *set-pmf-iff*)

lemma *set-pmf-eq*: *set-pmf* $M = \{x.$ *pmf* $M\ x \neq 0\}$
by (*auto simp*: *set-pmf-iff*)

lemma *set-pmf-eq'*: *set-pmf* $p = \{x.$ *pmf* $p\ x > 0\}$
proof *safe*
fix x **assume** $x \in$ *set-pmf* p
hence *pmf* $p\ x \neq 0$ **by** (*auto simp*: *set-pmf-eq*)
with *pmf-nonneg*[*of* $p\ x$] **show** *pmf* $p\ x > 0$ **by** *simp*
qed (*auto simp*: *set-pmf-eq*)

lemma *emeasure-pmf-single*:
fixes $M :: 'a$ *pmf*
shows *emeasure* $M\ \{x\} =$ *pmf* $M\ x$
by *transfer* (*simp add*: *finite-measure.emeasure-eq-measure*[*OF prob-space.finite-measure*])

lemma *measure-pmf-single*: *measure* (*measure-pmf* M) $\{x\} =$ *pmf* $M\ x$
using *emeasure-pmf-single*[*of* $M\ x$] **by**(*simp add*: *measure-pmf.emeasure-eq-measure pmf-nonneg measure-nonneg*)

lemma *emeasure-measure-pmf-finite*: *finite* $S \Longrightarrow$ *emeasure* (*measure-pmf* M) S $= (\sum s \in S.$ *pmf* $M\ s)$
by (*subst emeasure-eq-sum-singleton*) (*auto simp*: *emeasure-pmf-single pmf-nonneg*)

lemma *measure-measure-pmf-finite*: *finite* $S \Longrightarrow$ *measure* (*measure-pmf* M) $S =$

sum (*pmf M*) *S*
using *emeasure-measure-pmf-finite*[*of S M*]
by (*simp add*: *measure-pmf.emeasure-eq-measure measure-nonneg sum-nonneg pmf-nonneg*)

lemma *sum-pmf-eq-1*:
assumes *finite A set-pmf p* ⊆ *A*
shows (∑ *x*∈*A*. *pmf p x*) = *1*
proof −
have (∑ *x*∈*A*. *pmf p x*) = *measure-pmf.prob p A*
by (*simp add*: *measure-measure-pmf-finite assms*)
also from *assms* **have** ... = *1*
by (*subst measure-pmf.prob-eq-1*) (*auto simp*: *AE-measure-pmf-iff*)
finally show *?thesis* **.**
qed

lemma *nn-integral-measure-pmf-support*:
fixes *f* :: *'a* ⇒ *ennreal*
assumes *f*: *finite A* **and** *nn*: ⋀*x*. *x* ∈ *A* ⟹ *0* ≤ *f x* ⋀*x*. *x* ∈ *set-pmf M* ⟹ *x* ∉ *A* ⟹ *f x* = *0*
shows (∫⁺*x*. *f x* ∂*measure-pmf M*) = (∑ *x*∈*A*. *f x* ∗ *pmf M x*)
proof −
have (∫⁺*x*. *f x* ∂*M*) = (∫⁺*x*. *f x* ∗ *indicator A x* ∂*M*)
using *nn* **by** (*intro nn-integral-cong-AE*) (*auto simp*: *AE-measure-pmf-iff split*: *split-indicator*)
also have ... = (∑ *x*∈*A*. *f x* ∗ *emeasure M* {*x*})
using *assms* **by** (*intro nn-integral-indicator-finite*) *auto*
finally show *?thesis*
by (*simp add*: *emeasure-measure-pmf-finite*)
qed

lemma *nn-integral-measure-pmf-finite*:
fixes *f* :: *'a* ⇒ *ennreal*
assumes *f*: *finite* (*set-pmf M*) **and** *nn*: ⋀*x*. *x* ∈ *set-pmf M* ⟹ *0* ≤ *f x*
shows (∫⁺*x*. *f x* ∂*measure-pmf M*) = (∑ *x*∈*set-pmf M*. *f x* ∗ *pmf M x*)
using *assms* **by** (*intro nn-integral-measure-pmf-support*) *auto*

lemma *integrable-measure-pmf-finite*:
fixes *f* :: *'a* ⇒ *'b*::{*banach*, *second-countable-topology*}
shows *finite* (*set-pmf M*) ⟹ *integrable M f*
by (*auto intro*!: *integrableI-bounded simp*: *nn-integral-measure-pmf-finite ennreal-mult-less-top*)

lemma *integral-measure-pmf-real*:
assumes [*simp*]: *finite A* **and** ⋀*a*. *a* ∈ *set-pmf M* ⟹ *f a* ≠ *0* ⟹ *a* ∈ *A*
shows (∫ *x*. *f x* ∂*measure-pmf M*) = (∑ *a*∈*A*. *f a* ∗ *pmf M a*)
proof −
have (∫ *x*. *f x* ∂*measure-pmf M*) = (∫ *x*. *f x* ∗ *indicator A x* ∂*measure-pmf M*)
using *assms*(*2*) **by** (*intro integral-cong-AE*) (*auto split*: *split-indicator simp*: *AE-measure-pmf-iff*)

also have ... = ($\sum a{\in}A.\ f\ a * pmf\ M\ a$)
by (*subst integral-indicator-finite-real*)
(*auto simp*: *measure-def emeasure-measure-pmf-finite pmf-nonneg*)
finally show *?thesis* **.**
qed

lemma *integrable-pmf*: *integrable* (*count-space X*) (*pmf M*)
proof −
have ($\int^+ x.$ *pmf M x* ∂*count-space X*) = ($\int^+ x.$ *pmf M x* ∂*count-space* (*M* ∩ *X*))
by (*auto simp add*: *nn-integral-count-space-indicator set-pmf-iff intro*!: *nn-integral-cong split*: *split-indicator*)
then have *integrable* (*count-space X*) (*pmf M*) = *integrable* (*count-space* (*M* ∩ *X*)) (*pmf M*)
by (*simp add*: *integrable-iff-bounded pmf-nonneg*)
then show *?thesis*
by (*simp add*: *pmf.rep-eq measure-pmf.integrable-measure disjoint-family-on-def*)
qed

lemma *integral-pmf*: ($\int x.$ *pmf M x* ∂*count-space X*) = *measure M X*
proof −
have ($\int x.$ *pmf M x* ∂*count-space X*) = ($\int^+ x.$ *pmf M x* ∂*count-space X*)
by (*simp add*: *pmf-nonneg integrable-pmf nn-integral-eq-integral*)
also have ... = ($\int^+ x.$ *emeasure M* {*x*} ∂*count-space* (*X* ∩ *M*))
by (*auto intro*!: *nn-integral-cong-AE split*: *split-indicator*
simp: *pmf.rep-eq measure-pmf.emeasure-eq-measure nn-integral-count-space-indicator AE-count-space set-pmf-iff*)
also have ... = *emeasure M* (*X* ∩ *M*)
by (*rule emeasure-countable-singleton*[*symmetric*]) (*auto intro*: *countable-set-pmf*)
also have ... = *emeasure M X*
by (*auto intro*!: *emeasure-eq-AE simp*: *AE-measure-pmf-iff*)
finally show *?thesis*
by (*simp add*: *measure-pmf.emeasure-eq-measure measure-nonneg integral-nonneg pmf-nonneg*)
qed

lemma *integral-pmf-restrict*:
(*f*::′*a* ⇒ ′*b*::{*banach*, *second-countable-topology*}) ∈ *borel-measurable* (*count-space UNIV*) ⟹
($\int x.$ *f x* ∂*measure-pmf M*) = ($\int x.$ *f x* ∂*restrict-space M M*)
by (*auto intro*!: *integral-cong-AE simp add*: *integral-restrict-space AE-measure-pmf-iff*)

lemma *emeasure-pmf*: *emeasure* (*M*::′*a pmf*) *M* = *1*
proof −
have *emeasure* (*M*::′*a pmf*) *M* = *emeasure* (*M*::′*a pmf*) (*space M*)
by (*intro emeasure-eq-AE*) (*simp-all add*: *AE-measure-pmf*)
then show *?thesis*
using *measure-pmf.emeasure-space-1* **by** *simp*
qed

lemma *emeasure-pmf-UNIV* [*simp*]: *emeasure* (*measure-pmf M*) *UNIV* = *1*
using *measure-pmf.emeasure-space-1*[*of M*] **by** *simp*

lemma *in-null-sets-measure-pmfI*:
A ∩ *set-pmf p* = {} ⟹ *A* ∈ *null-sets* (*measure-pmf p*)
using *emeasure-eq-0-AE*[**where** *?P*=λ*x*. *x* ∈ *A* **and** *M*=*measure-pmf p*]
by(*auto simp add*: *null-sets-def AE-measure-pmf-iff*)

lemma *measure-subprob*: *measure-pmf M* ∈ *space* (*subprob-algebra* (*count-space UNIV*))
by (*simp add*: *space-subprob-algebra subprob-space-measure-pmf*)

20.2 Monad Interpretation

lemma *measurable-measure-pmf*[*measurable*]:
(λ*x*. *measure-pmf* (*M x*)) ∈ *measurable* (*count-space UNIV*) (*subprob-algebra* (*count-space UNIV*))
by (*auto simp*: *space-subprob-algebra intro*!: *prob-space-imp-subprob-space*) *unfold-locales*

lemma *bind-measure-pmf-cong*:
assumes ⋀*x*. *A x* ∈ *space* (*subprob-algebra N*) ⋀*x*. *B x* ∈ *space* (*subprob-algebra N*)
assumes ⋀*i*. *i* ∈ *set-pmf x* ⟹ *A i* = *B i*
shows *bind* (*measure-pmf x*) *A* = *bind* (*measure-pmf x*) *B*
proof (*rule measure-eqI*)
show *sets* (*measure-pmf x* ⤜ *A*) = *sets* (*measure-pmf x* ⤜ *B*)
using *assms* **by** (*subst* (*1 2*) *sets-bind*) (*auto simp*: *space-subprob-algebra*)
next
fix *X* **assume** *X* ∈ *sets* (*measure-pmf x* ⤜ *A*)
then have *X*: *X* ∈ *sets N*
using *assms* **by** (*subst* (*asm*) *sets-bind*) (*auto simp*: *space-subprob-algebra*)
show *emeasure* (*measure-pmf x* ⤜ *A*) *X* = *emeasure* (*measure-pmf x* ⤜ *B*) *X*
using *assms*
by (*subst* (*1 2*) *emeasure-bind*[**where** *N*=*N*, *OF* - - *X*])
(*auto intro*!: *nn-integral-cong-AE simp*: *AE-measure-pmf-iff*)
qed

lift-definition *bind-pmf* :: *'a pmf* ⇒ (*'a* ⇒ *'b pmf*) ⇒ *'b pmf* **is** *bind*
proof (*clarify*, *intro conjI*)
fix *f* :: *'a measure* **and** *g* :: *'a* ⇒ *'b measure*
assume *prob-space f*
then interpret *f*: *prob-space f* **.**
assume *sets f* = *UNIV* **and** *ae-f*: *AE x in f*. *measure f* {*x*} ≠ *0*
then have *s-f*[*simp*]: *sets f* = *sets* (*count-space UNIV*)
by *simp*
assume *g*: ⋀*x*. *prob-space* (*g x*) ∧ *sets* (*g x*) = *UNIV* ∧ (*AE y in g x*. *measure* (*g x*) {*y*} ≠ *0*)

then have *g*: ⋀*x*. *prob-space* (*g x*) **and** *s-g*[*simp*]: ⋀*x*. *sets* (*g x*) = *sets* (*count-space UNIV*)
and *ae-g*: ⋀*x*. *AE y in g x*. *measure* (*g x*) {*y*} ≠ *0*
by *auto*

have [*measurable*]: *g* ∈ *measurable f* (*subprob-algebra* (*count-space UNIV*))
by (*auto simp*: *measurable-def space-subprob-algebra prob-space-imp-subprob-space g*)

show *prob-space* (*f* ⤜ *g*)
using *g* **by** (*intro f.prob-space-bind*[**where** *S*=*count-space UNIV*]) *auto*
then interpret *fg*: *prob-space f* ⤜ *g* **.**
show [*simp*]: *sets* (*f* ⤜ *g*) = *UNIV*
using *sets-eq-imp-space-eq*[*OF s-f*]
by (*subst sets-bind*[**where** *N*=*count-space UNIV*]) *auto*
show *AE x in f* ⤜ *g*. *measure* (*f* ⤜ *g*) {*x*} ≠ *0*
apply (*simp add*: *fg.prob-eq-0 AE-bind*[**where** *B*=*count-space UNIV*])
using *ae-f*
apply *eventually-elim*
using *ae-g*
apply *eventually-elim*
apply (*auto dest*: *AE-measure-singleton*)
done
qed

adhoc-overloading *Monad-Syntax.bind bind-pmf*

lemma *ennreal-pmf-bind*: *pmf* (*bind-pmf N f*) *i* = (∫⁺*x*. *pmf* (*f x*) *i* ∂*measure-pmf N*)
unfolding *pmf.rep-eq bind-pmf.rep-eq*
by (*auto simp*: *measure-pmf.measure-bind*[**where** *N*=*count-space UNIV*] *measure-subprob measure-nonneg*
intro!: *nn-integral-eq-integral*[*symmetric*] *measure-pmf.integrable-const-bound*[**where** *B*=*1*])

lemma *pmf-bind*: *pmf* (*bind-pmf N f*) *i* = (∫ *x*. *pmf* (*f x*) *i* ∂*measure-pmf N*)
using *ennreal-pmf-bind*[*of N f i*]
by (*subst* (*asm*) *nn-integral-eq-integral*)
(*auto simp*: *pmf-nonneg pmf-le-1 pmf-nonneg integral-nonneg*
intro!: *nn-integral-eq-integral*[*symmetric*] *measure-pmf.integrable-const-bound*[**where** *B*=*1*])

lemma *bind-pmf-const*[*simp*]: *bind-pmf M* (λ*x*. *c*) = *c*
by *transfer* (*simp add*: *bind-const′ prob-space-imp-subprob-space*)

lemma *set-bind-pmf*[*simp*]: *set-pmf* (*bind-pmf M N*) = (⋃ *M*∈*set-pmf M*. *set-pmf* (*N M*))
proof −
have *set-pmf* (*bind-pmf M N*) = {*x*. *ennreal* (*pmf* (*bind-pmf M N*) *x*) ≠ *0*}

by (*simp add*: *set-pmf-eq pmf-nonneg*)
also have ... = ($\bigcup M \in$*set-pmf M*. *set-pmf* (*N M*))
unfolding *ennreal-pmf-bind*
by (*subst nn-integral-0-iff-AE*) (*auto simp*: *AE-measure-pmf-iff pmf-nonneg set-pmf-eq*)
finally show *?thesis* **.**
qed

lemma *bind-pmf-cong* [*fundef-cong*]:
assumes $p = q$
shows ($\bigwedge x.\ x \in$ *set-pmf q* $\Longrightarrow$ *f x* = *g x*) $\Longrightarrow$ *bind-pmf p f* = *bind-pmf q g*
unfolding ‹$p = q$›[*symmetric*] *measure-pmf-inject*[*symmetric*] *bind-pmf.rep-eq*
by (*auto simp*: *AE-measure-pmf-iff Pi-iff space-subprob-algebra subprob-space-measure-pmf sets-bind*[**where** *N*=*count-space UNIV*] *emeasure-bind*[**where** *N*=*count-space UNIV*]
intro!: *nn-integral-cong-AE measure-eqI*)

lemma *bind-pmf-cong-simp*:
$p = q \Longrightarrow (\bigwedge x.\ x \in$ *set-pmf q* =*simp*=> *f x* = *g x*) $\Longrightarrow$ *bind-pmf p f* = *bind-pmf q g*
by (*simp add*: *simp-implies-def cong*: *bind-pmf-cong*)

lemma *measure-pmf-bind*: *measure-pmf* (*bind-pmf M f*) = (*measure-pmf M* $\gg\!=$ (λx. *measure-pmf* (*f x*)))
by *transfer simp*

lemma *nn-integral-bind-pmf*[*simp*]: ($\int^+ x.\ f\ x\ \partial$*bind-pmf M N*) = ($\int^+ x.\ \int^+ y.\ f\ y\ \partial N\ x\ \partial M$)
using *measurable-measure-pmf*[*of N*]
unfolding *measure-pmf-bind*
apply (*intro nn-integral-bind*[**where** *B*=*count-space UNIV*])
apply *auto*
done

lemma *emeasure-bind-pmf*[*simp*]: *emeasure* (*bind-pmf M N*) *X* = ($\int^+ x.$ *emeasure* (*N x*) *X* ∂M)
using *measurable-measure-pmf*[*of N*]
unfolding *measure-pmf-bind*
by (*subst emeasure-bind*[**where** *N*=*count-space UNIV*]) *auto*

lift-definition *return-pmf* :: *'a* $\Rightarrow$ *'a pmf* **is** *return* (*count-space UNIV*)
by (*auto intro*!: *prob-space-return simp*: *AE-return measure-return*)

lemma *bind-return-pmf*: *bind-pmf* (*return-pmf x*) *f* = *f x*
by *transfer*
(*auto intro*!: *prob-space-imp-subprob-space bind-return*[**where** *N*=*count-space UNIV*]
simp: *space-subprob-algebra*)

lemma *set-return-pmf*[*simp*]: *set-pmf* (*return-pmf x*) = {*x*}
by *transfer* (*auto simp add*: *measure-return split*: *split-indicator*)

lemma *bind-return-pmf'*: *bind-pmf N return-pmf* = *N*
proof (*transfer*, *clarify*)
fix *N* :: *'a measure* **assume** *sets N* = *UNIV* **then show** *N* ⤜ *return* (*count-space UNIV*) = *N*
by (*subst return-sets-cong*[**where** *N*=*N*]) (*simp-all add*: *bind-return'*)
qed

lemma *bind-assoc-pmf*: *bind-pmf* (*bind-pmf A B*) *C* = *bind-pmf A* (λ*x*. *bind-pmf* (*B x*) *C*)
by *transfer*
(*auto intro*!: *bind-assoc*[**where** *N*=*count-space UNIV* **and** *R*=*count-space UNIV*]
simp: *measurable-def space-subprob-algebra prob-space-imp-subprob-space*)

definition *map-pmf f M* = *bind-pmf M* (λ*x*. *return-pmf* (*f x*))

lemma *map-bind-pmf*: *map-pmf f* (*bind-pmf M g*) = *bind-pmf M* (λ*x*. *map-pmf f* (*g x*))
by (*simp add*: *map-pmf-def bind-assoc-pmf*)

lemma *bind-map-pmf*: *bind-pmf* (*map-pmf f M*) *g* = *bind-pmf M* (λ*x*. *g* (*f x*))
by (*simp add*: *map-pmf-def bind-assoc-pmf bind-return-pmf*)

lemma *map-pmf-transfer*[*transfer-rule*]:
rel-fun (=) (*rel-fun cr-pmf cr-pmf*) (λ*f M*. *distr M* (*count-space UNIV*) *f*) *map-pmf*
proof −
have *rel-fun* (=) (*rel-fun pmf-as-measure.cr-pmf pmf-as-measure.cr-pmf*)
(λ*f M*. *M* ⤜ (*return* (*count-space UNIV*) *o f*)) *map-pmf*
unfolding *map-pmf-def*[*abs-def*] *comp-def* **by** *transfer-prover*
then show *?thesis*
by (*force simp*: *rel-fun-def cr-pmf-def bind-return-distr*)
qed

lemma *map-pmf-rep-eq*:
measure-pmf (*map-pmf f M*) = *distr* (*measure-pmf M*) (*count-space UNIV*) *f*
unfolding *map-pmf-def bind-pmf.rep-eq comp-def return-pmf.rep-eq*
using *bind-return-distr*[*of M f count-space UNIV*] **by** (*simp add*: *comp-def*)

lemma *map-pmf-id*[*simp*]: *map-pmf id* = *id*
by (*rule*, *transfer*) (*auto simp*: *emeasure-distr measurable-def intro*!: *measure-eqI*)

lemma *map-pmf-ident*[*simp*]: *map-pmf* (λ*x*. *x*) = (λ*x*. *x*)
using *map-pmf-id* **unfolding** *id-def* **.**

lemma *map-pmf-compose*: *map-pmf* (*f* ∘ *g*) = *map-pmf f* ∘ *map-pmf g*

by (*rule*, *transfer*) (*simp add*: *distr-distr*[*symmetric*, **where** *N*=*count-space UNIV*] *measurable-def*)

lemma *map-pmf-comp*: *map-pmf f* (*map-pmf g M*) = *map-pmf* (λ*x*. *f* (*g x*)) *M*
using *map-pmf-compose*[*of f g*] **by** (*simp add*: *comp-def*)

lemma *map-pmf-cong*: *p* = *q* ⟹ (⋀*x*. *x* ∈ *set-pmf q* ⟹ *f x* = *g x*) ⟹ *map-pmf f p* = *map-pmf g q*
unfolding *map-pmf-def* **by** (*rule bind-pmf-cong*) *auto*

lemma *pmf-set-map*: *set-pmf* ∘ *map-pmf f* = (‘) *f* ∘ *set-pmf*
by (*auto simp add*: *comp-def fun-eq-iff map-pmf-def*)

lemma *set-map-pmf*[*simp*]: *set-pmf* (*map-pmf f M*) = *f*‘*set-pmf M*
using *pmf-set-map*[*of f*] **by** (*auto simp*: *comp-def fun-eq-iff*)

lemma *emeasure-map-pmf*[*simp*]: *emeasure* (*map-pmf f M*) *X* = *emeasure M* (*f* −‘ *X*)
unfolding *map-pmf-rep-eq* **by** (*subst emeasure-distr*) *auto*

lemma *measure-map-pmf*[*simp*]: *measure* (*map-pmf f M*) *X* = *measure M* (*f* −‘ *X*)
using *emeasure-map-pmf*[*of f M X*] **by**(*simp add*: *measure-pmf*.*emeasure-eq-measure measure-nonneg*)

lemma *nn-integral-map-pmf*[*simp*]: (∫⁺*x*. *f x* ∂*map-pmf g M*) = (∫⁺*x*. *f* (*g x*) ∂*M*)
unfolding *map-pmf-rep-eq* **by** (*intro nn-integral-distr*) *auto*

lemma *ennreal-pmf-map*: *pmf* (*map-pmf f p*) *x* = (∫⁺ *y*. *indicator* (*f* −‘ {*x*}) *y* ∂*measure-pmf p*)
proof (*transfer fixing*: *f x*)
fix *p* :: *'b measure*
presume *prob-space p*
then interpret *prob-space p* **.**
presume *sets p* = *UNIV*
then show *ennreal* (*measure* (*distr p* (*count-space UNIV*) *f*) {*x*}) = *integral*^{N} *p* (*indicator* (*f* −‘ {*x*}))
by(*simp add*: *measure-distr measurable-def emeasure-eq-measure*)
qed *simp-all*

lemma *pmf-map*: *pmf* (*map-pmf f p*) *x* = *measure p* (*f* −‘ {*x*})
proof (*transfer fixing*: *f x*)
fix *p* :: *'b measure*
presume *prob-space p*
then interpret *prob-space p* **.**
presume *sets p* = *UNIV*
then show *measure* (*distr p* (*count-space UNIV*) *f*) {*x*} = *measure p* (*f* −‘ {*x*})
by(*simp add*: *measure-distr measurable-def emeasure-eq-measure*)

qed *simp-all*

lemma *nn-integral-pmf*: $(\int^+ x.\ pmf\ p\ x\ \partial count\text{-}space\ A) = emeasure\ (measure\text{-}pmf\ p)\ A$
proof −
have $(\int^+ x.\ pmf\ p\ x\ \partial count\text{-}space\ A) = (\int^+ x.\ pmf\ p\ x\ \partial count\text{-}space\ (A \cap set\text{-}pmf\ p))$
by(*auto simp add*: *nn-integral-count-space-indicator indicator-def set-pmf-iff intro*: *nn-integral-cong*)
also have $\ldots = emeasure\ (measure\text{-}pmf\ p)\ (\bigcup x{\in}A \cap set\text{-}pmf\ p.\ \{x\})$
by(*subst emeasure-UN-countable*)(*auto simp add*: *emeasure-pmf-single disjoint-family-on-def*)
also have $\ldots = emeasure\ (measure\text{-}pmf\ p)\ ((\bigcup x{\in}A \cap set\text{-}pmf\ p.\ \{x\}) \cup \{x.\ x \in A \wedge x \notin set\text{-}pmf\ p\})$
by(*rule emeasure-Un-null-set*[*symmetric*])(*auto intro*: *in-null-sets-measure-pmfI*)
also have $\ldots = emeasure\ (measure\text{-}pmf\ p)\ A$
by(*auto intro*: *arg-cong2*[**where** *f*=*emeasure*])
finally show *?thesis* **.**
qed

lemma *integral-map-pmf*[*simp*]:
fixes $f :: 'a \Rightarrow 'b{::}\{banach,\ second\text{-}countable\text{-}topology\}$
shows $integral^L\ (map\text{-}pmf\ g\ p)\ f = integral^L\ p\ (\lambda x.\ f\ (g\ x))$
by (*simp add*: *integral-distr map-pmf-rep-eq*)

lemma *integrable-map-pmf-eq* [*simp*]:
fixes $g :: 'a \Rightarrow 'b{::}\{banach,\ second\text{-}countable\text{-}topology\}$
shows $integrable\ (map\text{-}pmf\ f\ p)\ g \longleftrightarrow integrable\ (measure\text{-}pmf\ p)\ (\lambda x.\ g\ (f\ x))$

by (*subst map-pmf-rep-eq*, *subst integrable-distr-eq*) *auto*

lemma *integrable-map-pmf* [*intro*]:
fixes $g :: 'a \Rightarrow 'b{::}\{banach,\ second\text{-}countable\text{-}topology\}$
shows $integrable\ (measure\text{-}pmf\ p)\ (\lambda x.\ g\ (f\ x)) \Longrightarrow integrable\ (map\text{-}pmf\ f\ p)\ g$
by (*subst integrable-map-pmf-eq*)

lemma *pmf-abs-summable* [*intro*]: *pmf p abs-summable-on A*
by (*rule abs-summable-on-subset*[*OF - subset-UNIV*])
(*auto simp*: *abs-summable-on-def integrable-iff-bounded nn-integral-pmf*)

lemma *measure-pmf-conv-infsetsum*: $measure\ (measure\text{-}pmf\ p)\ A = infsetsum\ (pmf\ p)\ A$
unfolding *infsetsum-def* **by** (*simp add*: *integral-eq-nn-integral nn-integral-pmf measure-def*)

lemma *infsetsum-pmf-eq-1*:
assumes $set\text{-}pmf\ p \subseteq A$
shows $infsetsum\ (pmf\ p)\ A = 1$
proof −
have $infsetsum\ (pmf\ p)\ A = lebesgue\text{-}integral\ (count\text{-}space\ UNIV)\ (pmf\ p)$

```
    using assms unfolding infsetsum-altdef set-lebesgue-integral-def
   by (intro Bochner-Integration.integral-cong) (auto simp: indicator-def set-pmf-eq)
  also have ... = 1
    by (subst integral-eq-nn-integral) (auto simp: nn-integral-pmf)
  finally show ?thesis .
qed

lemma map-return-pmf [simp]: map-pmf f (return-pmf x) = return-pmf (f x)
  by transfer (simp add: distr-return)

lemma map-pmf-const[simp]: map-pmf (λ-. c) M = return-pmf c
  by transfer (auto simp: prob-space.distr-const)

lemma pmf-return [simp]: pmf (return-pmf x) y = indicator {y} x
  by transfer (simp add: measure-return)

lemma nn-integral-return-pmf[simp]: 0 ≤ f x ⟹ (∫⁺x. f x ∂return-pmf x) = f x
  unfolding return-pmf.rep-eq by (intro nn-integral-return) auto

lemma emeasure-return-pmf[simp]: emeasure (return-pmf x) X = indicator X x
  unfolding return-pmf.rep-eq by (intro emeasure-return) auto

lemma measure-return-pmf [simp]: measure-pmf.prob (return-pmf x) A = indicator A x
proof −
  have ennreal (measure-pmf.prob (return-pmf x) A) =
        emeasure (measure-pmf (return-pmf x)) A
    by (simp add: measure-pmf.emeasure-eq-measure)
  also have ... = ennreal (indicator A x) by (simp add: ennreal-indicator)
  finally show ?thesis by simp
qed

lemma return-pmf-inj[simp]: return-pmf x = return-pmf y ⟷ x = y
  by (metis insertI1 set-return-pmf singletonD)

lemma map-pmf-eq-return-pmf-iff:
  map-pmf f p = return-pmf x ⟷ (∀ y ∈ set-pmf p. f y = x)
proof
  assume map-pmf f p = return-pmf x
  then have set-pmf (map-pmf f p) = set-pmf (return-pmf x) by simp
  then show ∀ y ∈ set-pmf p. f y = x by auto
next
  assume ∀ y ∈ set-pmf p. f y = x
  then show map-pmf f p = return-pmf x
    unfolding map-pmf-const[symmetric, of - p] by (intro map-pmf-cong) auto
qed

definition pair-pmf A B = bind-pmf A (λx. bind-pmf B (λy. return-pmf (x, y)))
```

lemma *pmf-pair*: *pmf* (*pair-pmf M N*) (*a*, *b*) = *pmf M a* ∗ *pmf N b*
unfolding *pair-pmf-def pmf-bind pmf-return*
apply (*subst integral-measure-pmf-real*[**where** *A*={*b*}])
apply (*auto simp*: *indicator-eq-0-iff*)
apply (*subst integral-measure-pmf-real*[**where** *A*={*a*}])
apply (*auto simp*: *indicator-eq-0-iff sum-nonneg-eq-0-iff pmf-nonneg*)
done

lemma *set-pair-pmf*[*simp*]: *set-pmf* (*pair-pmf A B*) = *set-pmf A* × *set-pmf B*
unfolding *pair-pmf-def set-bind-pmf set-return-pmf* **by** *auto*

lemma *measure-pmf-in-subprob-space*[*measurable* (*raw*)]:
measure-pmf M ∈ *space* (*subprob-algebra* (*count-space UNIV*))
by (*simp add*: *space-subprob-algebra*) *intro-locales*

lemma *nn-integral-pair-pmf'*: (∫⁺*x*. *f x* ∂*pair-pmf A B*) = (∫⁺*a*. ∫⁺*b*. *f* (*a*, *b*) ∂*B* ∂*A*)
proof −
have (∫⁺*x*. *f x* ∂*pair-pmf A B*) = (∫⁺*x*. *f x* ∗ *indicator* (*A* × *B*) *x* ∂*pair-pmf A B*)
by (*auto simp*: *AE-measure-pmf-iff intro*!: *nn-integral-cong-AE*)
also have ... = (∫⁺*a*. ∫⁺*b*. *f* (*a*, *b*) ∗ *indicator* (*A* × *B*) (*a*, *b*) ∂*B* ∂*A*)
by (*simp add*: *pair-pmf-def*)
also have ... = (∫⁺*a*. ∫⁺*b*. *f* (*a*, *b*) ∂*B* ∂*A*)
by (*auto intro*!: *nn-integral-cong-AE simp*: *AE-measure-pmf-iff*)
finally show *?thesis* **.**
qed

lemma *bind-pair-pmf*:
assumes *M*[*measurable*]: *M* ∈ *measurable* (*count-space UNIV* ⨂*M* *count-space UNIV*) (*subprob-algebra N*)
shows *measure-pmf* (*pair-pmf A B*) ⤜ *M* = (*measure-pmf A* ⤜ (λ*x*. *measure-pmf B* ⤜ (λ*y*. *M* (*x*, *y*))))
(**is** *?L* = *?R*)
proof (*rule measure-eqI*)
have *M'*[*measurable*]: *M* ∈ *measurable* (*pair-pmf A B*) (*subprob-algebra N*)
using *M*[*THEN measurable-space*] **by** (*simp-all add*: *space-pair-measure*)

note *measurable-bind*[**where** *N*=*count-space UNIV*, *measurable*]
note *measure-pmf-in-subprob-space*[*simp*]

have *sets-eq-N*: *sets ?L* = *N*
by (*subst sets-bind*[*OF sets-kernel*[*OF M'*]]) *auto*
show *sets ?L* = *sets ?R*
using *measurable-space*[*OF M*]
by (*simp add*: *sets-eq-N space-pair-measure space-subprob-algebra*)
fix *X* **assume** *X* ∈ *sets ?L*
then have *X*[*measurable*]: *X* ∈ *sets N*
unfolding *sets-eq-N* **.**

then show *emeasure ?L X = emeasure ?R X*
apply (*simp add*: *emeasure-bind*[*OF - M' X*])
apply (*simp add*: *nn-integral-bind*[**where** *B*=*count-space UNIV*] *pair-pmf-def measure-pmf-bind*[*of A*]
nn-integral-measure-pmf-finite)
apply (*subst emeasure-bind*[*OF - - X*])
apply *measurable*
apply (*subst emeasure-bind*[*OF - - X*])
apply *measurable*
done
qed

lemma *map-fst-pair-pmf*: *map-pmf fst* (*pair-pmf A B*) = *A*
by (*simp add*: *pair-pmf-def map-pmf-def bind-assoc-pmf bind-return-pmf bind-return-pmf'*)

lemma *map-snd-pair-pmf*: *map-pmf snd* (*pair-pmf A B*) = *B*
by (*simp add*: *pair-pmf-def map-pmf-def bind-assoc-pmf bind-return-pmf bind-return-pmf'*)

lemma *nn-integral-pmf'*:
inj-on f A ⟹ (∫⁺*x*. *pmf p* (*f x*) ∂*count-space A*) = *emeasure p* (*f* ‘ *A*)
by (*subst nn-integral-bij-count-space*[**where** *g*=*f* **and** *B*=*f*‘*A*])
(*auto simp*: *bij-betw-def nn-integral-pmf*)

lemma *pmf-le-0-iff*[*simp*]: *pmf M p* ≤ *0* ⟷ *pmf M p* = *0*
using *pmf-nonneg*[*of M p*] **by** *arith*

lemma *min-pmf-0*[*simp*]: *min* (*pmf M p*) *0* = *0 min 0* (*pmf M p*) = *0*
using *pmf-nonneg*[*of M p*] **by** *arith+*

lemma *pmf-eq-0-set-pmf*: *pmf M p* = *0* ⟷ *p* ∉ *set-pmf M*
unfolding *set-pmf-iff* **by** *simp*

lemma *pmf-map-inj*: *inj-on f* (*set-pmf M*) ⟹ *x* ∈ *set-pmf M* ⟹ *pmf* (*map-pmf f M*) (*f x*) = *pmf M x*
by (*auto simp*: *pmf.rep-eq map-pmf-rep-eq measure-distr AE-measure-pmf-iff inj-onD*
intro!: *measure-pmf.finite-measure-eq-AE*)

lemma *pair-return-pmf* [*simp*]: *pair-pmf* (*return-pmf x*) (*return-pmf y*) = *return-pmf* (*x*, *y*)
by (*auto simp*: *pair-pmf-def bind-return-pmf*)

lemma *pmf-map-inj'*: *inj f* ⟹ *pmf* (*map-pmf f M*) (*f x*) = *pmf M x*
apply(*cases x* ∈ *set-pmf M*)
apply(*simp add*: *pmf-map-inj*[*OF subset-inj-on*])
apply(*simp add*: *pmf-eq-0-set-pmf*[*symmetric*])
apply(*auto simp add*: *pmf-eq-0-set-pmf dest*: *injD*)
done

lemma *expectation-pair-pmf-fst* [*simp*]:
fixes $f :: 'a \Rightarrow 'b::\{banach, second\text{-}countable\text{-}topology\}$
shows *measure-pmf.expectation* (*pair-pmf p q*) ($\lambda x.$ *f* (*fst x*)) = *measure-pmf.expectation p f*
proof −
have *measure-pmf.expectation* (*pair-pmf p q*) ($\lambda x.$ *f* (*fst x*)) = *measure-pmf.expectation* (*map-pmf fst* (*pair-pmf p q*)) *f* **by** *simp*
also have *map-pmf fst* (*pair-pmf p q*) = *p*
by (*simp add*: *map-fst-pair-pmf*)
finally show *?thesis* **.**
qed

lemma *expectation-pair-pmf-snd* [*simp*]:
fixes $f :: 'a \Rightarrow 'b::\{banach, second\text{-}countable\text{-}topology\}$
shows *measure-pmf.expectation* (*pair-pmf p q*) ($\lambda x.$ *f* (*snd x*)) = *measure-pmf.expectation q f*
proof −
have *measure-pmf.expectation* (*pair-pmf p q*) ($\lambda x.$ *f* (*snd x*)) = *measure-pmf.expectation* (*map-pmf snd* (*pair-pmf p q*)) *f* **by** *simp*
also have *map-pmf snd* (*pair-pmf p q*) = *q*
by (*simp add*: *map-snd-pair-pmf*)
finally show *?thesis* **.**
qed

lemma *pmf-map-outside*: $x \notin f$ ‘ *set-pmf M* $\Longrightarrow$ *pmf* (*map-pmf f M*) *x* = *0*
unfolding *pmf-eq-0-set-pmf* **by** *simp*

lemma *measurable-set-pmf*[*measurable*]: *Measurable.pred* (*count-space UNIV*) ($\lambda x.$ $x \in$ *set-pmf M*)
by *simp*

20.3 PMFs as function

context
fixes $f :: 'a \Rightarrow real$
assumes *nonneg*: $\bigwedge x.\ 0 \le f\ x$
assumes *prob*: $(\int^+ x.\ f\ x\ \partial count\text{-}space\ UNIV) = 1$
begin

lift-definition *embed-pmf* :: *'a pmf* **is** *density* (*count-space UNIV*) (*ennreal* ∘ *f*)
proof (*intro conjI*)
have ∗[*simp*]: $\bigwedge x\ y.$ *ennreal* (*f y*) ∗ *indicator* {*x*} *y* = *ennreal* (*f x*) ∗ *indicator* {*x*} *y*
by (*simp split*: *split-indicator*)
show *AE x in density* (*count-space UNIV*) (*ennreal* ∘ *f*).
measure (*density* (*count-space UNIV*) (*ennreal* ∘ *f*)) {*x*} ≠ *0*
by (*simp add*: *AE-density nonneg measure-def emeasure-density max-def*)
show *prob-space* (*density* (*count-space UNIV*) (*ennreal* ∘ *f*))
by *standard* (*simp add*: *emeasure-density prob*)

qed *simp*

lemma *pmf-embed-pmf*: *pmf embed-pmf x = f x*
proof *transfer*
have ∗[*simp*]: ⋀*x y. ennreal* (*f y*) ∗ *indicator* {*x*} *y* = *ennreal* (*f x*) ∗ *indicator* {*x*} *y*
by (*simp split*: *split-indicator*)
fix *x* **show** *measure* (*density* (*count-space UNIV*) (*ennreal* ∘ *f*)) {*x*} = *f x*
by *transfer* (*simp add*: *measure-def emeasure-density nonneg max-def*)
qed

lemma *set-embed-pmf*: *set-pmf embed-pmf* = {*x*. *f x* ≠ *0*}
by(*auto simp add*: *set-pmf-eq pmf-embed-pmf*)

end

lemma *embed-pmf-transfer*:
rel-fun (*eq-onp* (λ*f*. (∀ *x*. *0* ≤ *f x*) ∧ (∫ ⁺*x*. *ennreal* (*f x*) ∂*count-space UNIV*) = *1*)) *pmf-as-measure.cr-pmf* (λ*f*. *density* (*count-space UNIV*) (*ennreal* ∘ *f*)) *embed-pmf*
by (*auto simp*: *rel-fun-def eq-onp-def embed-pmf.transfer*)

lemma *measure-pmf-eq-density*: *measure-pmf p* = *density* (*count-space UNIV*) (*pmf p*)
proof (*transfer*, *elim conjE*)
fix *M* :: *'a measure* **assume** [*simp*]: *sets M* = *UNIV* **and** *ae*: *AE x in M*. *measure M* {*x*} ≠ *0*
assume *prob-space M* **then interpret** *prob-space M* **.**
show *M* = *density* (*count-space UNIV*) (λ*x*. *ennreal* (*measure M* {*x*}))
proof (*rule measure-eqI*)
fix *A* :: *'a set*
have (∫ ⁺ *x*. *ennreal* (*measure M* {*x*}) ∗ *indicator A x* ∂*count-space UNIV*) =
(∫ ⁺ *x*. *emeasure M* {*x*} ∗ *indicator* (*A* ∩ {*x*. *measure M* {*x*} ≠ *0*}) *x* ∂*count-space UNIV*)
by (*auto intro*!: *nn-integral-cong simp*: *emeasure-eq-measure split*: *split-indicator*)
also have ... = (∫ ⁺ *x*. *emeasure M* {*x*} ∂*count-space* (*A* ∩ {*x*. *measure M* {*x*} ≠ *0*}))
by (*subst nn-integral-restrict-space*[*symmetric*]) (*auto simp*: *restrict-count-space*)
also have ... = *emeasure M* (⋃*x*∈(*A* ∩ {*x*. *measure M* {*x*} ≠ *0*}). {*x*})
by (*intro emeasure-UN-countable*[*symmetric*] *countable-Int2 countable-support*)
(*auto simp*: *disjoint-family-on-def*)
also have ... = *emeasure M A*
using *ae* **by** (*intro emeasure-eq-AE*) *auto*
finally show *emeasure M A* = *emeasure* (*density* (*count-space UNIV*) (λ*x*. *ennreal* (*measure M* {*x*}))) *A*
using *emeasure-space-1* **by** (*simp add*: *emeasure-density*)
qed *simp*
qed

lemma *td-pmf-embed-pmf*:
type-definition pmf embed-pmf {*f*::'*a* ⇒ *real*. (∀ *x*. *0* ≤ *f x*) ∧ (∫⁺*x*. *ennreal* (*f x*) ∂*count-space UNIV*) = *1*}
unfolding *type-definition-def*
proof *safe*
fix *p* :: '*a pmf*
have (∫⁺ *x*. *1* ∂*measure-pmf p*) = *1*
using *measure-pmf.emeasure-space-1*[*of p*] **by** *simp*
then show ∗: (∫⁺ *x*. *ennreal* (*pmf p x*) ∂*count-space UNIV*) = *1*
by (*simp add*: *measure-pmf-eq-density nn-integral-density pmf-nonneg del*: *nn-integral-const*)

show *embed-pmf* (*pmf p*) = *p*
by (*intro measure-pmf-inject*[*THEN iffD1*])
(*simp add*: ∗ *embed-pmf.rep-eq pmf-nonneg measure-pmf-eq-density*[*of p*] *comp-def*)
next
fix *f* :: '*a* ⇒ *real* **assume** ∀ *x*. *0* ≤ *f x* (∫⁺*x*. *f x* ∂*count-space UNIV*) = *1*
then show *pmf* (*embed-pmf f*) = *f*
by (*auto intro*!: *pmf-embed-pmf*)
qed (*rule pmf-nonneg*)

end

lemma *nn-integral-measure-pmf*: (∫⁺ *x*. *f x* ∂*measure-pmf p*) = ∫⁺ *x*. *ennreal* (*pmf p x*) ∗ *f x* ∂*count-space UNIV*
by(*simp add*: *measure-pmf-eq-density nn-integral-density pmf-nonneg*)

lemma *integral-measure-pmf*:
fixes *f* :: '*a* ⇒ '*b*::{*banach*, *second-countable-topology*}
assumes *A*: *finite A*
shows (⋀*a*. *a* ∈ *set-pmf M* ⟹ *f a* ≠ *0* ⟹ *a* ∈ *A*) ⟹ (*LINT x*|*M*. *f x*) = (∑ *a*∈*A*. *pmf M a* ∗$_R$ *f a*)
unfolding *measure-pmf-eq-density*
apply (*simp add*: *integral-density*)
apply (*subst lebesgue-integral-count-space-finite-support*)
apply (*auto intro*!: *finite-subset*[*OF* - ‹*finite A*›] *sum.mono-neutral-left simp*: *pmf-eq-0-set-pmf*)
done

lemma *expectation-return-pmf* [*simp*]:
fixes *f* :: '*a* ⇒ '*b*::{*banach*, *second-countable-topology*}
shows *measure-pmf.expectation* (*return-pmf x*) *f* = *f x*
by (*subst integral-measure-pmf*[*of* {*x*}]) *simp-all*

lemma *pmf-expectation-bind*:
fixes *p* :: '*a pmf* **and** *f* :: '*a* ⇒ '*b pmf*
and *h* :: '*b* ⇒ '*c*::{*banach*, *second-countable-topology*}
assumes *finite A* ⋀*x*. *x* ∈ *A* ⟹ *finite* (*set-pmf* (*f x*)) *set-pmf p* ⊆ *A*
shows *measure-pmf.expectation* (*p* ⤜ *f*) *h* =

```
        (∑ a∈A. pmf p a *R measure-pmf.expectation (f a) h)
proof −
  have measure-pmf.expectation (p ⤜ f) h = (∑ a∈(⋃ x∈A. set-pmf (f x)). pmf
(p ⤜ f) a *R h a)
    using assms by (intro integral-measure-pmf) auto
  also have ... = (∑ x∈(⋃ x∈A. set-pmf (f x)). (∑ a∈A. (pmf p a * pmf (f a)
x) *R h x))
  proof (intro sum.cong refl, goal-cases)
    case (1 x)
    thus ?case
      by (subst pmf-bind, subst integral-measure-pmf[of A])
         (insert assms, auto simp: scaleR-sum-left)
  qed
  also have ... = (∑ j∈A. pmf p j *R (∑ i∈(⋃ x∈A. set-pmf (f x)). pmf (f j) i
*R h i))
    by (subst sum.swap) (simp add: scaleR-sum-right)
  also have ... = (∑ j∈A. pmf p j *R measure-pmf.expectation (f j) h)
  proof (intro sum.cong refl, goal-cases)
    case (1 x)
    thus ?case
      by (subst integral-measure-pmf[of (⋃ x∈A. set-pmf (f x))])
         (insert assms, auto simp: scaleR-sum-left)
  qed
  finally show ?thesis .
qed

lemma continuous-on-LINT-pmf: — This is dominated convergence!?
  fixes f :: 'i ⇒ 'a::topological-space ⇒ 'b::{banach, second-countable-topology}
  assumes f: ⋀i. i ∈ set-pmf M ⟹ continuous-on A (f i)
    and bnd: ⋀a i. a ∈ A ⟹ i ∈ set-pmf M ⟹ norm (f i a) ≤ B
  shows continuous-on A (λa. LINT i|M. f i a)
proof cases
  assume finite M with f show ?thesis
    using integral-measure-pmf[OF ‹finite M›]
    by (subst integral-measure-pmf[OF ‹finite M›])
       (auto intro!: continuous-on-sum continuous-on-scaleR continuous-on-const)
next
  assume infinite M
  let ?f = λi x. pmf (map-pmf (to-nat-on M) M) i *R f (from-nat-into M i) x

  show ?thesis
  proof (rule uniform-limit-theorem)
    show ∀F n in sequentially. continuous-on A (λa. ∑ i<n. ?f i a)
      by (intro always-eventually allI continuous-on-sum continuous-on-scaleR con-
tinuous-on-const f
                from-nat-into set-pmf-not-empty)
    show uniform-limit A (λn a. ∑ i<n. ?f i a) (λa. LINT i|M. f i a) sequentially
    proof (subst uniform-limit-cong[where g=λn a. ∑ i<n. ?f i a])
      fix a assume a ∈ A
```

have *1*: (*LINT i|M. f i a*) = (*LINT i|map-pmf* (*to-nat-on M*) *M. f* (*from-nat-into M i*) *a*)
by (*auto intro*!: *integral-cong-AE AE-pmfI*)
have *2*: ... = (*LINT i|count-space UNIV. pmf* (*map-pmf* (*to-nat-on M*) *M*) *i* $*_R$ *f* (*from-nat-into M i*) *a*)
by (*simp add*: *measure-pmf-eq-density integral-density*)
have (*λn. ?f n a*) *sums* (*LINT i|M. f i a*)
unfolding *1 2*
proof (*intro sums-integral-count-space-nat*)
have *A*: *integrable M* (*λi. f i a*)
using ‹*a*∈*A*› **by** (*auto intro*!: *measure-pmf.integrable-const-bound AE-pmfI bnd*)
have *integrable* (*map-pmf* (*to-nat-on M*) *M*) (*λi. f* (*from-nat-into M i*) *a*)
by (*auto simp add*: *map-pmf-rep-eq integrable-distr-eq intro*!: *AE-pmfI integrable-cong-AE-imp*[*OF A*])
then show *integrable* (*count-space UNIV*) (*λn. ?f n a*)
by (*simp add*: *measure-pmf-eq-density integrable-density*)
qed
then show (*LINT i|M. f i a*) = (∑ *n. ?f n a*)
by (*simp add*: *sums-unique*)
next
show *uniform-limit A* (*λn a.* ∑ *i*<*n. ?f i a*) (*λa.* (∑ *n. ?f n a*)) *sequentially*
proof (*rule Weierstrass-m-test*)
fix *n a* **assume** *a*∈*A*
then show *norm* (*?f n a*) ≤ *pmf* (*map-pmf* (*to-nat-on M*) *M*) *n* ∗ *B*
using *bnd* **by** (*auto intro*!: *mult-mono simp*: *from-nat-into set-pmf-not-empty*)
next
have *integrable* (*map-pmf* (*to-nat-on M*) *M*) (*λn. B*)
by *auto*
then show *summable* (*λn. pmf* (*map-pmf* (*to-nat-on* (*set-pmf M*)) *M*) *n* ∗ *B*)
by (*fastforce simp add*: *measure-pmf-eq-density integrable-density integrable-count-space-nat-iff summable-mult2*)
qed
qed *simp*
qed *simp*
qed

lemma *continuous-on-LBINT*:
fixes *f* :: *real* ⇒ *real*
assumes *f*: ⋀*b. a* ≤ *b* ⟹ *set-integrable lborel* {*a*..*b*} *f*
shows *continuous-on UNIV* (*λb. LBINT x*:{*a*..*b*}. *f x*)
proof (*subst set-borel-integral-eq-integral*)
{ fix *b* :: *real* **assume** *a* ≤ *b*
from *f*[*OF this*] **have** *continuous-on* {*a*..*b*} (*λb. integral* {*a*..*b*} *f*)
by (*intro indefinite-integral-continuous-1 set-borel-integral-eq-integral*) **}**
note ∗ = *this*

have *continuous-on* (⋃ *b*∈{*a*..}. {*a* <..< *b*}) (*λb. integral* {*a*..*b*} *f*)

proof (*intro continuous-on-open-UN*)
show $b \in \{a..\} \Longrightarrow$ *continuous-on* $\{a<..<b\}$ $(\lambda b.$ *integral* $\{a..b\}$ $f)$ **for** b
using $*[of\ b]$ **by** (*rule continuous-on-subset*) *auto*
qed *simp*
also have $(\bigcup b \in \{a..\}.\ \{a <..< b\}) = \{a <..\}$
by (*auto simp*: *lt-ex gt-ex less-imp-le*) (*simp add*: *Bex-def less-imp-le gt-ex cong*: *rev-conj-cong*)
finally have *continuous-on* $\{a+1\ ..\}$ $(\lambda b.$ *integral* $\{a..b\}$ $f)$
by (*rule continuous-on-subset*) *auto*
moreover have *continuous-on* $\{a..a+1\}$ $(\lambda b.$ *integral* $\{a..b\}$ $f)$
by (*rule* $*$) *simp*
moreover
have $x \leq a \Longrightarrow \{a..x\} = (if\ a = x\ then\ \{a\}\ else\ \{\})$ **for** x
by *auto*
then have *continuous-on* $\{..a\}$ $(\lambda b.$ *integral* $\{a..b\}$ $f)$
by (*subst continuous-on-cong*[*OF refl*, **where** $g=\lambda x.\ 0$]) (*auto intro*!: *continuous-on-const*)
ultimately have *continuous-on* $(\{..a\} \cup \{a..a+1\} \cup \{a+1\ ..\})$ $(\lambda b.$ *integral* $\{a..b\}$ $f)$
by (*intro continuous-on-closed-Un*) *auto*
also have $\{..a\} \cup \{a..a+1\} \cup \{a+1\ ..\} = UNIV$
by *auto*
finally show *continuous-on UNIV* $(\lambda b.$ *integral* $\{a..b\}$ $f)$
by *auto*
next
show *set-integrable lborel* $\{a..b\}$ f **for** b
using f **by** (*cases* $a \leq b$) *auto*
qed

locale *pmf-as-function*
begin

setup-lifting *td-pmf-embed-pmf*

lemma *set-pmf-transfer*[*transfer-rule*]:
assumes *bi-total A*
shows *rel-fun* (*pcr-pmf A*) (*rel-set A*) $(\lambda f.\ \{x.\ f\ x \neq 0\})$ *set-pmf*
using ‹*bi-total A*›
by (*auto simp*: *pcr-pmf-def cr-pmf-def rel-fun-def rel-set-def bi-total-def Bex-def set-pmf-iff*)
metis+

end

context
begin

interpretation *pmf-as-function* **.**

lemma *pmf-eqI*: $(\bigwedge i.\ pmf\ M\ i = pmf\ N\ i) \Longrightarrow M = N$
by *transfer auto*

lemma *pmf-eq-iff*: $M = N \longleftrightarrow (\forall\, i.\ pmf\ M\ i = pmf\ N\ i)$
by (*auto intro*: *pmf-eqI*)

lemma *pmf-neq-exists-less*:
assumes $M \neq N$
shows $\exists\, x.\ pmf\ M\ x < pmf\ N\ x$
proof (*rule ccontr*)
assume $\neg(\exists\, x.\ pmf\ M\ x < pmf\ N\ x)$
hence *ge*: $pmf\ M\ x \geq pmf\ N\ x$ **for** x **by** (*auto simp*: *not-less*)
from *assms* **obtain** x **where** $pmf\ M\ x \neq pmf\ N\ x$ **by** (*auto simp*: *pmf-eq-iff*)
with *ge*[*of x*] **have** *gt*: $pmf\ M\ x > pmf\ N\ x$ **by** *simp*
have $1 = measure\ (measure\text{-}pmf\ M)\ UNIV$ **by** *simp*
also have $\ldots = measure\ (measure\text{-}pmf\ N)\ \{x\} + measure\ (measure\text{-}pmf\ N)\ (UNIV - \{x\})$
by (*subst measure-pmf.finite-measure-Union* [*symmetric*]) *simp-all*
also from *gt* **have** $measure\ (measure\text{-}pmf\ N)\ \{x\} < measure\ (measure\text{-}pmf\ M)\ \{x\}$
by (*simp add*: *measure-pmf-single*)
also have $measure\ (measure\text{-}pmf\ N)\ (UNIV - \{x\}) \leq measure\ (measure\text{-}pmf\ M)\ (UNIV - \{x\})$
by (*subst* (*1 2*) *integral-pmf* [*symmetric*])
(*intro integral-mono integrable-pmf*, *simp-all add*: *ge*)
also have $measure\ (measure\text{-}pmf\ M)\ \{x\} + \ldots = 1$
by (*subst measure-pmf.finite-measure-Union* [*symmetric*]) *simp-all*
finally show *False* **by** *simp-all*
qed

lemma *bind-commute-pmf*: $bind\text{-}pmf\ A\ (\lambda x.\ bind\text{-}pmf\ B\ (C\ x)) = bind\text{-}pmf\ B\ (\lambda y.\ bind\text{-}pmf\ A\ (\lambda x.\ C\ x\ y))$
unfolding *pmf-eq-iff pmf-bind*
proof
fix i
interpret B: *prob-space restrict-space B B*
by (*intro prob-space-restrict-space measure-pmf.emeasure-eq-1-AE*)
(*auto simp*: *AE-measure-pmf-iff*)
interpret A: *prob-space restrict-space A A*
by (*intro prob-space-restrict-space measure-pmf.emeasure-eq-1-AE*)
(*auto simp*: *AE-measure-pmf-iff*)

interpret AB: *pair-prob-space restrict-space A A restrict-space B B*
by *unfold-locales*

have $(\int\ x.\ \int\ y.\ pmf\ (C\ x\ y)\ i\ \partial B\ \partial A) = (\int\ x.\ (\int\ y.\ pmf\ (C\ x\ y)\ i\ \partial restrict\text{-}space\ B\ B)\ \partial A)$
by (*rule Bochner-Integration.integral-cong*) (*auto intro*!: *integral-pmf-restrict*)
also have $\ldots = (\int\ x.\ (\int\ y.\ pmf\ (C\ x\ y)\ i\ \partial restrict\text{-}space\ B\ B)\ \partial restrict\text{-}space$

A A)
by (*intro integral-pmf-restrict B.borel-measurable-lebesgue-integral measurable-pair-restrict-pmf2*
countable-set-pmf borel-measurable-count-space)
also have ... = (∫ *y*. ∫ *x*. *pmf* (*C x y*) *i* ∂*restrict-space A A* ∂*restrict-space B*
B)
by (*rule AB.Fubini-integral*[*symmetric*])
(*auto intro*!: *AB.integrable-const-bound*[**where** *B=1*] *measurable-pair-restrict-pmf2*
simp: *pmf-nonneg pmf-le-1 measurable-restrict-space1*)
also have ... = (∫ *y*. ∫ *x*. *pmf* (*C x y*) *i* ∂*restrict-space A A* ∂*B*)
by (*intro integral-pmf-restrict*[*symmetric*] *A.borel-measurable-lebesgue-integral*
measurable-pair-restrict-pmf2
countable-set-pmf borel-measurable-count-space)
also have ... = (∫ *y*. ∫ *x*. *pmf* (*C x y*) *i* ∂*A* ∂*B*)
by (*rule Bochner-Integration.integral-cong*) (*auto intro*!: *integral-pmf-restrict*[*symmetric*])
finally show (∫ *x*. ∫ *y*. *pmf* (*C x y*) *i* ∂*B* ∂*A*) = (∫ *y*. ∫ *x*. *pmf* (*C x y*) *i* ∂*A*
∂*B*) **.**
qed

lemma *pair-map-pmf1*: *pair-pmf* (*map-pmf f A*) *B* = *map-pmf* (*apfst f*) (*pair-pmf*
A B)
proof (*safe intro*!: *pmf-eqI*)
fix *a* :: *'a* **and** *b* :: *'b*
have [*simp*]: ⋀*c d*. *indicator* (*apfst f* −‘ {(*a*, *b*)}) (*c*, *d*) = *indicator* (*f* −‘ {*a*})
c ∗ (*indicator* {*b*} *d*::*ennreal*)
by (*auto split*: *split-indicator*)

have *ennreal* (*pmf* (*pair-pmf* (*map-pmf f A*) *B*) (*a*, *b*)) =
ennreal (*pmf* (*map-pmf* (*apfst f*) (*pair-pmf A B*)) (*a*, *b*))
unfolding *pmf-pair ennreal-pmf-map*
by (*simp add*: *nn-integral-pair-pmf' max-def emeasure-pmf-single nn-integral-multc*
pmf-nonneg
emeasure-map-pmf[*symmetric*] *ennreal-mult del*: *emeasure-map-pmf*)
then show *pmf* (*pair-pmf* (*map-pmf f A*) *B*) (*a*, *b*) = *pmf* (*map-pmf* (*apfst f*)
(*pair-pmf A B*)) (*a*, *b*)
by (*simp add*: *pmf-nonneg*)
qed

lemma *pair-map-pmf2*: *pair-pmf A* (*map-pmf f B*) = *map-pmf* (*apsnd f*) (*pair-pmf*
A B)
proof (*safe intro*!: *pmf-eqI*)
fix *a* :: *'a* **and** *b* :: *'b*
have [*simp*]: ⋀*c d*. *indicator* (*apsnd f* −‘ {(*a*, *b*)}) (*c*, *d*) = *indicator* {*a*} *c* ∗
(*indicator* (*f* −‘ {*b*}) *d*::*ennreal*)
by (*auto split*: *split-indicator*)

have *ennreal* (*pmf* (*pair-pmf A* (*map-pmf f B*)) (*a*, *b*)) =
ennreal (*pmf* (*map-pmf* (*apsnd f*) (*pair-pmf A B*)) (*a*, *b*))
unfolding *pmf-pair ennreal-pmf-map*
by (*simp add*: *nn-integral-pair-pmf' max-def emeasure-pmf-single nn-integral-cmult*

nn-integral-multc pmf-nonneg
emeasure-map-pmf[*symmetric*] *ennreal-mult del*: *emeasure-map-pmf*)
then show *pmf* (*pair-pmf A* (*map-pmf f B*)) (*a*, *b*) = *pmf* (*map-pmf* (*apsnd f*) (*pair-pmf A B*)) (*a*, *b*)
by (*simp add*: *pmf-nonneg*)
qed

lemma *map-pair*: *map-pmf* (λ(*a*, *b*). (*f a*, *g b*)) (*pair-pmf A B*) = *pair-pmf* (*map-pmf f A*) (*map-pmf g B*)
by (*simp add*: *pair-map-pmf2 pair-map-pmf1 map-pmf-comp split-beta'*)

end

lemma *pair-return-pmf1*: *pair-pmf* (*return-pmf x*) *y* = *map-pmf* (*Pair x*) *y*
by(*simp add*: *pair-pmf-def bind-return-pmf map-pmf-def*)

lemma *pair-return-pmf2*: *pair-pmf x* (*return-pmf y*) = *map-pmf* (λ*x*. (*x*, *y*)) *x*
by(*simp add*: *pair-pmf-def bind-return-pmf map-pmf-def*)

lemma *pair-pair-pmf*: *pair-pmf* (*pair-pmf u v*) *w* = *map-pmf* (λ(*x*, (*y*, *z*)). ((*x*, *y*), *z*)) (*pair-pmf u* (*pair-pmf v w*))
by(*simp add*: *pair-pmf-def bind-return-pmf map-pmf-def bind-assoc-pmf*)

lemma *pair-commute-pmf*: *pair-pmf x y* = *map-pmf* (λ(*x*, *y*). (*y*, *x*)) (*pair-pmf y x*)
unfolding *pair-pmf-def* **by**(*subst bind-commute-pmf*)(*simp add*: *map-pmf-def bind-assoc-pmf bind-return-pmf*)

lemma *set-pmf-subset-singleton*: *set-pmf p* ⊆ {*x*} ⟷ *p* = *return-pmf x*
proof(*intro iffI pmf-eqI*)
fix *i*
assume *x*: *set-pmf p* ⊆ {*x*}
hence ∗: *set-pmf p* = {*x*} **using** *set-pmf-not-empty*[*of p*] **by** *auto*
have *ennreal* (*pmf p x*) = ∫⁺ *i*. *indicator* {*x*} *i* ∂*p* **by**(*simp add*: *emeasure-pmf-single*)
also have ... = ∫⁺ *i*. *1* ∂*p* **by**(*rule nn-integral-cong-AE*)(*simp add*: *AE-measure-pmf-iff* ∗)
also have ... = *1* **by** *simp*
finally show *pmf p i* = *pmf* (*return-pmf x*) *i* **using** *x*
by(*auto split*: *split-indicator simp add*: *pmf-eq-0-set-pmf*)
qed *auto*

lemma *bind-eq-return-pmf*:
bind-pmf p f = *return-pmf x* ⟷ (∀ *y*∈*set-pmf p*. *f y* = *return-pmf x*)
(**is** *?lhs* ⟷ *?rhs*)
proof(*intro iffI strip*)
fix *y*
assume *y*: *y* ∈ *set-pmf p*
assume *?lhs*
hence *set-pmf* (*bind-pmf p f*) = {*x*} **by** *simp*

hence $(\bigcup y{\in}$*set-pmf p. set-pmf* $(f\ y)) = \{x\}$ **by** *simp*
hence *set-pmf* $(f\ y) \subseteq \{x\}$ **using** *y* **by** *auto*
thus $f\ y =$ *return-pmf x* **by**(*simp add*: *set-pmf-subset-singleton*)
next
assume $*$: *?rhs*
show *?lhs*
proof(*rule pmf-eqI*)
fix *i*
have *ennreal* (*pmf* (*bind-pmf p f*) *i*) $= \int^+ y.$ *ennreal* (*pmf* (*f y*) *i*) ∂p
by (*simp add*: *ennreal-pmf-bind*)
also have $\ldots = \int^+ y.$ *ennreal* (*pmf* (*return-pmf x*) *i*) ∂p
by(*rule nn-integral-cong-AE*)(*simp add*: *AE-measure-pmf-iff* $*$)
also have $\ldots =$ *ennreal* (*pmf* (*return-pmf x*) *i*)
by *simp*
finally show *pmf* (*bind-pmf p f*) *i* = *pmf* (*return-pmf x*) *i*
by (*simp add*: *pmf-nonneg*)
qed
qed

lemma *pmf-False-conv-True*: *pmf p False* = *1* − *pmf p True*
proof −
have *pmf p False* + *pmf p True* = *measure p* {*False*} + *measure p* {*True*}
by(*simp add*: *measure-pmf-single*)
also have $\ldots =$ *measure p* ({*False*} ∪ {*True*})
by(*subst measure-pmf.finite-measure-Union*) *simp-all*
also have {*False*} ∪ {*True*} = *space p* **by** *auto*
finally show *?thesis* **by** *simp*
qed

lemma *pmf-True-conv-False*: *pmf p True* = *1* − *pmf p False*
by(*simp add*: *pmf-False-conv-True*)

20.4 Conditional Probabilities

lemma *measure-pmf-zero-iff*: *measure* (*measure-pmf p*) *s* = *0* ⟷ *set-pmf p* ∩ *s* = {}
by (*subst measure-pmf.prob-eq-0*) (*auto simp*: *AE-measure-pmf-iff*)

context
fixes *p* :: *'a pmf* **and** *s* :: *'a set*
assumes *not-empty*: *set-pmf p* ∩ *s* ≠ {}
begin

interpretation *pmf-as-measure* **.**

lemma *emeasure-measure-pmf-not-zero*: *emeasure* (*measure-pmf p*) *s* ≠ *0*
proof
assume *emeasure* (*measure-pmf p*) *s* = *0*
then have *AE x in measure-pmf p. x* ∉ *s*

by (*rule AE-I*[*rotated*]) *auto*
with *not-empty* **show** *False*
by (*auto simp*: *AE-measure-pmf-iff*)
qed

lemma *measure-measure-pmf-not-zero*: *measure* (*measure-pmf p*) *s* ≠ *0*
using *emeasure-measure-pmf-not-zero* **by** (*simp add*: *measure-pmf.emeasure-eq-measure measure-nonneg*)

lift-definition *cond-pmf* :: *'a pmf* **is**
uniform-measure (*measure-pmf p*) *s*
proof (*intro conjI*)
show *prob-space* (*uniform-measure* (*measure-pmf p*) *s*)
by (*intro prob-space-uniform-measure*) (*auto simp*: *emeasure-measure-pmf-not-zero*)
show *AE x in uniform-measure* (*measure-pmf p*) *s*. *measure* (*uniform-measure* (*measure-pmf p*) *s*) {*x*} ≠ *0*
by (*simp add*: *emeasure-measure-pmf-not-zero measure-measure-pmf-not-zero AE-uniform-measure*
AE-measure-pmf-iff set-pmf.rep-eq less-top[*symmetric*])
qed *simp*

lemma *pmf-cond*: *pmf cond-pmf x* = (*if x* ∈ *s then pmf p x* / *measure p s else 0*)
by *transfer* (*simp add*: *emeasure-measure-pmf-not-zero pmf.rep-eq*)

lemma *set-cond-pmf*[*simp*]: *set-pmf cond-pmf* = *set-pmf p* ∩ *s*
by (*auto simp add*: *set-pmf-iff pmf-cond measure-measure-pmf-not-zero split*: *if-split-asm*)

end

lemma *measure-pmf-posI*: *x* ∈ *set-pmf p* ⟹ *x* ∈ *A* ⟹ *measure-pmf.prob p A* > *0*
using *measure-measure-pmf-not-zero*[*of p A*] **by** (*subst zero-less-measure-iff*) *blast*

lemma *cond-map-pmf*:
assumes *set-pmf p* ∩ *f* −‘ *s* ≠ {}
shows *cond-pmf* (*map-pmf f p*) *s* = *map-pmf f* (*cond-pmf p* (*f* −‘ *s*))
proof −
have ∗: *set-pmf* (*map-pmf f p*) ∩ *s* ≠ {}
using *assms* **by** *auto*
{ fix *x*
have *ennreal* (*pmf* (*map-pmf f* (*cond-pmf p* (*f* −‘ *s*))) *x*) =
emeasure p (*f* −‘ *s* ∩ *f* −‘ {*x*}) / *emeasure p* (*f* −‘ *s*)
unfolding *ennreal-pmf-map cond-pmf.rep-eq*[*OF assms*] **by** (*simp add*: *nn-integral-uniform-measure*)
also have *f* −‘ *s* ∩ *f* −‘ {*x*} = (*if x* ∈ *s then f* −‘ {*x*} *else* {})
by *auto*
also have *emeasure p* (*if x* ∈ *s then f* −‘ {*x*} *else* {}) / *emeasure p* (*f* −‘ *s*) =
ennreal (*pmf* (*cond-pmf* (*map-pmf f p*) *s*) *x*)
using *measure-measure-pmf-not-zero*[*OF* ∗]

by (*simp add*: *pmf-cond*[*OF* ∗] *ennreal-pmf-map measure-pmf.emeasure-eq-measure divide-ennreal pmf-nonneg measure-nonneg zero-less-measure-iff pmf-map*)
finally have *ennreal* (*pmf* (*cond-pmf* (*map-pmf f p*) *s*) *x*) = *ennreal* (*pmf* (*map-pmf f* (*cond-pmf p* (*f* −‘ *s*))) *x*)
by *simp* **}**
then show *?thesis*
by (*intro pmf-eqI*) (*simp add*: *pmf-nonneg*)
qed

lemma *bind-cond-pmf-cancel*:
assumes [*simp*]: ⋀*x*. *x* ∈ *set-pmf p* ⟹ *set-pmf q* ∩ {*y*. *R x y*} ≠ {}
assumes [*simp*]: ⋀*y*. *y* ∈ *set-pmf q* ⟹ *set-pmf p* ∩ {*x*. *R x y*} ≠ {}
assumes [*simp*]: ⋀*x y*. *x* ∈ *set-pmf p* ⟹ *y* ∈ *set-pmf q* ⟹ *R x y* ⟹ *measure q* {*y*. *R x y*} = *measure p* {*x*. *R x y*}
shows *bind-pmf p* (λ*x*. *cond-pmf q* {*y*. *R x y*}) = *q*
proof (*rule pmf-eqI*)
fix *i*
have *ennreal* (*pmf* (*bind-pmf p* (λ*x*. *cond-pmf q* {*y*. *R x y*})) *i*) =
(∫⁺*x*. *ennreal* (*pmf q i* / *measure p* {*x*. *R x i*}) ∗ *ennreal* (*indicator* {*x*. *R x i*} *x*) ∂*p*)
by (*auto simp add*: *ennreal-pmf-bind AE-measure-pmf-iff pmf-cond pmf-eq-0-set-pmf pmf-nonneg measure-nonneg intro*!: *nn-integral-cong-AE*)
also have ... = (*pmf q i* ∗ *measure p* {*x*. *R x i*}) / *measure p* {*x*. *R x i*}
by (*simp add*: *pmf-nonneg measure-nonneg zero-ennreal-def*[*symmetric*] *ennreal-indicator nn-integral-cmult measure-pmf.emeasure-eq-measure ennreal-mult*[*symmetric*])
also have ... = *pmf q i*
by (*cases pmf q i* = *0*) (*simp-all add*: *pmf-eq-0-set-pmf measure-measure-pmf-not-zero pmf-nonneg*)
finally show *pmf* (*bind-pmf p* (λ*x*. *cond-pmf q* {*y*. *R x y*})) *i* = *pmf q i*
by (*simp add*: *pmf-nonneg*)
qed

20.5 Relator

inductive *rel-pmf* :: ('*a* ⇒ '*b* ⇒ *bool*) ⇒ '*a pmf* ⇒ '*b pmf* ⇒ *bool*
for *R p q*
where
⟦ ⋀*x y*. (*x*, *y*) ∈ *set-pmf pq* ⟹ *R x y*;
map-pmf fst pq = *p*; *map-pmf snd pq* = *q* ⟧
⟹ *rel-pmf R p q*

lemma *rel-pmfI*:
assumes *R*: *rel-set R* (*set-pmf p*) (*set-pmf q*)
assumes *eq*: ⋀*x y*. *x* ∈ *set-pmf p* ⟹ *y* ∈ *set-pmf q* ⟹ *R x y* ⟹ *measure p* {*x*. *R x y*} = *measure q* {*y*. *R x y*}
shows *rel-pmf R p q*

proof
let *?pq* = *bind-pmf p* (λ*x*. *bind-pmf* (*cond-pmf q* {*y*. *R x y*}) (λ*y*. *return-pmf* (*x*, *y*)))
have ⋀*x*. *x* ∈ *set-pmf p* ⟹ *set-pmf q* ∩ {*y*. *R x y*} ≠ {}
using *R* **by** (*auto simp*: *rel-set-def*)
then show ⋀*x y*. (*x*, *y*) ∈ *set-pmf ?pq* ⟹ *R x y*
by *auto*
show *map-pmf fst ?pq* = *p*
by (*simp add*: *map-bind-pmf bind-return-pmf'*)

show *map-pmf snd ?pq* = *q*
using *R eq*
apply (*simp add*: *bind-cond-pmf-cancel map-bind-pmf bind-return-pmf'*)
apply (*rule bind-cond-pmf-cancel*)
apply (*auto simp*: *rel-set-def*)
done
qed

lemma *rel-pmf-imp-rel-set*: *rel-pmf R p q* ⟹ *rel-set R* (*set-pmf p*) (*set-pmf q*)
by (*force simp add*: *rel-pmf.simps rel-set-def*)

lemma *rel-pmfD-measure*:
assumes *rel-R*: *rel-pmf R p q* **and** *R*: ⋀*a b*. *R a b* ⟹ *R a y* ⟷ *R x b*
assumes *x* ∈ *set-pmf p y* ∈ *set-pmf q*
shows *measure p* {*x*. *R x y*} = *measure q* {*y*. *R x y*}
proof −
from *rel-R* **obtain** *pq* **where** *pq*: ⋀*x y*. (*x*, *y*) ∈ *set-pmf pq* ⟹ *R x y*
and *eq*: *p* = *map-pmf fst pq q* = *map-pmf snd pq*
by (*auto elim*: *rel-pmf.cases*)
have *measure p* {*x*. *R x y*} = *measure pq* {*x*. *R* (*fst x*) *y*}
by (*simp add*: *eq map-pmf-rep-eq measure-distr*)
also have ... = *measure pq* {*y*. *R x* (*snd y*)}
by (*intro measure-pmf.finite-measure-eq-AE*
(*auto simp*: *AE-measure-pmf-iff R dest*!: *pq*)
also have ... = *measure q* {*y*. *R x y*}
by (*simp add*: *eq map-pmf-rep-eq measure-distr*)
finally show *measure p* {*x*. *R x y*} = *measure q* {*y*. *R x y*} **.**
qed

lemma *rel-pmf-measureD*:
assumes *rel-pmf R p q*
shows *measure* (*measure-pmf p*) *A* ≤ *measure* (*measure-pmf q*) {*y*. ∃*x*∈*A*. *R x y*} (**is** *?lhs* ≤ *?rhs*)
using *assms*
proof *cases*
fix *pq*
assume *R*: ⋀*x y*. (*x*, *y*) ∈ *set-pmf pq* ⟹ *R x y*
and *p*[*symmetric*]: *map-pmf fst pq* = *p*
and *q*[*symmetric*]: *map-pmf snd pq* = *q*

have *?lhs* = *measure* (*measure-pmf pq*) (*fst* −‘ *A*) **by**(*simp add*: *p*)
also have ... ≤ *measure* (*measure-pmf pq*) {*y*. ∃ *x*∈*A*. *R x* (*snd y*)}
by(*rule measure-pmf.finite-measure-mono-AE*)(*auto 4 3 simp add*: *AE-measure-pmf-iff dest*: *R*)
also have ... = *?rhs* **by**(*simp add*: *q*)
finally show *?thesis* **.**
qed

lemma *rel-pmf-iff-measure*:
assumes *symp R transp R*
shows *rel-pmf R p q* ⟷
rel-set R (*set-pmf p*) (*set-pmf q*) ∧
(∀ *x*∈*set-pmf p*. ∀ *y*∈*set-pmf q*. *R x y* ⟶ *measure p* {*x*. *R x y*} = *measure q* {*y*. *R x y*})
by (*safe intro*!: *rel-pmf-imp-rel-set rel-pmfI*)
(*auto intro*!: *rel-pmfD-measure dest*: *sympD*[*OF* ‹*symp R*›] *transpD*[*OF* ‹*transp R*›])

lemma *quotient-rel-set-disjoint*:
equivp R ⟹ *C* ∈ *UNIV* // {(*x*, *y*). *R x y*} ⟹ *rel-set R A B* ⟹ *A* ∩ *C* = {} ⟷ *B* ∩ *C* = {}
using *in-quotient-imp-closed*[*of UNIV* {(*x*, *y*). *R x y*} *C*]
by (*auto 0 0 simp*: *equivp-equiv rel-set-def set-eq-iff elim*: *equivpE*)
(*blast dest*: *equivp-symp*)+

lemma *quotientD*: *equiv X R* ⟹ *A* ∈ *X* // *R* ⟹ *x* ∈ *A* ⟹ *A* = *R* `` {*x*}
by (*metis Image-singleton-iff equiv-class-eq-iff quotientE*)

lemma *rel-pmf-iff-equivp*:
assumes *equivp R*
shows *rel-pmf R p q* ⟷ (∀ *C*∈*UNIV* // {(*x*, *y*). *R x y*}. *measure p C* = *measure q C*)
(**is** - ⟷ (∀ *C*∈-// *?R*. -))
proof (*subst rel-pmf-iff-measure*, *safe*)
show *symp R transp R*
using *assms* **by** (*auto simp*: *equivp-reflp-symp-transp*)
next
fix *C* **assume** *C*: *C* ∈ *UNIV* // *?R* **and** *R*: *rel-set R* (*set-pmf p*) (*set-pmf q*)
assume *eq*: ∀ *x*∈*set-pmf p*. ∀ *y*∈*set-pmf q*. *R x y* ⟶ *measure p* {*x*. *R x y*} = *measure q* {*y*. *R x y*}

show *measure p C* = *measure q C*
proof (*cases p* ∩ *C* = {})
case *True*
then have *q* ∩ *C* = {}
using *quotient-rel-set-disjoint*[*OF assms C R*] **by** *simp*
with *True* **show** *?thesis*
unfolding *measure-pmf-zero-iff*[*symmetric*] **by** *simp*
next

```
    case False
    then have q ∩ C ≠ {}
      using quotient-rel-set-disjoint[OF assms C R] by simp
    with False obtain x y where in-set: x ∈ set-pmf p y ∈ set-pmf q and in-C:
x ∈ C y ∈ C
      by auto
    then have R x y
      using in-quotient-imp-in-rel[of UNIV ?R C x y] C assms
      by (simp add: equivp-equiv)
    with in-set eq have measure p {x. R x y} = measure q {y. R x y}
      by auto
    moreover have {y. R x y} = C
     using assms ‹x ∈ C› C quotientD[of UNIV ?R C x] by (simp add: equivp-equiv)
    moreover have {x. R x y} = C
      using assms ‹y ∈ C› C quotientD[of UNIV ?R C y] sympD[of R]
      by (auto simp add: equivp-equiv elim: equivpE)
    ultimately show ?thesis
      by auto
  qed
next
  assume eq: ∀ C∈UNIV // ?R. measure p C = measure q C
  show rel-set R (set-pmf p) (set-pmf q)
    unfolding rel-set-def
  proof safe
    fix x assume x: x ∈ set-pmf p
    have {y. R x y} ∈ UNIV // ?R
      by (auto simp: quotient-def)
    with eq have ∗: measure q {y. R x y} = measure p {y. R x y}
      by auto
    have measure q {y. R x y} ≠ 0
       using x assms unfolding ∗ by (auto simp: measure-pmf-zero-iff set-eq-iff
dest: equivp-reflp)
    then show ∃ y∈set-pmf q. R x y
      unfolding measure-pmf-zero-iff by auto
  next
    fix y assume y: y ∈ set-pmf q
    have {x. R x y} ∈ UNIV // ?R
      using assms by (auto simp: quotient-def dest: equivp-symp)
    with eq have ∗: measure p {x. R x y} = measure q {x. R x y}
      by auto
    have measure p {x. R x y} ≠ 0
       using y assms unfolding ∗ by (auto simp: measure-pmf-zero-iff set-eq-iff
dest: equivp-reflp)
    then show ∃ x∈set-pmf p. R x y
      unfolding measure-pmf-zero-iff by auto
  qed

  fix x y assume x ∈ set-pmf p y ∈ set-pmf q R x y
  have {y. R x y} ∈ UNIV // ?R {x. R x y} = {y. R x y}
```

```
  using assms ‹R x y› by (auto simp: quotient-def dest: equivp-symp equivp-transp)
 with eq show measure p {x. R x y} = measure q {y. R x y}
   by auto
qed

bnf pmf: 'a pmf map: map-pmf sets: set-pmf bd : natLeq rel: rel-pmf
proof −
 show map-pmf id = id by (rule map-pmf-id)
 show ⋀f g. map-pmf (f ∘ g) = map-pmf f ∘ map-pmf g by (rule map-pmf-compose)
 show ⋀f g::'a ⇒ 'b. ⋀p. (⋀x. x ∈ set-pmf p ⟹ f x = g x) ⟹ map-pmf f p =
map-pmf g p
   by (intro map-pmf-cong refl)

 show ⋀f::'a ⇒ 'b. set-pmf ∘ map-pmf f = (`) f ∘ set-pmf
   by (rule pmf-set-map)

 show (card-of (set-pmf p), natLeq) ∈ ordLeq for p :: 's pmf
 proof −
   have (card-of (set-pmf p), card-of (UNIV :: nat set)) ∈ ordLeq
     by (rule card-of-ordLeqI[where f=to-nat-on (set-pmf p)])
        (auto intro: countable-set-pmf)
   also have (card-of (UNIV :: nat set), natLeq) ∈ ordLeq
     by (metis Field-natLeq card-of-least natLeq-Well-order)
   finally show ?thesis .
 qed

 show ⋀R. rel-pmf R = (λx y. ∃z. set-pmf z ⊆ {(x, y). R x y} ∧
   map-pmf fst z = x ∧ map-pmf snd z = y)
    by (auto simp add: fun-eq-iff rel-pmf.simps)

 show rel-pmf R OO rel-pmf S ≤ rel-pmf (R OO S)
   for R :: 'a ⇒ 'b ⇒ bool and S :: 'b ⇒ 'c ⇒ bool
 proof −
   { fix p q r
     assume pq: rel-pmf R p q
       and qr:rel-pmf S q r
     from pq obtain pq where pq: ⋀x y. (x, y) ∈ set-pmf pq ⟹ R x y
       and p: p = map-pmf fst pq and q: q = map-pmf snd pq by cases auto
     from qr obtain qr where qr: ⋀y z. (y, z) ∈ set-pmf qr ⟹ S y z
       and q': q = map-pmf fst qr and r: r = map-pmf snd qr by cases auto

     define pr where pr =
       bind-pmf pq (λxy. bind-pmf (cond-pmf qr {yz. fst yz = snd xy})
         (λyz. return-pmf (fst xy, snd yz)))
     have pr-welldefined: ⋀y. y ∈ q ⟹ qr ∩ {yz. fst yz = y} ≠ {}
       by (force simp: q')

     have rel-pmf (R OO S) p r
     proof (rule rel-pmf.intros)
```

```
      fix x z assume (x, z) ∈ pr
      then have ∃ y. (x, y) ∈ pq ∧ (y, z) ∈ qr
        by (auto simp: q pr-welldefined pr-def split-beta)
      with pq qr show (R OO S) x z
        by blast
    next
      have map-pmf snd pr = map-pmf snd (bind-pmf q (λy. cond-pmf qr {yz. fst
yz = y}))
        by (simp add: pr-def q split-beta bind-map-pmf map-pmf-def[symmetric]
map-bind-pmf map-pmf-comp)
      then show map-pmf snd pr = r
       unfolding r q' bind-map-pmf by (subst (asm) bind-cond-pmf-cancel) (auto
simp: eq-commute)
      qed (simp add: pr-def map-bind-pmf split-beta map-pmf-def[symmetric] p
map-pmf-comp)
  }
  then show ?thesis
    by(auto simp add: le-fun-def)
 qed
qed (fact natLeq-card-order natLeq-cinfinite)+

lemma map-pmf-idI: (⋀x. x ∈ set-pmf p ⟹ f x = x) ⟹ map-pmf f p = p
by(simp cong: pmf.map-cong)

lemma rel-pmf-conj[simp]:
  rel-pmf (λx y. P ∧ Q x y) x y ⟷ P ∧ rel-pmf Q x y
  rel-pmf (λx y. Q x y ∧ P) x y ⟷ P ∧ rel-pmf Q x y
  using set-pmf-not-empty by (fastforce simp: pmf.in-rel subset-eq)+

lemma rel-pmf-top[simp]: rel-pmf top = top
  by (auto simp: pmf.in-rel[abs-def] fun-eq-iff map-fst-pair-pmf map-snd-pair-pmf
        intro: exI[of - pair-pmf x y for x y])

lemma rel-pmf-return-pmf1: rel-pmf R (return-pmf x) M ⟷ (∀ a∈M. R x a)
proof safe
  fix a assume a ∈ M rel-pmf R (return-pmf x) M
  then obtain pq where ∗: ⋀a b. (a, b) ∈ set-pmf pq ⟹ R a b
    and eq: return-pmf x = map-pmf fst pq M = map-pmf snd pq
    by (force elim: rel-pmf.cases)
  moreover have set-pmf (return-pmf x) = {x}
    by simp
  with ‹a ∈ M› have (x, a) ∈ pq
    by (force simp: eq)
  with ∗ show R x a
    by auto
qed (auto intro!: rel-pmf.intros[where pq=pair-pmf (return-pmf x) M]
        simp: map-fst-pair-pmf map-snd-pair-pmf)

lemma rel-pmf-return-pmf2: rel-pmf R M (return-pmf x) ⟷ (∀ a∈M. R a x)
```

by (*subst pmf.rel-flip*[*symmetric*]) (*simp add*: *rel-pmf-return-pmf1*)

lemma *rel-return-pmf*[*simp*]: *rel-pmf R* (*return-pmf x1*) (*return-pmf x2*) = *R x1 x2*
unfolding *rel-pmf-return-pmf2 set-return-pmf* **by** *simp*

lemma *rel-pmf-False*[*simp*]: *rel-pmf* (λ*x y*. *False*) *x y* = *False*
unfolding *pmf.in-rel fun-eq-iff* **using** *set-pmf-not-empty* **by** *fastforce*

lemma *rel-pmf-rel-prod*:
rel-pmf (*rel-prod R S*) (*pair-pmf A A'*) (*pair-pmf B B'*) ⟷ *rel-pmf R A B* ∧ *rel-pmf S A' B'*
proof *safe*
assume *rel-pmf* (*rel-prod R S*) (*pair-pmf A A'*) (*pair-pmf B B'*)
then obtain *pq* **where** *pq*: ⋀*a b c d*. ((*a*, *c*), (*b*, *d*)) ∈ *set-pmf pq* ⟹ *R a b* ∧ *S c d*
and *eq*: *map-pmf fst pq* = *pair-pmf A A' map-pmf snd pq* = *pair-pmf B B'*
by (*force elim*: *rel-pmf.cases*)
show *rel-pmf R A B*
proof (*rule rel-pmf.intros*)
let *?f* = λ(*a*, *b*). (*fst a*, *fst b*)
have [*simp*]: (λ*x*. *fst* (*?f x*)) = *fst o fst* (λ*x*. *snd* (*?f x*)) = *fst o snd*
by *auto*

show *map-pmf fst* (*map-pmf ?f pq*) = *A*
by (*simp add*: *map-pmf-comp pmf.map-comp*[*symmetric*] *eq map-fst-pair-pmf*)
show *map-pmf snd* (*map-pmf ?f pq*) = *B*
by (*simp add*: *map-pmf-comp pmf.map-comp*[*symmetric*] *eq map-fst-pair-pmf*)

fix *a b* **assume** (*a*, *b*) ∈ *set-pmf* (*map-pmf ?f pq*)
then obtain *c d* **where** ((*a*, *c*), (*b*, *d*)) ∈ *set-pmf pq*
by *auto*
from *pq*[*OF this*] **show** *R a b* **..**
qed
show *rel-pmf S A' B'*
proof (*rule rel-pmf.intros*)
let *?f* = λ(*a*, *b*). (*snd a*, *snd b*)
have [*simp*]: (λ*x*. *fst* (*?f x*)) = *snd o fst* (λ*x*. *snd* (*?f x*)) = *snd o snd*
by *auto*

show *map-pmf fst* (*map-pmf ?f pq*) = *A'*
by (*simp add*: *map-pmf-comp pmf.map-comp*[*symmetric*] *eq map-snd-pair-pmf*)
show *map-pmf snd* (*map-pmf ?f pq*) = *B'*
by (*simp add*: *map-pmf-comp pmf.map-comp*[*symmetric*] *eq map-snd-pair-pmf*)

fix *c d* **assume** (*c*, *d*) ∈ *set-pmf* (*map-pmf ?f pq*)
then obtain *a b* **where** ((*a*, *c*), (*b*, *d*)) ∈ *set-pmf pq*
by *auto*
from *pq*[*OF this*] **show** *S c d* **..**

```
  qed
next
  assume rel-pmf R A B rel-pmf S A' B'
  then obtain Rpq Spq
    where Rpq: ⋀a b. (a, b) ∈ set-pmf Rpq ⟹ R a b
        map-pmf fst Rpq = A map-pmf snd Rpq = B
      and Spq: ⋀a b. (a, b) ∈ set-pmf Spq ⟹ S a b
        map-pmf fst Spq = A' map-pmf snd Spq = B'
    by (force elim: rel-pmf.cases)

  let ?f = (λ((a, c), (b, d)). ((a, b), (c, d)))
  let ?pq = map-pmf ?f (pair-pmf Rpq Spq)
  have [simp]: (λx. fst (?f x)) = (λ(a, b). (fst a, fst b)) (λx. snd (?f x)) = (λ(a,
b). (snd a, snd b))
    by auto

  show rel-pmf (rel-prod R S) (pair-pmf A A') (pair-pmf B B')
    by (rule rel-pmf.intros[where pq=?pq])
       (auto simp: map-snd-pair-pmf map-fst-pair-pmf map-pmf-comp Rpq Spq
                   map-pair)
qed

lemma rel-pmf-reflI:
  assumes ⋀x. x ∈ set-pmf p ⟹ P x x
  shows rel-pmf P p p
  by (rule rel-pmf.intros[where pq=map-pmf (λx. (x, x)) p])
     (auto simp add: pmf.map-comp o-def assms)

lemma rel-pmf-bij-betw:
  assumes f: bij-betw f (set-pmf p) (set-pmf q)
  and eq: ⋀x. x ∈ set-pmf p ⟹ pmf p x = pmf q (f x)
  shows rel-pmf (λx y. f x = y) p q
proof(rule rel-pmf.intros)
  let ?pq = map-pmf (λx. (x, f x)) p
  show map-pmf fst ?pq = p by(simp add: pmf.map-comp o-def)

  have map-pmf f p = q
  proof(rule pmf-eqI)
    fix i
    show pmf (map-pmf f p) i = pmf q i
    proof(cases i ∈ set-pmf q)
      case True
      with f obtain j where i = f j j ∈ set-pmf p
        by(auto simp add: bij-betw-def image-iff)
      thus ?thesis using f by(simp add: bij-betw-def pmf-map-inj eq)
    next
      case False thus ?thesis
        by(subst pmf-map-outside)(auto simp add: set-pmf-iff eq[symmetric])
    qed
```

qed
then show *map-pmf snd ?pq = q* **by**(*simp add*: *pmf.map-comp o-def*)
qed *auto*

context
begin

interpretation *pmf-as-measure* **.**

definition *join-pmf M = bind-pmf M* (λ*x*. *x*)

lemma *bind-eq-join-pmf*: *bind-pmf M f = join-pmf* (*map-pmf f M*)
unfolding *join-pmf-def bind-map-pmf* **..**

lemma *join-eq-bind-pmf*: *join-pmf M = bind-pmf M id*
by (*simp add*: *join-pmf-def id-def*)

lemma *pmf-join*: *pmf* (*join-pmf N*) *i* = (∫ *M*. *pmf M i* ∂*measure-pmf N*)
unfolding *join-pmf-def pmf-bind* **..**

lemma *ennreal-pmf-join*: *ennreal* (*pmf* (*join-pmf N*) *i*) = (∫$^+$*M*. *pmf M i* ∂*measure-pmf N*)
unfolding *join-pmf-def ennreal-pmf-bind* **..**

lemma *set-pmf-join-pmf*[*simp*]: *set-pmf* (*join-pmf f*) = (⋃*p*∈*set-pmf f*. *set-pmf p*)
by (*simp add*: *join-pmf-def*)

lemma *join-return-pmf*: *join-pmf* (*return-pmf M*) = *M*
by (*simp add*: *integral-return pmf-eq-iff pmf-join return-pmf.rep-eq*)

lemma *map-join-pmf*: *map-pmf f* (*join-pmf AA*) = *join-pmf* (*map-pmf* (*map-pmf f*) *AA*)
by (*simp add*: *join-pmf-def map-pmf-def bind-assoc-pmf bind-return-pmf*)

lemma *join-map-return-pmf*: *join-pmf* (*map-pmf return-pmf A*) = *A*
by (*simp add*: *join-pmf-def map-pmf-def bind-assoc-pmf bind-return-pmf bind-return-pmf'*)

end

lemma *rel-pmf-joinI*:
assumes *rel-pmf* (*rel-pmf P*) *p q*
shows *rel-pmf P* (*join-pmf p*) (*join-pmf q*)
proof −
from *assms* **obtain** *pq* **where** *p*: *p = map-pmf fst pq*
and *q*: *q = map-pmf snd pq*
and *P*: ⋀*x y*. (*x*, *y*) ∈ *set-pmf pq* ⟹ *rel-pmf P x y*
by *cases auto*
from *P* **obtain** *PQ*

where *PQ*: $\bigwedge x\ y\ a\ b.\ [\![(x, y) \in \mathit{set\text{-}pmf}\ pq;\ (a, b) \in \mathit{set\text{-}pmf}\ (PQ\ x\ y)]\!] \Longrightarrow P\ a\ b$
and *x*: $\bigwedge x\ y.\ (x, y) \in \mathit{set\text{-}pmf}\ pq \Longrightarrow \mathit{map\text{-}pmf}\ \mathit{fst}\ (PQ\ x\ y) = x$
and *y*: $\bigwedge x\ y.\ (x, y) \in \mathit{set\text{-}pmf}\ pq \Longrightarrow \mathit{map\text{-}pmf}\ \mathit{snd}\ (PQ\ x\ y) = y$
by(*metis rel-pmf.simps*)

let *?r* = *bind-pmf pq* (λ(*x*, *y*). *PQ x y*)
have $\bigwedge a\ b.\ (a, b) \in \mathit{set\text{-}pmf}\ ?r \Longrightarrow P\ a\ b$ **by** (*auto intro*: *PQ*)
moreover have *map-pmf fst ?r* = *join-pmf p map-pmf snd ?r* = *join-pmf q*
by (*simp-all add*: *p q x y join-pmf-def map-bind-pmf bind-map-pmf split-def cong*: *bind-pmf-cong*)
ultimately show *?thesis* **..**
qed

lemma *rel-pmf-bindI*:
assumes *pq*: *rel-pmf R p q*
and *fg*: $\bigwedge x\ y.\ R\ x\ y \Longrightarrow \mathit{rel\text{-}pmf}\ P\ (f\ x)\ (g\ y)$
shows *rel-pmf P* (*bind-pmf p f*) (*bind-pmf q g*)
unfolding *bind-eq-join-pmf*
by (*rule rel-pmf-joinI*)
(*auto simp add*: *pmf.rel-map intro*: *pmf.rel-mono*[*THEN le-funD*, *THEN le-funD*, *THEN le-boolD*, *THEN mp*, *OF - pq*] *fg*)

Proof that *rel-pmf* preserves orders. Antisymmetry proof follows Thm. 1 in N. Saheb-Djahromi, Cpo's of measures for nondeterminism, Theoretical Computer Science 12(1):19–37, 1980, https://doi.org/10.1016/0304-3975(80)90003-1

lemma
assumes ∗: *rel-pmf R p q*
and *refl*: *reflp R* **and** *trans*: *transp R*
shows *measure-Ici*: *measure p* {*y*. *R x y*} ≤ *measure q* {*y*. *R x y*} (**is** *?thesis1*)
and *measure-Ioi*: *measure p* {*y*. *R x y* ∧ ¬ *R y x*} ≤ *measure q* {*y*. *R x y* ∧ ¬ *R y x*} (**is** *?thesis2*)
proof −
from ∗ **obtain** *pq*
where *pq*: $\bigwedge x\ y.\ (x, y) \in \mathit{set\text{-}pmf}\ pq \Longrightarrow R\ x\ y$
and *p*: *p* = *map-pmf fst pq*
and *q*: *q* = *map-pmf snd pq*
by *cases auto*
show *?thesis1 ?thesis2* **unfolding** *p q map-pmf-rep-eq* **using** *refl trans*
by(*auto 4 3 simp add*: *measure-distr reflpD AE-measure-pmf-iff intro*!: *measure-pmf.finite-measure-mono-AE dest*!: *pq elim*: *transpE*)
qed

lemma *rel-pmf-inf*:
fixes *p q* :: *'a pmf*
assumes *1*: *rel-pmf R p q*
assumes *2*: *rel-pmf R q p*
and *refl*: *reflp R* **and** *trans*: *transp R*

shows *rel-pmf* (*inf R* R^{-1-1}) *p q*
proof (*subst rel-pmf-iff-equivp*, *safe*)
show *equivp* (*inf R* R^{-1-1})
using *trans refl* **by** (*auto simp*: *equivp-reflp-symp-transp intro*: *sympI transpI reflpI dest*: *transpD reflpD*)

fix *C* **assume** $C \in UNIV$ // $\{(x, y).$ *inf R* R^{-1-1} *x y*$\}$
then obtain *x* **where** *C*: $C = \{y.\ R\ x\ y \wedge R\ y\ x\}$
by (*auto elim*: *quotientE*)

let *?R* = $\lambda x\ y.\ R\ x\ y \wedge R\ y\ x$
let *?μR* = $\lambda y.$ *measure q* $\{x.$ *?R x y*$\}$
have *measure p* $\{y.$ *?R x y*$\}$ = *measure p* $(\{y.\ R\ x\ y\} - \{y.\ R\ x\ y \wedge \neg\ R\ y\ x\})$
by(*auto intro*!: *arg-cong*[**where** *f*=*measure p*])
also have ... = *measure p* $\{y.\ R\ x\ y\}$ − *measure p* $\{y.\ R\ x\ y \wedge \neg\ R\ y\ x\}$
by (*rule measure-pmf.finite-measure-Diff*) *auto*
also have *measure p* $\{y.\ R\ x\ y \wedge \neg\ R\ y\ x\}$ = *measure q* $\{y.\ R\ x\ y \wedge \neg\ R\ y\ x\}$
using *1 2 refl trans* **by**(*auto intro*!: *Orderings.antisym measure-Ioi*)
also have *measure p* $\{y.\ R\ x\ y\}$ = *measure q* $\{y.\ R\ x\ y\}$
using *1 2 refl trans* **by**(*auto intro*!: *Orderings.antisym measure-Ici*)
also have *measure q* $\{y.\ R\ x\ y\}$ − *measure q* $\{y.\ R\ x\ y \wedge \neg\ R\ y\ x\}$ =
measure q $(\{y.\ R\ x\ y\} - \{y.\ R\ x\ y \wedge \neg\ R\ y\ x\})$
by(*rule measure-pmf.finite-measure-Diff*[*symmetric*]) *auto*
also have ... = *?μR x*
by(*auto intro*!: *arg-cong*[**where** *f*=*measure q*])
finally show *measure p C* = *measure q C*
by (*simp add*: *C conj-commute*)
qed

lemma *rel-pmf-antisym*:
fixes *p q* :: *'a pmf*
assumes *1*: *rel-pmf R p q*
assumes *2*: *rel-pmf R q p*
and *refl*: *reflp R* **and** *trans*: *transp R* **and** *antisym*: *antisymp R*
shows *p* = *q*
proof −
from *1 2 refl trans* **have** *rel-pmf* (*inf R* R^{-1-1}) *p q* **by**(*rule rel-pmf-inf*)
also have *inf R* R^{-1-1} = (=)
using *refl antisym* **by** (*auto intro*!: *ext simp add*: *reflpD dest*: *antisympD*)
finally show *?thesis* **unfolding** *pmf.rel-eq* **.**
qed

lemma *reflp-rel-pmf*: *reflp R* ⟹ *reflp* (*rel-pmf R*)
by (*fact pmf.rel-reflp*)

lemma *antisymp-rel-pmf*:
⟦ *reflp R*; *transp R*; *antisymp R* ⟧
⟹ *antisymp* (*rel-pmf R*)
by(*rule antisympI*)(*blast intro*: *rel-pmf-antisym*)

lemma *transp-rel-pmf*:
assumes *transp R*
shows *transp* (*rel-pmf R*)
using *assms* **by** (*fact pmf.rel-transp*)

20.6 Distributions

context
begin

interpretation *pmf-as-function* **.**

20.6.1 Bernoulli Distribution

lift-definition *bernoulli-pmf* :: *real* ⇒ *bool pmf* **is**
λp b. ((*λp. if b then p else 1 − p*) ∘ *min 1* ∘ *max 0*) *p*
by (*auto simp*: *nn-integral-count-space-finite*[**where** *A*={*False*, *True*}] *UNIV-bool split*: *split-max split-min*)

lemma *pmf-bernoulli-True*[*simp*]: *0* ≤ *p* ⟹ *p* ≤ *1* ⟹ *pmf* (*bernoulli-pmf p*) *True* = *p*
by *transfer simp*

lemma *pmf-bernoulli-False*[*simp*]: *0* ≤ *p* ⟹ *p* ≤ *1* ⟹ *pmf* (*bernoulli-pmf p*) *False* = *1* − *p*
by *transfer simp*

lemma *set-pmf-bernoulli*[*simp*]: *0* < *p* ⟹ *p* < *1* ⟹ *set-pmf* (*bernoulli-pmf p*) = *UNIV*
by (*auto simp add*: *set-pmf-iff UNIV-bool*)

lemma *nn-integral-bernoulli-pmf*[*simp*]:
assumes [*simp*]: *0* ≤ *p p* ≤ *1* ⋀*x. 0* ≤ *f x*
shows (∫⁺*x. f x* ∂*bernoulli-pmf p*) = *f True* ∗ *p* + *f False* ∗ (*1* − *p*)
by (*subst nn-integral-measure-pmf-support*[*of UNIV*])
(*auto simp*: *UNIV-bool field-simps*)

lemma *integral-bernoulli-pmf*[*simp*]:
assumes [*simp*]: *0* ≤ *p p* ≤ *1*
shows (∫ *x. f x* ∂*bernoulli-pmf p*) = *f True* ∗ *p* + *f False* ∗ (*1* − *p*)
by (*subst integral-measure-pmf*[*of UNIV*]) (*auto simp*: *UNIV-bool*)

lemma *pmf-bernoulli-half* [*simp*]: *pmf* (*bernoulli-pmf* (*1* / *2*)) *x* = *1* / *2*
by(*cases x*) *simp-all*

lemma *measure-pmf-bernoulli-half*: *measure-pmf* (*bernoulli-pmf* (*1* / *2*)) = *uniform-count-measure UNIV*
by (*rule measure-eqI*)

(*simp-all add*: *nn-integral-pmf*[*symmetric*] *emeasure-uniform-count-measure ennreal-divide-numeral*[*symmetric*]
nn-integral-count-space-finite sets-uniform-count-measure divide-ennreal-def mult-ac
ennreal-of-nat-eq-real-of-nat)

20.6.2 Geometric Distribution

context
fixes p :: *real* **assumes** p[*arith*]: $0 < p\ p \leq 1$
begin

lift-definition *geometric-pmf* :: *nat pmf* **is** $\lambda n.\ (1 - p) \hat{}\ n * p$
proof
have $(\sum i.\ ennreal\ (p * (1 - p) \hat{}\ i)) = ennreal\ (p * (1 / (1 - (1 - p))))$
by (*intro suminf-ennreal-eq sums-mult geometric-sums*) *auto*
then show $(\int^+ x.\ ennreal\ ((1 - p) \hat{}\ x * p)\ \partial count\text{-}space\ UNIV) = 1$
by (*simp add*: *nn-integral-count-space-nat field-simps*)
qed *simp*

lemma *pmf-geometric*[*simp*]: *pmf geometric-pmf n* $= (1 - p) \hat{}\ n * p$
by *transfer rule*

end

lemma *geometric-pmf-1* [*simp*]: *geometric-pmf 1* = *return-pmf 0*
by (*intro pmf-eqI*) (*auto simp*: *indicator-def*)

lemma *set-pmf-geometric*: $0 < p \Longrightarrow p < 1 \Longrightarrow$ *set-pmf* (*geometric-pmf p*) = *UNIV*
by (*auto simp*: *set-pmf-iff*)

lemma *geometric-sums-times-n*:
fixes c::$'a$::{*banach*,*real-normed-field*}
assumes *norm c* < 1
shows $(\lambda n.\ c \hat{}\ n * of\text{-}nat\ n)$ *sums* $(c / (1 - c)^2)$
proof −
have $(\lambda n.\ c * z \hat{}\ n)$ *sums* $(c / (1 - z))$ **if** *norm z* < 1 **for** z
using *geometric-sums sums-mult that* **by** *fastforce*
moreover have $((\lambda z.\ c / (1 - z))\ has\text{-}field\text{-}derivative\ (c / (1 - c)^2))\ (at\ c)$
using *assms* **by** (*auto intro*!: *derivative-eq-intros simp add*: *semiring-normalization-rules*)
ultimately have $(\lambda n.\ diffs\ (\lambda n.\ c)\ n * c \hat{}\ n)$ *sums* $(c / (1 - c)^2)$
using *assms* **by** (*intro termdiffs-sums-strong*)
then have $(\lambda n.\ of\text{-}nat\ (Suc\ n) * c \hat{}\ (Suc\ n))$ *sums* $(c / (1 - c)^2)$
unfolding *diffs-def* **by** (*simp add*: *power-eq-if mult.assoc*)
then show *?thesis*
by (*subst* (*asm*) *sums-Suc-iff*) (*auto simp add*: *mult.commute*)
qed

lemma *geometric-sums-times-norm*:
fixes *c*::*'a*::{*banach*,*real-normed-field*}
assumes *norm c < 1*
shows (λ*n*. *norm* (*c*^*n* ∗ *of-nat n*)) *sums* (*norm c* / (*1* − *norm c*)²)
proof −
have *norm* (*c*^*n* ∗ *of-nat n*) = (*norm c*) ^ *n* ∗ *of-nat n* **for** *n*::*nat*
by (*simp add*: *norm-power norm-mult*)
then show *?thesis*
using *geometric-sums-times-n*[*of norm c*] *assms*
by *force*
qed

lemma *integrable-real-geometric-pmf*:
assumes *p* ∈ {*0*<..*1*}
shows *integrable* (*geometric-pmf p*) *real*
proof −
have *summable* (λ*x*. *p* ∗ ((*1* − *p*) ^ *x* ∗ *real x*))
using *geometric-sums-times-norm*[*of 1* − *p*] *assms*
by (*intro summable-mult*) (*auto simp*: *sums-iff*)
hence *summable* (λ*x*. (*1* − *p*) ^ *x* ∗ *real x*)
by (*rule summable-mult-D*) (*use assms* **in** *auto*)
thus *?thesis*
unfolding *measure-pmf-eq-density* **using** *assms*
by (*subst integrable-density*)
(*auto simp*: *integrable-count-space-nat-iff mult-ac*)
qed

lemma *expectation-geometric-pmf*:
assumes *p* ∈ {*0*<..*1*}
shows *measure-pmf*.*expectation* (*geometric-pmf p*) *real* = (*1* − *p*) / *p*
proof −
have (λ*n*. *p* ∗ ((*1* − *p*) ^ *n* ∗ *n*)) *sums* (*p* ∗ ((*1* − *p*) / *p*^*2*))
using *assms geometric-sums-times-n*[*of 1*−*p*] **by** (*intro sums-mult*) *auto*
moreover have (λ*n*. *p* ∗ ((*1* − *p*) ^ *n* ∗ *n*)) = (λ*n*. (*1* − *p*) ^ *n* ∗ *p* ∗ *real n*)
by *auto*
ultimately have ∗: (λ*n*. (*1* − *p*) ^ *n* ∗ *p* ∗ *real n*) *sums* ((*1* − *p*) / *p*)
using *assms sums-subst* **by** (*auto simp add*: *power2-eq-square*)
have *measure-pmf*.*expectation* (*geometric-pmf p*) *real* =
(∫ *n*. *pmf* (*geometric-pmf p*) *n* ∗ *real n* ∂*count-space UNIV*)
unfolding *measure-pmf-eq-density* **by** (*subst integral-density*) *auto*
also have *integrable* (*count-space UNIV*) (λ*n*. *pmf* (*geometric-pmf p*) *n* ∗ *real n*)
using ∗ *assms* **unfolding** *integrable-count-space-nat-iff* **by** (*simp add*: *sums-iff*)
hence (∫ *n*. *pmf* (*geometric-pmf p*) *n* ∗ *real n* ∂*count-space UNIV*) = (*1* − *p*) / *p*
using ∗ *assms* **by** (*subst integral-count-space-nat*) (*simp-all add*: *sums-iff*)
finally show *?thesis* **by** *auto*
qed

lemma *geometric-bind-pmf-unfold*:
assumes $p \in \{0<..1\}$
shows *geometric-pmf p* =
do {*b* ← *bernoulli-pmf p*;
if b then return-pmf 0 else map-pmf Suc (*geometric-pmf p*)}
proof −
have ∗: (*Suc* −‘ {*i*}) = (*if i* = *0 then* {} *else* {*i* − *1*}) **for** *i*
by *force*
have *pmf* (*geometric-pmf p*) *i* =
pmf (*bernoulli-pmf p* ⤜
(λ*b*. *if b then return-pmf 0 else map-pmf Suc* (*geometric-pmf p*)))
i **for** *i*
proof −
have *pmf* (*geometric-pmf p*) *i* =
(*if i* = *0 then p else* (*1* − *p*) ∗ *pmf* (*geometric-pmf p*) (*i* − *1*))
using *assms* **by** (*simp add*: *power-eq-if*)
also have ... = (*if i* = *0 then p else* (*1* − *p*) ∗ *pmf* (*map-pmf Suc* (*geometric-pmf p*)) *i*)
by (*simp add*: *pmf-map indicator-def measure-pmf-single* ∗)
also have ... = *measure-pmf.expectation* (*bernoulli-pmf p*)
(λ*x*. *pmf* (*if x then return-pmf 0 else map-pmf Suc* (*geometric-pmf p*)) *i*)
using *assms* **by** (*auto simp add*: *pmf-map* ∗)
also have ... = *pmf* (*bernoulli-pmf p* ⤜
(λ*b*. *if b then return-pmf 0 else map-pmf Suc* (*geometric-pmf p*)))
i
by (*auto simp add*: *pmf-bind*)
finally show *?thesis* **.**
qed
then show *?thesis*
using *pmf-eqI* **by** *blast*
qed

20.6.3 Uniform Multiset Distribution

context
fixes *M* :: *'a multiset* **assumes** *M-not-empty*: $M \neq \{\#\}$
begin

lift-definition *pmf-of-multiset* :: *'a pmf* **is** λ*x*. *count M x* / *size M*
proof
show (∫⁺ *x*. *ennreal* (*real* (*count M x*) / *real* (*size M*)) ∂*count-space UNIV*) = *1*
using *M-not-empty*
by (*simp add*: *zero-less-divide-iff nn-integral-count-space nonempty-has-size*
sum-divide-distrib[*symmetric*])
(*auto simp*: *size-multiset-overloaded-eq intro*!: *sum.cong*)
qed *simp*

lemma *pmf-of-multiset*[*simp*]: *pmf pmf-of-multiset x* = *count M x* / *size M*

by *transfer rule*

lemma *set-pmf-of-multiset*[*simp*]: *set-pmf pmf-of-multiset = set-mset M*
by (*auto simp*: *set-pmf-iff*)

end

20.6.4 Uniform Distribution

context
fixes *S* :: *'a set* **assumes** *S-not-empty*: *S* ≠ {} **and** *S-finite*: *finite S*
begin

lift-definition *pmf-of-set* :: *'a pmf* **is** λ*x*. *indicator S x / card S*
proof
show (∫⁺ *x*. *ennreal* (*indicator S x / real* (*card S*)) ∂*count-space UNIV*) = *1*
using *S-not-empty S-finite*
by (*subst nn-integral-count-space'*[*of S*])
(*auto simp*: *ennreal-of-nat-eq-real-of-nat ennreal-mult*[*symmetric*])
qed *simp*

lemma *pmf-of-set*[*simp*]: *pmf pmf-of-set x = indicator S x / card S*
by *transfer rule*

lemma *set-pmf-of-set*[*simp*]: *set-pmf pmf-of-set = S*
using *S-finite S-not-empty* **by** (*auto simp*: *set-pmf-iff*)

lemma *emeasure-pmf-of-set-space*[*simp*]: *emeasure pmf-of-set S = 1*
by (*rule measure-pmf.emeasure-eq-1-AE*) (*auto simp*: *AE-measure-pmf-iff*)

lemma *nn-integral-pmf-of-set*: *nn-integral* (*measure-pmf pmf-of-set*) *f* = *sum f S / card S*
by (*subst nn-integral-measure-pmf-finite*)
(*simp-all add*: *sum-distrib-right*[*symmetric*] *card-gt-0-iff S-not-empty S-finite divide-ennreal-def*
divide-ennreal[*symmetric*] *ennreal-of-nat-eq-real-of-nat*[*symmetric*] *ennreal-times-divide*)

lemma *integral-pmf-of-set*: *integral*^{L} (*measure-pmf pmf-of-set*) *f* = *sum f S / card S*
by (*subst integral-measure-pmf*[*of S*]) (*auto simp*: *S-finite sum-divide-distrib*)

lemma *emeasure-pmf-of-set*: *emeasure* (*measure-pmf pmf-of-set*) *A* = *card* (*S* ∩ *A*) / *card S*
by (*subst nn-integral-indicator*[*symmetric*], *simp*)
(*simp add*: *S-finite S-not-empty card-gt-0-iff indicator-def sum.If-cases divide-ennreal*
ennreal-of-nat-eq-real-of-nat nn-integral-pmf-of-set)

lemma *measure-pmf-of-set*: *measure* (*measure-pmf pmf-of-set*) *A* = *card* (*S* ∩ *A*) / *card S*
using *emeasure-pmf-of-set*[*of A*]
by (*simp add*: *measure-nonneg measure-pmf.emeasure-eq-measure*)

end

lemma *pmf-expectation-bind-pmf-of-set*:
fixes *A* :: *'a set* **and** *f* :: *'a* ⇒ *'b pmf*
and *h* :: *'b* ⇒ *'c*::{*banach*, *second-countable-topology*}
assumes *A* ≠ {} *finite A* ⋀*x*. *x* ∈ *A* ⟹ *finite* (*set-pmf* (*f x*))
shows *measure-pmf.expectation* (*pmf-of-set A* ⤜ *f*) *h* =
(∑ *a*∈*A*. *measure-pmf.expectation* (*f a*) *h* /$_R$ *real* (*card A*))
using *assms* **by** (*subst pmf-expectation-bind*[*of A*]) (*auto simp*: *field-split-simps*)

lemma *map-pmf-of-set*:
assumes *finite A A* ≠ {}
shows *map-pmf f* (*pmf-of-set A*) = *pmf-of-multiset* (*image-mset f* (*mset-set A*))
(**is** *?lhs* = *?rhs*)
proof (*intro pmf-eqI*)
fix *x*
from *assms* **have** *ennreal* (*pmf ?lhs x*) = *ennreal* (*pmf ?rhs x*)
by (*subst ennreal-pmf-map*)
(*simp-all add*: *emeasure-pmf-of-set mset-set-empty-iff count-image-mset Int-commute*)
thus *pmf ?lhs x* = *pmf ?rhs x* **by** *simp*
qed

lemma *pmf-bind-pmf-of-set*:
assumes *A* ≠ {} *finite A*
shows *pmf* (*bind-pmf* (*pmf-of-set A*) *f*) *x* =
(∑ *xa*∈*A*. *pmf* (*f xa*) *x*) / *real-of-nat* (*card A*) (**is** *?lhs* = *?rhs*)
proof −
from *assms* **have** *card A* > *0* **by** *auto*
with *assms* **have** *ennreal ?lhs* = *ennreal ?rhs*
by (*subst ennreal-pmf-bind*)
(*simp-all add*: *nn-integral-pmf-of-set max-def pmf-nonneg divide-ennreal* [*symmetric*]
sum-nonneg ennreal-of-nat-eq-real-of-nat)
thus *?thesis* **by** (*subst* (*asm*) *ennreal-inj*) (*auto intro*!: *sum-nonneg divide-nonneg-nonneg*)
qed

lemma *pmf-of-set-singleton*: *pmf-of-set* {*x*} = *return-pmf x*
by(*rule pmf-eqI*)(*simp add*: *indicator-def*)

lemma *map-pmf-of-set-inj*:
assumes *f*: *inj-on f A*
and [*simp*]: *A* ≠ {} *finite A*

shows *map-pmf f* (*pmf-of-set A*) = *pmf-of-set* (*f ' A*) (**is** *?lhs* = *?rhs*)
proof(*rule pmf-eqI*)
fix *i*
show *pmf ?lhs i* = *pmf ?rhs i*
proof(*cases i* ∈ *f ' A*)
case *True*
then obtain *i'* **where** *i* = *f i' i'* ∈ *A* **by** *auto*
thus *?thesis* **using** *f* **by**(*simp add*: *card-image pmf-map-inj*)
next
case *False*
hence *pmf ?lhs i* = *0* **by**(*simp add*: *pmf-eq-0-set-pmf set-map-pmf*)
moreover have *pmf ?rhs i* = *0* **using** *False* **by** *simp*
ultimately show *?thesis* **by** *simp*
qed
qed

lemma *map-pmf-of-set-bij-betw*:
assumes *bij-betw f A B A* ≠ {} *finite A*
shows *map-pmf f* (*pmf-of-set A*) = *pmf-of-set B*
proof −
have *map-pmf f* (*pmf-of-set A*) = *pmf-of-set* (*f ' A*)
by (*intro map-pmf-of-set-inj assms bij-betw-imp-inj-on*[*OF assms*(*1*)])
also from *assms* **have** *f ' A* = *B* **by** (*simp add*: *bij-betw-def*)
finally show *?thesis* **.**
qed

Choosing an element uniformly at random from the union of a disjoint family of finite non-empty sets with the same size is the same as first choosing a set from the family uniformly at random and then choosing an element from the chosen set uniformly at random.

lemma *pmf-of-set-UN*:
assumes *finite* (⋃(*f ' A*)) *A* ≠ {} ⋀*x*. *x* ∈ *A* ⟹ *f x* ≠ {}
⋀*x*. *x* ∈ *A* ⟹ *card* (*f x*) = *n disjoint-family-on f A*
shows *pmf-of-set* (⋃(*f ' A*)) = *do* {*x* ← *pmf-of-set A*; *pmf-of-set* (*f x*)}
(**is** *?lhs* = *?rhs*)
proof (*intro pmf-eqI*)
fix *x*
from *assms* **have** [*simp*]: *finite A*
using *infinite-disjoint-family-imp-infinite-UNION*[*of A f*] **by** *blast*
from *assms* **have** *ereal* (*pmf* (*pmf-of-set* (⋃(*f ' A*))) *x*) =
ereal (*indicator* (⋃*x*∈*A*. *f x*) *x* / *real* (*card* (⋃*x*∈*A*. *f x*)))
by (*subst pmf-of-set*) *auto*
also from *assms* **have** *card* (⋃*x*∈*A*. *f x*) = *card A* ∗ *n*
by (*subst card-UN-disjoint*) (*auto simp*: *disjoint-family-on-def*)
also from *assms*
have *indicator* (⋃*x*∈*A*. *f x*) *x* / *real* ... =
indicator (⋃*x*∈*A*. *f x*) *x* / (*n* ∗ *real* (*card A*))
by (*simp add*: *sum-divide-distrib* [*symmetric*] *mult-ac*)
also from *assms* **have** *indicator* (⋃*x*∈*A*. *f x*) *x* = (∑ *y*∈*A*. *indicator* (*f y*) *x*)

by (*intro indicator-UN-disjoint*) *simp-all*
also from *assms* **have** *ereal* ((∑ *y*∈*A*. *indicator* (*f y*) *x*) / (*real n* ∗ *real* (*card A*))) =
ereal (*pmf ?rhs x*)
by (*subst pmf-bind-pmf-of-set*) (*simp-all add*: *sum-divide-distrib*)
finally show *pmf ?lhs x* = *pmf ?rhs x* **by** *simp*
qed

lemma *bernoulli-pmf-half-conv-pmf-of-set*: *bernoulli-pmf* (*1 / 2*) = *pmf-of-set UNIV*
by (*rule pmf-eqI*) *simp-all*

20.6.5 Poisson Distribution

context
fixes *rate* :: *real* **assumes** *rate-pos*: *0* < *rate*
begin

lift-definition *poisson-pmf* :: *nat pmf* **is** λ*k*. *rate* ^ *k* / *fact k* ∗ *exp* (−*rate*)
proof
have *summable*: *summable* (λ*x*::*nat*. *rate* ^ *x* / *fact x*) **using** *summable-exp*
by (*simp add*: *field-simps divide-inverse* [*symmetric*])
have (∫⁺(*x*::*nat*). *rate* ^ *x* / *fact x* ∗ *exp* (−*rate*) ∂*count-space UNIV*) =
exp (−*rate*) ∗ (∫⁺(*x*::*nat*). *rate* ^ *x* / *fact x* ∂*count-space UNIV*)
by (*simp add*: *field-simps nn-integral-cmult*[*symmetric*] *ennreal-mult'*[*symmetric*])
also from *rate-pos* **have** (∫⁺(*x*::*nat*). *rate* ^ *x* / *fact x* ∂*count-space UNIV*) = (∑ *x*. *rate* ^ *x* / *fact x*)
by (*simp-all add*: *nn-integral-count-space-nat suminf-ennreal summable ennreal-suminf-neq-top*)
also have ... = *exp rate* **unfolding** *exp-def*
by (*simp add*: *field-simps divide-inverse* [*symmetric*])
also have *ennreal* (*exp* (−*rate*)) ∗ *ennreal* (*exp rate*) = *1*
by (*simp add*: *mult-exp-exp ennreal-mult*[*symmetric*])
finally show (∫⁺ *x*. *ennreal* (*rate* ^ *x* / (*fact x*) ∗ *exp* (− *rate*)) ∂*count-space UNIV*) = *1* .
qed (*simp add*: *rate-pos*[*THEN less-imp-le*])

lemma *pmf-poisson*[*simp*]: *pmf poisson-pmf k* = *rate* ^ *k* / *fact k* ∗ *exp* (−*rate*)
by *transfer rule*

lemma *set-pmf-poisson*[*simp*]: *set-pmf poisson-pmf* = *UNIV*
using *rate-pos* **by** (*auto simp*: *set-pmf-iff*)

end

20.6.6 Binomial Distribution

context
fixes *n* :: *nat* **and** *p* :: *real* **assumes** *p-nonneg*: *0* ≤ *p* **and** *p-le-1*: *p* ≤ *1*
begin

lift-definition *binomial-pmf* :: *nat pmf* **is** $\lambda k.$ (*n choose k*) $*$ *p*^*k* $*$ (*1* $-$ *p*)^(*n* $-$ *k*)
proof
have ($\int^+ k.$ *ennreal* (*real* (*n choose k*) $*$ *p* ^ *k* $*$ (*1* $-$ *p*) ^ (*n* $-$ *k*)) ∂*count-space UNIV*) $=$
ennreal ($\sum k{\leq}n.$ *real* (*n choose k*) $*$ *p* ^ *k* $*$ (*1* $-$ *p*) ^ (*n* $-$ *k*))
using *p-le-1 p-nonneg* **by** (*subst nn-integral-count-space′*) *auto*
also have ($\sum k{\leq}n.$ *real* (*n choose k*) $*$ *p* ^ *k* $*$ (*1* $-$ *p*) ^ (*n* $-$ *k*)) $=$ (*p* $+$ (*1* $-$ *p*)) ^ *n*
by (*subst binomial-ring*) (*simp add*: *atLeast0AtMost*)
finally show ($\int^+ x.$ *ennreal* (*real* (*n choose x*) $*$ *p* ^ *x* $*$ (*1* $-$ *p*) ^ (*n* $-$ *x*)) ∂*count-space UNIV*) $=$ *1*
by *simp*
qed (*insert p-nonneg p-le-1*, *simp*)

lemma *pmf-binomial*[*simp*]: *pmf binomial-pmf k* $=$ (*n choose k*) $*$ *p*^*k* $*$ (*1* $-$ *p*)^(*n* $-$ *k*)
by *transfer rule*

lemma *set-pmf-binomial-eq*: *set-pmf binomial-pmf* $=$ (*if p* $=$ *0 then* {*0*} *else if p* $=$ *1 then* {*n*} *else* {.. *n*})
using *p-nonneg p-le-1* **unfolding** *set-eq-iff set-pmf-iff pmf-binomial* **by** (*auto simp*: *set-pmf-iff*)

end

end

lemma *set-pmf-binomial-0*[*simp*]: *set-pmf* (*binomial-pmf n 0*) $=$ {*0*}
by (*simp add*: *set-pmf-binomial-eq*)

lemma *set-pmf-binomial-1*[*simp*]: *set-pmf* (*binomial-pmf n 1*) $=$ {*n*}
by (*simp add*: *set-pmf-binomial-eq*)

lemma *set-pmf-binomial*[*simp*]: *0* $<$ *p* $\Longrightarrow$ *p* $<$ *1* $\Longrightarrow$ *set-pmf* (*binomial-pmf n p*) $=$ {*..n*}
by (*simp add*: *set-pmf-binomial-eq*)

lemma *finite-set-pmf-binomial-pmf* [*intro*]: *p* $\in$ {*0..1*} $\Longrightarrow$ *finite* (*set-pmf* (*binomial-pmf n p*))
by (*subst set-pmf-binomial-eq*) *auto*

lemma *expectation-binomial-pmf′*:
fixes *f* :: *nat* $\Rightarrow$ *'a* :: {*banach*, *second-countable-topology*}
assumes *p*: *p* $\in$ {*0..1*}
shows *measure-pmf.expectation* (*binomial-pmf n p*) *f* $=$
($\sum k{\leq}n.$ (*real* (*n choose k*) $*$ *p* ^ *k* $*$ (*1* $-$ *p*) ^ (*n* $-$ *k*)) $*_R$ *f k*)
using *p* **by** (*subst integral-measure-pmf*[**where** *A* $=$ {*..n*}])
(*auto simp*: *set-pmf-binomial-eq split*: *if-splits*)

lemma *integrable-binomial-pmf* [*simp*, *intro*]:
fixes *f* :: *nat* ⇒ *'a* :: {*banach*, *second-countable-topology*}
assumes *p*: *p* ∈ {*0..1*}
shows *integrable* (*binomial-pmf n p*) *f*
by (*rule integrable-measure-pmf-finite*) (*use assms* **in** *auto*)

context includes *lifting-syntax*
begin

lemma *bind-pmf-parametric* [*transfer-rule*]:
(*rel-pmf A* ===> (*A* ===> *rel-pmf B*) ===> *rel-pmf B*) *bind-pmf bind-pmf*
by(*blast intro*: *rel-pmf-bindI dest*: *rel-funD*)

lemma *return-pmf-parametric* [*transfer-rule*]: (*A* ===> *rel-pmf A*) *return-pmf return-pmf*
by(*rule rel-funI*) *simp*

end

primrec *replicate-pmf* :: *nat* ⇒ *'a pmf* ⇒ *'a list pmf* **where**
replicate-pmf 0 - = return-pmf []
| *replicate-pmf* (*Suc n*) *p* = *do* {*x* ← *p*; *xs* ← *replicate-pmf n p*; *return-pmf* (*x*#*xs*)}

lemma *replicate-pmf-1*: *replicate-pmf 1 p* = *map-pmf* (λ*x*. [*x*]) *p*
by (*simp add*: *map-pmf-def bind-return-pmf*)

lemma *set-replicate-pmf*:
set-pmf (*replicate-pmf n p*) = {*xs*∈*lists* (*set-pmf p*). *length xs* = *n*}
by (*induction n*) (*auto simp*: *length-Suc-conv*)

lemma *replicate-pmf-distrib*:
replicate-pmf (*m* + *n*) *p* =
do {*xs* ← *replicate-pmf m p*; *ys* ← *replicate-pmf n p*; *return-pmf* (*xs* @ *ys*)}
by (*induction m*) (*simp-all add*: *bind-return-pmf bind-return-pmf' bind-assoc-pmf*)

lemma *power-diff'*:
assumes *b* ≤ *a*
shows *x* ^ (*a* − *b*) = (*if x* = *0* ∧ *a* = *b then 1 else x* ^ *a* / (*x*::*'a*::*field*) ^ *b*)
proof (*cases x* = *0*)
case *True*
with *assms* **show** *?thesis* **by** (*cases a* − *b*) *simp-all*
qed (*insert assms*, *simp-all add*: *power-diff*)

lemma *binomial-pmf-Suc*:
assumes *p* ∈ {*0..1*}
shows *binomial-pmf* (*Suc n*) *p* =

```
        do {b ← bernoulli-pmf p;
            k ← binomial-pmf n p;
            return-pmf ((if b then 1 else 0) + k)} (is - = ?rhs)
proof (intro pmf-eqI)
  fix k
  have A: indicator {Suc a} (Suc b) = indicator {a} b for a b
    by (simp add: indicator-def)
  show pmf (binomial-pmf (Suc n) p) k = pmf ?rhs k
    by (cases k; cases k > n)
       (insert assms, auto simp: pmf-bind measure-pmf-single A field-split-simps
algebra-simps
        not-less less-eq-Suc-le [symmetric] power-diff')
qed

lemma binomial-pmf-0: p ∈ {0..1} ⟹ binomial-pmf 0 p = return-pmf 0
  by (rule pmf-eqI) (simp-all add: indicator-def)

lemma binomial-pmf-altdef:
  assumes p ∈ {0..1}
  shows   binomial-pmf n p = map-pmf (length ∘ filter id) (replicate-pmf n
(bernoulli-pmf p))
  by (induction n)
       (insert assms, auto simp: binomial-pmf-Suc map-pmf-def bind-return-pmf
bind-assoc-pmf
        bind-return-pmf' binomial-pmf-0 intro!: bind-pmf-cong)
```

20.7 Negative Binomial distribution

The negative binomial distribution counts the number of times a weighted coin comes up tails before having come up heads n times. In other words: how many failures do we see before seeing the n-th success?

An alternative view is that the negative binomial distribution is the sum of n i.i.d. geometric variables (this is the definition that we use).

Note that there are sometimes different conventions for this distributions in the literature; for instance, sometimes the number of *attempts* is counted instead of the number of failures. This only shifts the entire distribution by a constant number and is thus not a big difference. I think that the convention we use is the most natural one since the support of the distribution starts at 0, whereas for the other convention it starts at n.

```
primrec neg-binomial-pmf :: nat ⇒ real ⇒ nat pmf where
  neg-binomial-pmf 0 p = return-pmf 0
| neg-binomial-pmf (Suc n) p =
    map-pmf (λ(x,y). (x + y)) (pair-pmf (geometric-pmf p) (neg-binomial-pmf n
p))

lemma neg-binomial-pmf-Suc-0 [simp]: neg-binomial-pmf (Suc 0) p = geomet-
ric-pmf p
```

by (*auto simp*: *pair-pmf-def bind-return-pmf map-pmf-def bind-assoc-pmf bind-return-pmf'*)

lemmas *neg-binomial-pmf-Suc* [*simp del*] = *neg-binomial-pmf.simps*(*2*)

lemma *neg-binomial-prob-1* [*simp*]: *neg-binomial-pmf n 1* = *return-pmf 0*
by (*induction n*) (*simp-all add*: *neg-binomial-pmf-Suc*)

We can now show the aforementioned intuition about counting the failures before the n-th success with the following recurrence:

lemma *neg-binomial-pmf-unfold*:
assumes *p*: *p* ∈ {*0*<..*1*}
shows *neg-binomial-pmf* (*Suc n*) *p* =
do {*b* ← *bernoulli-pmf p*;
if b then neg-binomial-pmf n p else map-pmf Suc (*neg-binomial-pmf* (*Suc n*) *p*)}
(**is** - = *?rhs*)
unfolding *neg-binomial-pmf-Suc*
by (*subst geometric-bind-pmf-unfold*[*OF p*])
(*auto simp*: *map-pmf-def pair-pmf-def bind-assoc-pmf bind-return-pmf bind-return-pmf'*
intro!: *bind-pmf-cong*)

Next, we show an explicit formula for the probability mass function of the negative binomial distribution:

lemma *pmf-neg-binomial*:
assumes *p*: *p* ∈ {*0*<..*1*}
shows *pmf* (*neg-binomial-pmf n p*) *k* = *real* ((*k* + *n* − *1*) *choose k*) ∗ *p* ^ *n* ∗ (*1* − *p*) ^ *k*
proof (*induction n arbitrary*: *k*)
case *0*
thus *?case* **using** *assms* **by** (*auto simp*: *indicator-def*)
next
case (*Suc n*)
show *?case*
proof (*cases n* = *0*)
case *True*
thus *?thesis* **using** *assms* **by** *auto*
next
case *False*
let *?f* = *pmf* (*neg-binomial-pmf n p*)
have *pmf* (*neg-binomial-pmf* (*Suc n*) *p*) *k* =
pmf (*geometric-pmf p* ⤜ (λ*x*. *map-pmf* ((+) *x*) (*neg-binomial-pmf n p*))) *k*
by (*auto simp*: *pair-pmf-def bind-return-pmf map-pmf-def bind-assoc-pmf neg-binomial-pmf-Suc*)
also have ... = *measure-pmf.expectation* (*geometric-pmf p*)
(λ*x*. *measure-pmf.prob* (*neg-binomial-pmf n p*) ((+) *x* −‘ {*k*}))
by (*simp add*: *pmf-bind pmf-map*)
also have (λ*x*. (+) *x* −‘ {*k*}) = (λ*x*. *if x* ≤ *k then* {*k* − *x*} *else* {})
by (*auto simp*: *fun-eq-iff*)

```
  also have (λx. measure-pmf.prob (neg-binomial-pmf n p) (... x)) =
            (λx. if x ≤ k then ?f(k − x) else 0)
    by (auto simp: fun-eq-iff measure-pmf-single)
  also have measure-pmf.expectation (geometric-pmf p) ... =
              (∑ i≤k. pmf (neg-binomial-pmf n p) (k − i) * pmf (geometric-pmf
p) i)
    by (subst integral-measure-pmf-real[where A = {..k}]) (auto split: if-splits)
  also have ... = p^(n+1) * (1−p)^k * real (∑ i≤k. (k − i + n − 1) choose
(k − i))
    unfolding sum-distrib-left of-nat-sum
  proof (intro sum.cong refl, goal-cases)
    case (1 i)
    have pmf (neg-binomial-pmf n p) (k − i) * pmf (geometric-pmf p) i =
           real ((k − i + n − 1) choose (k − i)) * p^(n+1) * ((1−p)^(k−i) *
(1−p)^i)
      using assms Suc.IH by (simp add: mult-ac)
    also have (1−p)^(k−i) * (1−p)^i = (1−p)^k
      using 1 by (subst power-add [symmetric]) auto
    finally show ?case by simp
  qed
  also have (∑ i≤k. (k − i + n − 1) choose (k − i)) = (∑ i≤k. (n − 1 + i)
choose i)
    by (intro sum.reindex-bij-witness[of - λi. k − i λi. k − i])
      (use ‹n ≠ 0› in ‹auto simp: algebra-simps›)
  also have ... = (n + k) choose k
    by (subst sum-choose-lower) (use ‹n ≠ 0› in auto)
  finally show ?thesis
    by (simp add: add-ac)
 qed
qed


lemma gbinomial-0-left: 0 gchoose k = (if k = 0 then 1 else 0)
 by (cases k) auto
```

The following alternative formula highlights why it is called 'negative binomial distribution':

```
lemma pmf-neg-binomial':
 assumes p: p ∈ {0<..1}
 shows   pmf (neg-binomial-pmf n p) k = (−1) ^ k * ((−real n) gchoose k) * p
^ n * (1 − p) ^ k
proof (cases n > 0)
 case n: True
 have pmf (neg-binomial-pmf n p) k = real ((k + n − 1) choose k) * p ^ n * (1
− p) ^ k
   by (rule pmf-neg-binomial) fact+
 also have real ((k + n − 1) choose k) = ((real k + real n − 1) gchoose k)
   using n by (subst binomial-gbinomial) (auto simp: of-nat-diff)
 also have ... = (−1) ^ k * ((−real n) gchoose k)
```

by (*subst gbinomial-negated-upper*) *auto*
finally show *?thesis* **by** *simp*
qed (*auto simp*: *indicator-def gbinomial-0-left*)

The cumulative distribution function of the negative binomial distribution can be expressed in terms of that of the 'normal' binomial distribution.

lemma *prob-neg-binomial-pmf-atMost*:
assumes *p*: $p \in \{0<..1\}$
shows *measure-pmf.prob* (*neg-binomial-pmf n p*) $\{..k\}$ =
measure-pmf.prob (*binomial-pmf* $(n + k)$ $(1 - p)$) $\{..k\}$
proof (*cases* $n = 0$)
case [*simp*]: *True*
have *set-pmf* (*binomial-pmf* $(n + k)$ $(1 - p)$) $\subseteq \{..n+k\}$
using *p* **by** (*subst set-pmf-binomial-eq*) *auto*
hence *measure-pmf.prob* (*binomial-pmf* $(n + k)$ $(1 - p)$) $\{..k\} = 1$
by (*subst measure-pmf.prob-eq-1*) (*auto intro*!: *AE-pmfI*)
thus *?thesis* **by** *simp*
next
case *False*
hence *n*: $n > 0$ **by** *auto*
have *measure-pmf.prob* (*binomial-pmf* $(n + k)$ $(1 - p)$) $\{..k\} = (\sum i{\le}k.$ *pmf* (*binomial-pmf* $(n + k)$ $(1 - p)$) $i)$
by (*intro measure-measure-pmf-finite*) *auto*
also have $\ldots = (\sum i{\le}k.$ *real* $((n + k)$ *choose* $i) * p \,\hat{}\, (n + k - i) * (1 - p) \,\hat{}\, i)$
using *p* **by** (*simp add*: *mult-ac*)
also have $\ldots = p \,\hat{}\, n * (\sum i{\le}k.$ *real* $((n + k)$ *choose* $i) * (1 - p) \,\hat{}\, i * p \,\hat{}\, (k - i))$
unfolding *sum-distrib-left* **by** (*intro sum.cong*) (*auto simp*: *algebra-simps simp flip*: *power-add*)
also have $(\sum i{\le}k.$ *real* $((n + k)$ *choose* $i) * (1 - p) \,\hat{}\, i * p \,\hat{}\, (k - i)) =$
$(\sum i{\le}k.$ $((n + i - 1)$ *choose* $i) * (1 - p) \,\hat{}\, i)$
using *gbinomial-partial-sum-poly-xpos*[*of k real n 1 − p p*] *n*
by (*simp add*: *binomial-gbinomial add-ac of-nat-diff*)
also have $p \,\hat{}\, n * \ldots = (\sum i{\le}k.$ *pmf* (*neg-binomial-pmf n p*) $i)$
using *p* **unfolding** *sum-distrib-left* **by** (*simp add*: *pmf-neg-binomial algebra-simps*)
also have $\ldots =$ *measure-pmf.prob* (*neg-binomial-pmf n p*) $\{..k\}$
by (*intro measure-measure-pmf-finite* [*symmetric*]) *auto*
finally show *?thesis* **..**
qed

lemma *prob-neg-binomial-pmf-lessThan*:
assumes *p*: $p \in \{0<..1\}$
shows *measure-pmf.prob* (*neg-binomial-pmf n p*) $\{..<k\}$ =
measure-pmf.prob (*binomial-pmf* $(n + k - 1)$ $(1 - p)$) $\{..<k\}$
proof (*cases* $k = 0$)
case *False*
hence $\{..<k\} = \{..k-1\}$

by *auto*
thus *?thesis*
using *prob-neg-binomial-pmf-atMost*[*OF p*, *of n k − 1*] *False* **by** *simp*
qed *auto*

The expected value of the negative binomial distribution is $n(1-p)/p$:

lemma *nn-integral-neg-binomial-pmf-real*:
assumes *p*: $p \in \{0<..1\}$
shows *nn-integral* (*measure-pmf* (*neg-binomial-pmf n p*)) *of-nat* = *ennreal* (*n* ∗ (*1* − *p*) / *p*)
proof (*induction n*)
case *0*
thus *?case* **by** *auto*
next
case (*Suc n*)
have *nn-integral* (*measure-pmf* (*neg-binomial-pmf* (*Suc n*) *p*)) *of-nat* =
nn-integral (*measure-pmf* (*geometric-pmf p*)) *of-nat* +
nn-integral (*measure-pmf* (*neg-binomial-pmf n p*)) *of-nat*
by (*simp add*: *neg-binomial-pmf-Suc case-prod-unfold nn-integral-add nn-integral-pair-pmf'*)
also have *nn-integral* (*measure-pmf* (*geometric-pmf p*)) *of-nat* = *ennreal* ((*1*−*p*) / *p*)
unfolding *ennreal-of-nat-eq-real-of-nat*
using *expectation-geometric-pmf*[*OF p*] *integrable-real-geometric-pmf*[*OF p*]
by (*subst nn-integral-eq-integral*) *auto*
also have *nn-integral* (*measure-pmf* (*neg-binomial-pmf n p*)) *of-nat* = *n* ∗ (*1* − *p*) / *p* **using** *p*
by (*subst Suc.IH*)
(*auto simp*: *ennreal-of-nat-eq-real-of-nat ennreal-mult simp flip*: *divide-ennreal ennreal-minus*)
also have *ennreal* ((*1* − *p*) / *p*) + *ennreal* (*real n* ∗ (*1* − *p*) / *p*) =
ennreal ((*1*−*p*) / *p* + *real n* ∗ (*1* − *p*) / *p*)
by (*intro ennreal-plus* [*symmetric*] *divide-nonneg-pos mult-nonneg-nonneg*) (*use p* **in** *auto*)
also have (*1*−*p*) / *p* + *real n* ∗ (*1* − *p*) / *p* = *real* (*Suc n*) ∗ (*1* − *p*) / *p*
using *p* **by** (*auto simp*: *field-simps*)
finally show *?case*
by (*simp add*: *ennreal-of-nat-eq-real-of-nat*)
qed

lemma *integrable-neg-binomial-pmf-real*:
assumes *p*: $p \in \{0<..1\}$
shows *integrable* (*measure-pmf* (*neg-binomial-pmf n p*)) *real*
using *nn-integral-neg-binomial-pmf-real*[*OF p*, *of n*]
by (*subst integrable-iff-bounded*) (*auto simp flip*: *ennreal-of-nat-eq-real-of-nat*)

lemma *expectation-neg-binomial-pmf*:
assumes *p*: $p \in \{0<..1\}$
shows *measure-pmf.expectation* (*neg-binomial-pmf n p*) *real* = *n* ∗ (*1* − *p*) / *p*
proof −

have *nn-integral* (*measure-pmf* (*neg-binomial-pmf n p*)) *of-nat* = *ennreal* (*n* ∗ (*1* − *p*) / *p*)
by (*intro nn-integral-neg-binomial-pmf-real p*)
also have *of-nat* = (λ*x*. *ennreal* (*real x*))
by (*simp add*: *ennreal-of-nat-eq-real-of-nat fun-eq-iff*)
finally show *?thesis*
using *p* **by** (*subst* (*asm*) *nn-integral-eq-integrable*) *auto*
qed

20.8 PMFs from association lists

definition *pmf-of-list* :: (*'a* × *real*) *list* ⇒ *'a pmf* **where**
pmf-of-list xs = *embed-pmf* (λ*x*. *sum-list* (*map snd* (*filter* (λ*z*. *fst z* = *x*) *xs*)))

definition *pmf-of-list-wf* **where**
pmf-of-list-wf xs ⟷ (∀ *x*∈*set* (*map snd xs*) . *x* ≥ *0*) ∧ *sum-list* (*map snd xs*) = *1*

lemma *pmf-of-list-wfI*:
(⋀*x*. *x* ∈ *set* (*map snd xs*) ⟹ *x* ≥ *0*) ⟹ *sum-list* (*map snd xs*) = *1* ⟹ *pmf-of-list-wf xs*
unfolding *pmf-of-list-wf-def* **by** *simp*

context
begin

private lemma *pmf-of-list-aux*:
assumes ⋀*x*. *x* ∈ *set* (*map snd xs*) ⟹ *x* ≥ *0*
assumes *sum-list* (*map snd xs*) = *1*
shows (∫⁺ *x*. *ennreal* (*sum-list* (*map snd* [*z*←*xs* . *fst z* = *x*])) ∂*count-space UNIV*) = *1*
proof −
have (∫⁺ *x*. *ennreal* (*sum-list* (*map snd* (*filter* (λ*z*. *fst z* = *x*) *xs*))) ∂*count-space UNIV*) =
(∫⁺ *x*. *ennreal* (*sum-list* (*map* (λ(*x'*,*p*). *indicator* {*x'*} *x* ∗ *p*) *xs*)) ∂*count-space UNIV*)
apply (*intro nn-integral-cong ennreal-cong*, *subst sum-list-map-filter'*)
apply (*rule arg-cong*[**where** *f* = *sum-list*])
apply (*auto cong*: *map-cong*)
done
also have ... = (∑ (*x'*,*p*)←*xs*. (∫⁺ *x*. *ennreal* (*indicator* {*x'*} *x* ∗ *p*) ∂*count-space UNIV*))
using *assms*(*1*)
proof (*induction xs*)
case (*Cons x xs*)
from *Cons.prems* **have** *snd x* ≥ *0* **by** *simp*
moreover have *b* ≥ *0* **if** (*a*,*b*) ∈ *set xs* **for** *a b*
using *Cons.prems*[*of b*] *that* **by** *force*
ultimately have (∫⁺ *y*. *ennreal* (∑ (*x'*, *p*)←*x* # *xs*. *indicator* {*x'*} *y* ∗ *p*)

∂count-space UNIV) =
(∫⁺ *y*. *ennreal* (*indicator* {*fst x*} *y* ∗ *snd x*) +
ennreal (∑ (*x'*, *p*)←*xs*. *indicator* {*x'*} *y* ∗ *p*) *∂count-space UNIV*)
by (*intro nn-integral-cong*, *subst ennreal-plus* [*symmetric*])
(*auto simp*: *case-prod-unfold indicator-def intro*!: *sum-list-nonneg*)
also have ... = (∫⁺ *y*. *ennreal* (*indicator* {*fst x*} *y* ∗ *snd x*) *∂count-space UNIV*) +
(∫⁺ *y*. *ennreal* (∑ (*x'*, *p*)←*xs*. *indicator* {*x'*} *y* ∗ *p*) *∂count-space UNIV*)
by (*intro nn-integral-add*)
(*force intro*!: *sum-list-nonneg AE-I2 intro*: *Cons simp*: *indicator-def*)+
also have (∫⁺ *y*. *ennreal* (∑ (*x'*, *p*)←*xs*. *indicator* {*x'*} *y* ∗ *p*) *∂count-space UNIV*) =
(∑ (*x'*, *p*)←*xs*. (∫⁺ *y*. *ennreal* (*indicator* {*x'*} *y* ∗ *p*) *∂count-space UNIV*))
using *Cons*(*1*) **by** (*intro Cons*) *simp-all*
finally show *?case* **by** (*simp add*: *case-prod-unfold*)
qed *simp*
also have ... = (∑ (*x'*,*p*)←*xs*. *ennreal p* ∗ (∫⁺ *x*. *indicator* {*x'*} *x ∂count-space UNIV*))
using *assms*(*1*)
by (*simp cong*: *map-cong only*: *case-prod-unfold*, *subst nn-integral-cmult* [*symmetric*])
(*auto intro*!: *assms*(*1*) *simp*: *max-def times-ereal.simps* [*symmetric*] *mult-ac ereal-indicator*
simp del: *times-ereal.simps*)+
also from *assms* **have** ... = *sum-list* (*map snd xs*) **by** (*simp add*: *case-prod-unfold sum-list-ennreal*)
also have ... = *1* **using** *assms*(*2*) **by** *simp*
finally show *?thesis* **.**
qed

lemma *pmf-pmf-of-list*:
assumes *pmf-of-list-wf xs*
shows *pmf* (*pmf-of-list xs*) *x* = *sum-list* (*map snd* (*filter* (λ*z*. *fst z* = *x*) *xs*))
using *assms pmf-of-list-aux*[*of xs*] **unfolding** *pmf-of-list-def pmf-of-list-wf-def*
by (*subst pmf-embed-pmf*) (*auto intro*!: *sum-list-nonneg*)

end

lemma *set-pmf-of-list*:
assumes *pmf-of-list-wf xs*
shows *set-pmf* (*pmf-of-list xs*) ⊆ *set* (*map fst xs*)
proof *clarify*
fix *x* **assume** *A*: *x* ∈ *set-pmf* (*pmf-of-list xs*)
show *x* ∈ *set* (*map fst xs*)
proof (*rule ccontr*)
assume *x* ∉ *set* (*map fst xs*)
hence [*z*←*xs* . *fst z* = *x*] = [] **by** (*auto simp*: *filter-empty-conv*)
with *A assms* **show** *False* **by** (*simp add*: *pmf-pmf-of-list set-pmf-eq*)

qed
qed

lemma *finite-set-pmf-of-list*:
assumes *pmf-of-list-wf xs*
shows *finite* (*set-pmf* (*pmf-of-list xs*))
using *assms* **by** (*rule finite-subset*[*OF set-pmf-of-list*]) *simp-all*

lemma *emeasure-Int-set-pmf*:
emeasure (*measure-pmf p*) (*A* ∩ *set-pmf p*) = *emeasure* (*measure-pmf p*) *A*
by (*rule emeasure-eq-AE*) (*auto simp*: *AE-measure-pmf-iff*)

lemma *measure-Int-set-pmf*:
measure (*measure-pmf p*) (*A* ∩ *set-pmf p*) = *measure* (*measure-pmf p*) *A*
using *emeasure-Int-set-pmf*[*of p A*] **by** (*simp add*: *Sigma-Algebra.measure-def*)

lemma *measure-prob-cong-0*:
assumes ⋀*x*. *x* ∈ *A* − *B* ⟹ *pmf p x* = *0*
assumes ⋀*x*. *x* ∈ *B* − *A* ⟹ *pmf p x* = *0*
shows *measure* (*measure-pmf p*) *A* = *measure* (*measure-pmf p*) *B*
proof −
have *measure-pmf.prob p A* = *measure-pmf.prob p* (*A* ∩ *set-pmf p*)
by (*simp add*: *measure-Int-set-pmf*)
also have *A* ∩ *set-pmf p* = *B* ∩ *set-pmf p*
using *assms* **by** (*auto simp*: *set-pmf-eq*)
also have *measure-pmf.prob p* ... = *measure-pmf.prob p B*
by (*simp add*: *measure-Int-set-pmf*)
finally show *?thesis* **.**
qed

lemma *emeasure-pmf-of-list*:
assumes *pmf-of-list-wf xs*
shows *emeasure* (*pmf-of-list xs*) *A* = *ennreal* (*sum-list* (*map snd* (*filter* (λ*x*. *fst x* ∈ *A*) *xs*)))
proof −
have *emeasure* (*pmf-of-list xs*) *A* = *nn-integral* (*measure-pmf* (*pmf-of-list xs*)) (*indicator A*)
by *simp*
also from *assms*
have ... = (∑ *x*∈*set-pmf* (*pmf-of-list xs*) ∩ *A*. *ennreal* (*sum-list* (*map snd* [*z*←*xs* . *fst z* = *x*])))
by (*subst nn-integral-measure-pmf-finite*) (*simp-all add*: *finite-set-pmf-of-list pmf-pmf-of-list Int-def*)
also from *assms*
have ... = *ennreal* (∑ *x*∈*set-pmf* (*pmf-of-list xs*) ∩ *A*. *sum-list* (*map snd* [*z*←*xs* . *fst z* = *x*]))
by (*subst sum-ennreal*) (*auto simp*: *pmf-of-list-wf-def intro*!: *sum-list-nonneg*)
also have ... = *ennreal* (∑ *x*∈*set-pmf* (*pmf-of-list xs*) ∩ *A*.
indicator A x ∗ *pmf* (*pmf-of-list xs*) *x*) (**is** - = *ennreal ?S*)

using *assms* **by** (*intro ennreal-cong sum.cong*) (*auto simp*: *pmf-pmf-of-list*)
also have *?S* = ($\sum$ *x*∈*set-pmf* (*pmf-of-list xs*). *indicator A x* ∗ *pmf* (*pmf-of-list xs*) *x*)
using *assms* **by** (*intro sum.mono-neutral-left set-pmf-of-list finite-set-pmf-of-list*) *auto*
also have ... = ($\sum$ *x*∈*set* (*map fst xs*). *indicator A x* ∗ *pmf* (*pmf-of-list xs*) *x*)
using *assms* **by** (*intro sum.mono-neutral-left set-pmf-of-list*) (*auto simp*: *set-pmf-eq*)
also have ... = ($\sum$ *x*∈*set* (*map fst xs*). *indicator A x* ∗
sum-list (*map snd* (*filter* (λ*z*. *fst z* = *x*) *xs*)))
using *assms* **by** (*simp add*: *pmf-pmf-of-list*)
also have ... = ($\sum$ *x*∈*set* (*map fst xs*). *sum-list* (*map snd* (*filter* (λ*z*. *fst z* = *x* ∧ *x* ∈ *A*) *xs*)))
by (*intro sum.cong*) (*auto simp*: *indicator-def*)
also have ... = ($\sum$ *x*∈*set* (*map fst xs*). ($\sum$ *xa* = *0*..<*length xs*.
if fst (*xs* ! *xa*) = *x* ∧ *x* ∈ *A then snd* (*xs* ! *xa*) *else 0*))
by (*intro sum.cong refl*, *subst sum-list-map-filter′*, *subst sum-list-sum-nth*) *simp*
also have ... = ($\sum$ *xa* = *0*..<*length xs*. ($\sum$ *x*∈*set* (*map fst xs*).
if fst (*xs* ! *xa*) = *x* ∧ *x* ∈ *A then snd* (*xs* ! *xa*) *else 0*))
by (*rule sum.swap*)
also have ... = ($\sum$ *xa* = *0*..<*length xs*. *if fst* (*xs* ! *xa*) ∈ *A then*
($\sum$ *x*∈*set* (*map fst xs*). *if x* = *fst* (*xs* ! *xa*) *then snd* (*xs* ! *xa*) *else 0*) *else 0*)
by (*auto intro*!: *sum.cong sum.neutral simp del*: *sum.delta*)
also have ... = ($\sum$ *xa* = *0*..<*length xs*. *if fst* (*xs* ! *xa*) ∈ *A then snd* (*xs* ! *xa*) *else 0*)
by (*intro sum.cong refl*) (*simp-all add*: *sum.delta*)
also have ... = *sum-list* (*map snd* (*filter* (λ*x*. *fst x* ∈ *A*) *xs*))
by (*subst sum-list-map-filter′*, *subst sum-list-sum-nth*) *simp-all*
finally show *?thesis* **.**
qed

lemma *measure-pmf-of-list*:
assumes *pmf-of-list-wf xs*
shows *measure* (*pmf-of-list xs*) *A* = *sum-list* (*map snd* (*filter* (λ*x*. *fst x* ∈ *A*) *xs*))
using *assms* **unfolding** *pmf-of-list-wf-def Sigma-Algebra.measure-def*
by (*subst emeasure-pmf-of-list* [*OF assms*], *subst enn2real-ennreal*) (*auto intro*!: *sum-list-nonneg*)

lemma *sum-list-nonneg-eq-zero-iff*:
fixes *xs* :: *'a* :: *linordered-ab-group-add list*
shows (⋀*x*. *x* ∈ *set xs* ⟹ *x* ≥ *0*) ⟹ *sum-list xs* = *0* ⟷ *set xs* ⊆ {*0*}
proof (*induction xs*)
case (*Cons x xs*)
from *Cons.prems* **have** *sum-list* (*x*#*xs*) = *0* ⟷ *x* = *0* ∧ *sum-list xs* = *0*
unfolding *sum-list-simps* **by** (*subst add-nonneg-eq-0-iff*) (*auto intro*: *sum-list-nonneg*)
with *Cons.IH Cons.prems* **show** *?case* **by** *simp*
qed *simp-all*

lemma *sum-list-filter-nonzero*:
sum-list (*filter* (λ*x*. *x* ≠ *0*) *xs*) = *sum-list xs*
by (*induction xs*) *simp-all*

lemma *set-pmf-of-list-eq*:
assumes *pmf-of-list-wf xs* ⋀*x*. *x* ∈ *snd* ‘ *set xs* ⟹ *x* > *0*
shows *set-pmf* (*pmf-of-list xs*) = *fst* ‘ *set xs*
proof
{
fix *x* **assume** *A*: *x* ∈ *fst* ‘ *set xs* **and** *B*: *x* ∉ *set-pmf* (*pmf-of-list xs*)
then obtain *y* **where** *y*: (*x*, *y*) ∈ *set xs* **by** *auto*
from *B* **have** *sum-list* (*map snd* [*z*←*xs*. *fst z* = *x*]) = *0*
by (*simp add*: *pmf-pmf-of-list*[*OF assms*(*1*)] *set-pmf-eq*)
moreover from *y* **have** *y* ∈ *snd* ‘ {*xa* ∈ *set xs*. *fst xa* = *x*} **by** *force*
ultimately have *y* = *0* **using** *assms*(*1*)
by (*subst* (*asm*) *sum-list-nonneg-eq-zero-iff*) (*auto simp*: *pmf-of-list-wf-def*)
with *assms*(*2*) *y* **have** *False* **by** *force*
}
thus *fst* ‘ *set xs* ⊆ *set-pmf* (*pmf-of-list xs*) **by** *blast*
qed (*insert set-pmf-of-list*[*OF assms*(*1*)], *simp-all*)

lemma *pmf-of-list-remove-zeros*:
assumes *pmf-of-list-wf xs*
defines *xs′* ≡ *filter* (λ*z*. *snd z* ≠ *0*) *xs*
shows *pmf-of-list-wf xs′ pmf-of-list xs′* = *pmf-of-list xs*
proof −
have *map snd* [*z*←*xs* . *snd z* ≠ *0*] = *filter* (λ*x*. *x* ≠ *0*) (*map snd xs*)
by (*induction xs*) *simp-all*
with *assms*(*1*) **show** *wf*: *pmf-of-list-wf xs′*
by (*auto simp*: *pmf-of-list-wf-def xs′-def sum-list-filter-nonzero*)
have *sum-list* (*map snd* [*z*←*xs′* . *fst z* = *i*]) = *sum-list* (*map snd* [*z*←*xs* . *fst z* = *i*]) **for** *i*
unfolding *xs′-def* **by** (*induction xs*) *simp-all*
with *assms*(*1*) *wf* **show** *pmf-of-list xs′* = *pmf-of-list xs*
by (*intro pmf-eqI*) (*simp-all add*: *pmf-pmf-of-list*)
qed

end

21 Code generation for PMFs

theory *PMF-Impl*
imports *Probability-Mass-Function HOL−Library.AList-Mapping*
begin

21.1 General code generation setup

definition *pmf-of-mapping* :: ('*a*, *real*) *mapping* ⇒ '*a pmf* **where**
pmf-of-mapping m = *embed-pmf* (*Mapping.lookup-default 0 m*)

lemma *nn-integral-lookup-default*:
fixes *m* :: ('*a*, *real*) *mapping*
assumes *finite* (*Mapping.keys m*) *All-mapping m* (λ- *x*. *x* ≥ *0*)
shows *nn-integral* (*count-space UNIV*) (λ*k*. *ennreal* (*Mapping.lookup-default 0 m k*)) =
ennreal (∑ *k*∈*Mapping.keys m*. *Mapping.lookup-default 0 m k*)
proof −
have *nn-integral* (*count-space UNIV*) (λ*k*. *ennreal* (*Mapping.lookup-default 0 m k*)) =
(∑ *x*∈*Mapping.keys m*. *ennreal* (*Mapping.lookup-default 0 m x*)) **using** *assms*
by (*subst nn-integral-count-space'*[*of Mapping.keys m*])
(*auto simp*: *Mapping.lookup-default-def keys-is-none-rep Option.is-none-def*)
also from *assms* **have** ... = *ennreal* (∑ *k*∈*Mapping.keys m*. *Mapping.lookup-default 0 m k*)
by (*intro sum-ennreal*)
(*auto simp*: *Mapping.lookup-default-def All-mapping-def split*: *option.splits*)
finally show *?thesis* **.**
qed

lemma *pmf-of-mapping*:
assumes *finite* (*Mapping.keys m*) *All-mapping m* (λ- *p*. *p* ≥ *0*)
assumes (∑ *x*∈*Mapping.keys m*. *Mapping.lookup-default 0 m x*) = *1*
shows *pmf* (*pmf-of-mapping m*) *x* = *Mapping.lookup-default 0 m x*
unfolding *pmf-of-mapping-def*
proof (*intro pmf-embed-pmf*)
from *assms* **show** (∫ +*x*. *ennreal* (*Mapping.lookup-default 0 m x*) ∂*count-space UNIV*) = *1*
by (*subst nn-integral-lookup-default*) (*simp-all*)
qed (*insert assms*, *simp add*: *All-mapping-def Mapping.lookup-default-def split*: *option.splits*)

lemma *pmf-of-set-pmf-of-mapping*:
assumes *A* ≠ {} *set xs* = *A distinct xs*
shows *pmf-of-set A* = *pmf-of-mapping* (*Mapping.tabulate xs* (λ- *1* / *real* (*length xs*)))
(**is** *?lhs* = *?rhs*)
by (*rule pmf-eqI*, *subst pmf-of-mapping*)
(*insert assms*, *auto intro*!: *All-mapping-tabulate*
simp: *Mapping.lookup-default-def lookup-tabulate distinct-card*)

lift-definition *mapping-of-pmf* :: '*a pmf* ⇒ ('*a*, *real*) *mapping* **is**
λ*p x*. *if pmf p x* = *0 then None else Some* (*pmf p x*) **.**

lemma *lookup-default-mapping-of-pmf*:

Mapping.lookup-default 0 (mapping-of-pmf p) x = pmf p x
by (*simp add*: *mapping-of-pmf.abs-eq lookup-default-def Mapping.lookup.abs-eq*)

context
begin

interpretation *pmf-as-function* **.**

lemma *nn-integral-pmf-eq-1*: $(\int^+ x.\ ennreal\ (pmf\ p\ x)\ \partial count\text{-}space\ UNIV) = 1$
by *transfer simp-all*
end

lemma *pmf-of-mapping-mapping-of-pmf* [*code abstype*]:
pmf-of-mapping (mapping-of-pmf p) = p
unfolding *pmf-of-mapping-def*
by (*rule pmf-eqI*, *subst pmf-embed-pmf*)
(*insert nn-integral-pmf-eq-1*[*of p*],
auto simp: *lookup-default-mapping-of-pmf split*: *option.splits*)

lemma *mapping-of-pmfI*:
assumes $\bigwedge x.\ x \in Mapping.keys\ m \Longrightarrow Mapping.lookup\ m\ x = Some\ (pmf\ p\ x)$
assumes *Mapping.keys m = set-pmf p*
shows *mapping-of-pmf p = m*
using *assms* **by** *transfer* (*rule ext*, *auto simp*: *set-pmf-eq*)

lemma *mapping-of-pmfI'*:
assumes $\bigwedge x.\ x \in Mapping.keys\ m \Longrightarrow Mapping.lookup\text{-}default\ 0\ m\ x = pmf\ p\ x$

assumes *Mapping.keys m = set-pmf p*
shows *mapping-of-pmf p = m*
using *assms* **unfolding** *Mapping.lookup-default-def*
by *transfer* (*rule ext*, *force simp*: *set-pmf-eq*)

lemma *return-pmf-code* [*code abstract*]:
mapping-of-pmf (return-pmf x) = Mapping.update x 1 Mapping.empty
by (*intro mapping-of-pmfI*) (*auto simp*: *lookup-update'*)

lemma *pmf-of-set-code-aux*:
assumes *A ≠ {} set xs = A distinct xs*
shows *mapping-of-pmf (pmf-of-set A) = Mapping.tabulate xs (λ-. 1 / real (length xs))*
using *assms*
by (*intro mapping-of-pmfI*, *subst pmf-of-set*)
(*auto simp*: *lookup-tabulate distinct-card*)

definition *pmf-of-set-impl* **where**
pmf-of-set-impl A = mapping-of-pmf (pmf-of-set A)

lemma *pmf-of-set-impl-code-alt*:
assumes $A \neq \{\}$ *finite A*
shows *pmf-of-set-impl A* =
(*let p* = *1 / real* (*card A*)
in Finite-Set.fold (λ*x. Mapping.update x p*) *Mapping.empty A*)
proof −
define *p* **where** *p* = *1 / real* (*card A*)
let *?m* = *Finite-Set.fold* (λ*x. Mapping.update x p*) *Mapping.empty A*
interpret *comp-fun-idem* λ*x. Mapping.update x p*
by *standard* (*transfer*, *force simp*: *fun-eq-iff*)+
have *keys*: *Mapping.keys ?m* = *A*
using *assms*(*2*) **by** (*induction A rule*: *finite-induct*) *simp-all*
have *lookup*: *Mapping.lookup ?m x* = *Some p* **if** *x* ∈ *A* **for** *x*
using *assms*(*2*) *that* **by** (*induction A rule*: *finite-induct*) (*auto simp*: *lookup-update′*)
from *keys lookup assms* **show** *?thesis* **unfolding** *pmf-of-set-impl-def*
by (*intro mapping-of-pmfI*) (*simp-all add*: *Let-def p-def*)
qed

lemma *pmf-of-set-impl-code* [*code*]:
pmf-of-set-impl (*set xs*) =
(*if xs* = [] *then*
Code.abort (*STR ''pmf-of-set of empty set''*) (λ*-. mapping-of-pmf* (*pmf-of-set* (*set xs*)))
else let xs′ = *remdups xs*; *p* = *1 / real* (*length xs′*) *in*
Mapping.tabulate xs′ (λ*-. p*))
unfolding *pmf-of-set-impl-def*
using *pmf-of-set-code-aux*[*of set xs remdups xs*] **by** (*simp add*: *Let-def*)

lemma *pmf-of-set-code* [*code abstract*]:
mapping-of-pmf (*pmf-of-set A*) = *pmf-of-set-impl A*
by (*simp add*: *pmf-of-set-impl-def*)

lemma *pmf-of-multiset-pmf-of-mapping*:
assumes $A \neq \{\#\}$ *set xs* = *set-mset A distinct xs*
shows *mapping-of-pmf* (*pmf-of-multiset A*) = *Mapping.tabulate xs* (λ*x. count A x / real* (*size A*))
using *assms* **by** (*intro mapping-of-pmfI*) (*auto simp*: *lookup-tabulate*)

definition *pmf-of-multiset-impl* **where**
pmf-of-multiset-impl A = *mapping-of-pmf* (*pmf-of-multiset A*)

lemma *pmf-of-multiset-impl-code-alt*:
assumes $A \neq \{\#\}$
shows *pmf-of-multiset-impl A* =
(*let p* = *1 / real* (*size A*)
in fold-mset (λ*x. Mapping.map-default x 0* ((+) *p*)) *Mapping.empty A*)
proof −
define *p* **where** *p* = *1 / real* (*size A*)

interpret *comp-fun-commute* λx. *Mapping.map-default x 0* ((+) *p*)
unfolding *Mapping.map-default-def* [*abs-def*]
by (*standard*, *intro mapping-eqI ext*)
(*simp-all add*: *o-def lookup-map-entry' lookup-default' lookup-default-def*)
let *?m* = *fold-mset* (λx. *Mapping.map-default x 0* ((+) *p*)) *Mapping.empty A*
have *keys*: *Mapping.keys ?m* = *set-mset A* **by** (*induction A*) *simp-all*
have *lookup*: *Mapping.lookup-default 0 ?m x* = *real* (*count A x*) ∗ *p* **for** *x*
by (*induction A*)
(*simp-all add*: *lookup-map-default' lookup-default-def lookup-empty ring-distribs*)
from *keys lookup assms* **show** *?thesis* **unfolding** *pmf-of-multiset-impl-def*
by (*intro mapping-of-pmfI'*) (*simp-all add*: *Let-def p-def*)
qed

lemma *pmf-of-multiset-impl-code* [*code*]:
pmf-of-multiset-impl (*mset xs*) =
(*if xs* = [] *then*
Code.abort (*STR ''pmf-of-multiset of empty multiset''*)
(λ-. *mapping-of-pmf* (*pmf-of-multiset* (*mset xs*)))
else let xs' = *remdups xs*; *p* = *1* / *real* (*length xs*) *in*
Mapping.tabulate xs' (λx. *real* (*count* (*mset xs*) *x*) ∗ *p*))
using *pmf-of-multiset-pmf-of-mapping*[*of mset xs remdups xs*]
by (*simp add*: *pmf-of-multiset-impl-def*)

lemma *pmf-of-multiset-code* [*code abstract*]:
mapping-of-pmf (*pmf-of-multiset A*) = *pmf-of-multiset-impl A*
by (*simp add*: *pmf-of-multiset-impl-def*)

lemma *bernoulli-pmf-code* [*code abstract*]:
mapping-of-pmf (*bernoulli-pmf p*) =
(*if p* ≤ *0 then Mapping.update False 1 Mapping.empty*
else if p ≥ *1 then Mapping.update True 1 Mapping.empty*
else Mapping.update False (*1* − *p*) (*Mapping.update True p Mapping.empty*))
by (*intro mapping-of-pmfI*) (*auto simp*: *bernoulli-pmf.rep-eq lookup-update' set-pmf-eq*)

lemma *pmf-code* [*code*]: *pmf p x* = *Mapping.lookup-default 0* (*mapping-of-pmf p*) *x*
unfolding *mapping-of-pmf-def Mapping.lookup-default-def*
by (*auto split*: *option.splits simp*: *id-def Mapping.lookup.abs-eq*)

lemma *set-pmf-code* [*code*]: *set-pmf p* = *Mapping.keys* (*mapping-of-pmf p*)
by *transfer* (*auto simp*: *dom-def set-pmf-eq*)

lemma *keys-mapping-of-pmf* [*simp*]: *Mapping.keys* (*mapping-of-pmf p*) = *set-pmf p*
by *transfer* (*auto simp*: *dom-def set-pmf-eq*)

definition *fold-combine-plus* **where**
fold-combine-plus = *comm-monoid-set.F* (*Mapping.combine* ((+) :: *real* ⇒ -)) *Mapping.empty*

context
begin

interpretation *fold-combine-plus*: *combine-mapping-abel-semigroup* (+) :: *real* ⇒ -
by *unfold-locales* (*simp-all add*: *add-ac*)

qualified lemma *lookup-default-fold-combine-plus*:
fixes *A* :: *'b set* **and** *f* :: *'b* ⇒ (*'a*, *real*) *mapping*
assumes *finite A*
shows *Mapping.lookup-default 0* (*fold-combine-plus f A*) *x* =
(∑ *y*∈*A*. *Mapping.lookup-default 0* (*f y*) *x*)
unfolding *fold-combine-plus-def* **using** *assms*
by (*induction A rule*: *finite-induct*)
(*simp-all add*: *lookup-default-empty lookup-default-neutral-combine*)

qualified lemma *keys-fold-combine-plus*:
finite A ⟹ *Mapping.keys* (*fold-combine-plus f A*) = (⋃ *x*∈*A*. *Mapping.keys* (*f x*))
by (*simp add*: *fold-combine-plus-def fold-combine-plus.keys-fold-combine*)

qualified lemma *fold-combine-plus-code* [*code*]:
fold-combine-plus g (*set xs*) = *foldr* (λ*x*. *Mapping.combine* (+) (*g x*)) (*remdups xs*) *Mapping.empty*
by (*simp add*: *fold-combine-plus-def fold-combine-plus.fold-combine-code*)

private lemma *lookup-default-0-map-values*:
assumes *f x 0* = *0*
shows *Mapping.lookup-default 0* (*Mapping.map-values f m*) *x* = *f x* (*Mapping.lookup-default 0 m x*)
unfolding *Mapping.lookup-default-def*
using *assms* **by** *transfer* (*auto split*: *option.splits*)

qualified lemma *mapping-of-bind-pmf*:
assumes *finite* (*set-pmf p*)
shows *mapping-of-pmf* (*bind-pmf p f*) =
fold-combine-plus (λ*x*. *Mapping.map-values* (λ-. (∗)) (*pmf p x*))
(*mapping-of-pmf* (*f x*))) (*set-pmf p*)
using *assms*
by (*intro mapping-of-pmfI'*)
(*auto simp*: *keys-fold-combine-plus lookup-default-fold-combine-plus*
pmf-bind integral-measure-pmf lookup-default-0-map-values
lookup-default-mapping-of-pmf mult-ac)

lift-definition *bind-pmf-aux* :: *'a pmf* ⇒ (*'a* ⇒ *'b pmf*) ⇒ *'a set* ⇒ (*'b*, *real*) *mapping* **is**
  λ(*p* :: *'a pmf*) (*f* :: *'a* ⇒ *'b pmf*) (*A*::*'a set*) (*x*::*'b*).
    if x ∈ (⋃ *y*∈*A*. *set-pmf* (*f y*)) *then*
      Some (*measure-pmf.expectation p* (λ*y*. *indicator A y* ∗ *pmf* (*f y*) *x*))
    else None .

lemma *keys-bind-pmf-aux* [*simp*]:
  Mapping.keys (*bind-pmf-aux p f A*) = (⋃ *x*∈*A*. *set-pmf* (*f x*))
  by *transfer* (*auto split*: *if-splits*)

lemma *lookup-default-bind-pmf-aux*:
  Mapping.lookup-default 0 (*bind-pmf-aux p f A*) *x* =
    (*if x* ∈ (⋃ *y*∈*A*. *set-pmf* (*f y*)) *then*
      measure-pmf.expectation p (λ*y*. *indicator A y* ∗ *pmf* (*f y*) *x*) *else 0*)
  unfolding *lookup-default-def* **by** *transfer'* *simp-all*

lemma *lookup-default-bind-pmf-aux'* [*simp*]:
  Mapping.lookup-default 0 (*bind-pmf-aux p f* (*set-pmf p*)) *x* = *pmf* (*bind-pmf p f*) *x*
  unfolding *lookup-default-def*
  by *transfer* (*auto simp*: *pmf-bind AE-measure-pmf-iff set-pmf-eq*
      intro!: *integral-cong-AE integral-eq-zero-AE*)

lemma *bind-pmf-aux-correct*:
  mapping-of-pmf (*bind-pmf p f*) = *bind-pmf-aux p f* (*set-pmf p*)
  by (*intro mapping-of-pmfI'*) *simp-all*

lemma *bind-pmf-aux-code-aux*:
  assumes *finite A*
  shows *bind-pmf-aux p f A* =
      fold-combine-plus (λ*x*. *Mapping.map-values* (λ-. (∗) (*pmf p x*))
        (*mapping-of-pmf* (*f x*))) *A* (**is** *?lhs* = *?rhs*)
proof (*intro mapping-eqI'*[**where** *d* = *0*])
  fix *x* **assume** *x* ∈ *Mapping.keys ?lhs*
  then obtain *y* **where** *y*: *y* ∈ *A x* ∈ *set-pmf* (*f y*) **by** *auto*
  hence *Mapping.lookup-default 0 ?lhs x* =
      measure-pmf.expectation p (λ*y*. *indicator A y* ∗ *pmf* (*f y*) *x*)
    by (*auto simp*: *lookup-default-bind-pmf-aux*)
  also from *assms* **have** ... = (∑ *y*∈*A*. *pmf p y* ∗ *pmf* (*f y*) *x*)
    by (*subst integral-measure-pmf* [*of A*])
      (*auto simp*: *set-pmf-eq indicator-def mult-ac split*: *if-splits*)
  also from *assms* **have** ... = *Mapping.lookup-default 0 ?rhs x*
    by (*simp add*: *lookup-default-fold-combine-plus lookup-default-0-map-values*
      lookup-default-mapping-of-pmf)
  finally show *Mapping.lookup-default 0 ?lhs x* = *Mapping.lookup-default 0 ?rhs x* .
qed (*insert assms*, *simp-all add*: *keys-fold-combine-plus*)

lemma *bind-pmf-aux-code* [*code*]:
bind-pmf-aux p f (*set xs*) =
fold-combine-plus (λ*x*. *Mapping.map-values* (λ-. (∗) (*pmf p x*))
(*mapping-of-pmf* (*f x*))) (*set xs*)
by (*rule bind-pmf-aux-code-aux*) *simp-all*

lemmas *bind-pmf-code* [*code abstract*] = *bind-pmf-aux-correct*

end

hide-const (**open**) *fold-combine-plus*

lift-definition *cond-pmf-impl* :: *'a pmf* ⇒ *'a set* ⇒ (*'a*, *real*) *mapping option* **is**
λ*p A*. *if A* ∩ *set-pmf p* = {} *then None else*
Some (λ*x*. *if x* ∈ *A* ∩ *set-pmf p then Some* (*pmf p x* / *measure-pmf.prob p A*)
else None) **.**

lemma *cond-pmf-impl-code-alt*:
assumes *finite A*
shows *cond-pmf-impl p A* = (
let C = *A* ∩ *set-pmf p*;
prob = (∑ *x*∈*C*. *pmf p x*)
in if prob = *0 then*
None
else
Some (*Mapping.map-values* (λ- *y*. *y* / *prob*)
(*Mapping.filter* (λ*k* -. *k* ∈ *C*) (*mapping-of-pmf p*))))
proof −
define *C* **where** *C* = *A* ∩ *set-pmf p*
define *prob* **where** *prob* = (∑ *x*∈*C*. *pmf p x*)
also note *C-def*
also from *assms* **have** (∑ *x*∈*A* ∩ *set-pmf p*. *pmf p x*) = (∑ *x*∈*A*. *pmf p x*)
by (*intro sum.mono-neutral-left*) (*auto simp*: *set-pmf-eq*)
finally have *prob1*: *prob* = (∑ *x*∈*A*. *pmf p x*) **.**
hence *prob2*: *prob* = *measure-pmf.prob p A*
using *assms* **by** (*subst measure-measure-pmf-finite*) *simp-all*
have *prob3*: *prob* = *0* ⟷ *A* ∩ *set-pmf p* = {}
by (*subst prob1*, *subst sum-nonneg-eq-0-iff*) (*auto simp*: *set-pmf-eq assms*)
from *assms* **have** *prob4*: *prob* = *measure-pmf.prob p C*
unfolding *prob-def* **by** (*intro measure-measure-pmf-finite* [*symmetric*]) (*simp-all add*: *C-def*)

show *?thesis*
proof (*cases prob* = *0*)
case *True*
hence *A* ∩ *set-pmf p* = {} **by** (*subst* (*asm*) *prob3*)
with *True* **show** *?thesis* **by** (*simp add*: *Let-def prob-def C-def cond-pmf-impl.abs-eq*)
next

```
    case False
    hence A: C ≠ {} unfolding C-def by (subst (asm) prob3) auto
    with prob3 have prob-nz: prob ≠ 0 by (auto simp: C-def)
    fix x
    have cond-pmf-impl p A =
           Some (mapping.Mapping (λx. if x ∈ C then
             Some (pmf p x / measure-pmf.prob p C) else None))
         (is - = Some ?m)
      using A prob2 prob4 unfolding C-def by transfer (auto simp: fun-eq-iff)
    also have ?m = Mapping.map-values (λ- y. y / prob)
                (Mapping.filter (λk -. k ∈ C) (mapping-of-pmf p))
      using prob-nz prob4 assms unfolding C-def
      by transfer (auto simp: fun-eq-iff set-pmf-eq)
    finally show ?thesis using False by (simp add: Let-def prob-def C-def)
  qed
qed

lemma cond-pmf-impl-code [code]:
  cond-pmf-impl p (set xs) = (
    let C = set xs ∩ set-pmf p;
        prob = (∑ x∈C. pmf p x)
    in  if prob = 0 then
          None
        else
          Some (Mapping.map-values (λ- y. y / prob)
            (Mapping.filter (λk -. k ∈ C) (mapping-of-pmf p))))
  by (rule cond-pmf-impl-code-alt) simp-all

lemma cond-pmf-code [code abstract]:
  mapping-of-pmf (cond-pmf p A) =
    (case cond-pmf-impl p A of
       None ⇒ Code.abort (STR ''cond-pmf with set of probability 0'')
                (λ-. mapping-of-pmf (cond-pmf p A))
     | Some m ⇒ m)
proof (cases cond-pmf-impl p A)
  case (Some m)
  hence A: set-pmf p ∩ A ≠ {} by transfer (auto split: if-splits)
  from Some have B: Mapping.keys m = set-pmf (cond-pmf p A)
    by (subst set-cond-pmf[OF A], transfer) (auto split: if-splits)
  with Some A have mapping-of-pmf (cond-pmf p A) = m
   by (intro mapping-of-pmfI[OF - B], transfer) (auto split: if-splits simp: pmf-cond)
  with Some show ?thesis by simp
qed simp-all


lemma binomial-pmf-code [code abstract]:
  mapping-of-pmf (binomial-pmf n p) = (
    if p < 0 ∨ p > 1 then
      Code.abort (STR ''binomial-pmf with invalid probability'')
```

```
      (λ-. mapping-of-pmf (binomial-pmf n p))
    else if p = 0 then Mapping.update 0 1 Mapping.empty
    else if p = 1 then Mapping.update n 1 Mapping.empty
    else Mapping.tabulate [0..<Suc n] (λk. real (n choose k) * p ^ k * (1 - p) ^
(n - k)))
  by (cases p < 0 ∨ p > 1)
    (simp, intro mapping-of-pmfI,
      auto simp: lookup-update' lookup-empty set-pmf-binomial-eq lookup-tabulate
split: if-splits)


lemma pred-pmf-code [code]:
  pred-pmf P p = (∀ x∈set-pmf p. P x)
  by (auto simp: pred-pmf-def)


lemma mapping-of-pmf-pmf-of-list:
  assumes ⋀x. x ∈ snd ` set xs ⟹ x > 0 sum-list (map snd xs) = 1
  shows   mapping-of-pmf (pmf-of-list xs) =
          Mapping.tabulate (remdups (map fst xs))
            (λx. sum-list (map snd (filter (λz. fst z = x) xs)))
proof −
  from assms have wf: pmf-of-list-wf xs by (intro pmf-of-list-wfI) force
  with assms have set-pmf (pmf-of-list xs) = fst ` set xs
    by (intro set-pmf-of-list-eq) auto
  with wf show ?thesis
    by (intro mapping-of-pmfI) (auto simp: lookup-tabulate pmf-pmf-of-list)
qed

lemma mapping-of-pmf-pmf-of-list':
  assumes pmf-of-list-wf xs
  defines xs' ≡ filter (λz. snd z ≠ 0) xs
  shows   mapping-of-pmf (pmf-of-list xs) =
          Mapping.tabulate (remdups (map fst xs'))
            (λx. sum-list (map snd (filter (λz. fst z = x) xs'))) (is - = ?rhs)
proof −
  have wf: pmf-of-list-wf xs' unfolding xs'-def by (rule pmf-of-list-remove-zeros)
fact
  have pos: ∀ x∈snd`set xs'. x > 0 using assms(1) unfolding xs'-def
    by (force simp: pmf-of-list-wf-def)
  from assms have pmf-of-list xs = pmf-of-list xs'
    unfolding xs'-def by (subst pmf-of-list-remove-zeros) simp-all
  also from wf pos have mapping-of-pmf ... = ?rhs
    by (intro mapping-of-pmf-pmf-of-list) (auto simp: pmf-of-list-wf-def)
  finally show ?thesis .
qed

lemma pmf-of-list-wf-code [code]:
  pmf-of-list-wf xs ⟷ list-all (λz. snd z ≥ 0) xs ∧ sum-list (map snd xs) = 1
```

by (*auto simp add*: *pmf-of-list-wf-def list-all-def*)

lemma *pmf-of-list-code* [*code abstract*]:
mapping-of-pmf (*pmf-of-list xs*) = (
if pmf-of-list-wf xs then
let xs′ = *filter* (λ*z*. *snd z* ≠ *0*) *xs*
in Mapping.tabulate (*remdups* (*map fst xs′*))
(λ*x*. *sum-list* (*map snd* (*filter* (λ*z*. *fst z* = *x*) *xs′*)))
else
Code.abort (*STR ''Invalid list for pmf-of-list''*) (λ-. *mapping-of-pmf* (*pmf-of-list xs*)))
using *mapping-of-pmf-pmf-of-list′*[*of xs*] **by** (*simp add*: *Let-def*)

lemma *mapping-of-pmf-eq-iff* [*simp*]:
mapping-of-pmf p = *mapping-of-pmf q* ⟷ *p* = (*q* :: *′a pmf*)
proof (*transfer*, *intro iffI pmf-eqI*)
fix *p q* :: *′a pmf* **and** *x* :: *′a*
assume (λ*x*. *if pmf p x* = *0 then None else Some* (*pmf p x*)) =
(λ*x*. *if pmf q x* = *0 then None else Some* (*pmf q x*))
hence (*if pmf p x* = *0 then None else Some* (*pmf p x*)) =
(*if pmf q x* = *0 then None else Some* (*pmf q x*)) **for** *x*
by (*simp add*: *fun-eq-iff*)
from *this*[*of x*] **show** *pmf p x* = *pmf q x* **by** (*auto split*: *if-splits*)
qed (*simp-all cong*: *if-cong*)

21.2 Code abbreviations for integrals and probabilities

Integrals and probabilities are defined for general measures, so we cannot give any code equations directly. We can, however, specialise these constants them to PMFs, give code equations for these specialised constants, and tell the code generator to unfold the original constants to the specialised ones whenever possible.

definition *pmf-integral* **where**
pmf-integral p f = *lebesgue-integral* (*measure-pmf p*) (*f* :: - ⇒ *real*)

definition *pmf-set-integral* **where**
pmf-set-integral p f A = *lebesgue-integral* (*measure-pmf p*) (λ*x*. *indicator A x* ∗ *f x* :: *real*)

definition *pmf-prob* **where**
pmf-prob p A = *measure-pmf.prob p A*

lemma *pmf-prob-compl*: *pmf-prob p* (−*A*) = *1* − *pmf-prob p A*
using *measure-pmf.prob-compl*[*of A p*] **by** (*simp add*: *pmf-prob-def Compl-eq-Diff-UNIV*)

lemma *pmf-integral-pmf-set-integral* [*code*]:
pmf-integral p f = *pmf-set-integral p f* (*set-pmf p*)
unfolding *pmf-integral-def pmf-set-integral-def*

by (*intro integral-cong-AE*) (*simp-all add*: *AE-measure-pmf-iff*)

lemma *pmf-prob-pmf-set-integral*:
pmf-prob p A = *pmf-set-integral p* (λ-. *1*) *A*
by (*simp add*: *pmf-prob-def pmf-set-integral-def*)

lemma *pmf-set-integral-code-alt-finite*:
finite A ⟹ *pmf-set-integral p f A* = (∑ *x*∈*A*. *pmf p x* ∗ *f x*)
unfolding *pmf-set-integral-def*
by (*subst integral-measure-pmf*[*of A*]) (*auto simp*: *indicator-def mult-ac split*: *if-splits*)

lemma *pmf-set-integral-code* [*code*]:
pmf-set-integral p f (*set xs*) = (∑ *x*∈*set xs*. *pmf p x* ∗ *f x*)
by (*rule pmf-set-integral-code-alt-finite*) *simp-all*

lemma *pmf-prob-code-alt-finite*:
finite A ⟹ *pmf-prob p A* = (∑ *x*∈*A*. *pmf p x*)
by (*simp add*: *pmf-prob-pmf-set-integral pmf-set-integral-code-alt-finite*)

lemma *pmf-prob-code* [*code*]:
pmf-prob p (*set xs*) = (∑ *x*∈*set xs*. *pmf p x*)
pmf-prob p (*List.coset xs*) = *1* − (∑ *x*∈*set xs*. *pmf p x*)
by (*simp-all add*: *pmf-prob-code-alt-finite pmf-prob-compl*)

lemma *pmf-prob-code-unfold* [*code-abbrev*]: *pmf-prob p* = *measure-pmf.prob p*
by (*intro ext*) (*simp add*: *pmf-prob-def*)

lemma *pmf-integral-code-unfold* [*code-abbrev*]: *pmf-integral p* = *measure-pmf.expectation p*
by (*intro ext*) (*simp add*: *pmf-integral-def*)

definition *pmf-of-alist xs* = *embed-pmf* (λ*x*. *case map-of xs x of Some p* ⇒ *p* | *None* ⇒ *0*)

lemma *pmf-of-mapping-Mapping* [*code-post*]:
pmf-of-mapping (*Mapping xs*) = *pmf-of-alist xs*
unfolding *pmf-of-mapping-def Mapping.lookup-default-def* [*abs-def*] *pmf-of-alist-def*
by *transfer simp-all*

instantiation *pmf* :: (*equal*) *equal*
begin

definition *equal-pmf p q* = (*mapping-of-pmf p* = *mapping-of-pmf* (*q* :: *'a pmf*))

instance by *standard* (*simp add*: *equal-pmf-def*)
end

definition *single* :: *'a* ⇒ *'a multiset* **where**
single s = {#*s*#}

instantiation *pmf* :: (*random*) *random*
begin

context
includes *state-combinator-syntax term-syntax*
begin

definition
pmfify :: (*'b*::*typerep multiset* × (*unit* ⇒ *Code-Evaluation.term*)) ⇒
'b × (*unit* ⇒ *Code-Evaluation.term*) ⇒
'b pmf × (*unit* ⇒ *Code-Evaluation.term*) **where**
[*code-unfold*]: *pmfify A x* =
Code-Evaluation.valtermify pmf-of-multiset {·}
(*Code-Evaluation.valtermify* (+) {·} *A* {·}
(*Code-Evaluation.valtermify single* {·} *x*))

definition
Quickcheck-Random.random i =
Quickcheck-Random.random i ∘→ (λ*A*.
Quickcheck-Random.random i ∘→ (λ*x*. *Pair* (*pmfify A x*)))

instance ..

end

end

instantiation *pmf* :: (*full-exhaustive*) *full-exhaustive*
begin

definition *full-exhaustive-pmf* :: (*'a pmf* × (*unit* ⇒ *term*) ⇒ (*bool* × *term list*) *option*) ⇒ *natural* ⇒ (*bool* × *term list*) *option*
where
full-exhaustive-pmf f i =
Quickcheck-Exhaustive.full-exhaustive (λ*A*.
Quickcheck-Exhaustive.full-exhaustive (λ*x*. *f* (*pmfify A x*)) *i*) *i*

instance ..

end

end

22 Finite Maps

theory *Fin-Map*
imports *HOL−Analysis.Finite-Product-Measure HOL−Library.Finite-Map*
begin

The *fmap* type can be instantiated to *polish-space*, needed for the proof of projective limit. *extensional* functions are used for the representation in order to stay close to the developments of (finite) products Pi_E and their sigma-algebra Pi_M.

type-notation *fmap* ((- $\Rightarrow_F$ /-) [*22*, *21*] *21*)

unbundle *fmap.lifting*

22.1 Domain and Application

lift-definition *domain*::('*i* $\Rightarrow_F$ '*a*) $\Rightarrow$ '*i set* **is** *dom* **.**

lemma *finite-domain*[*simp*, *intro*]: *finite* (*domain P*)
by *transfer simp*

lift-definition *proj* :: ('*i* $\Rightarrow_F$ '*a*) $\Rightarrow$ '*i* $\Rightarrow$ '*a* ('((-)')$_F$ [*0*] *1000*) **is**
$\lambda f\ x.$ *if* $x \in$ *dom f then the* (*f x*) *else undefined* **.**

declare [[*coercion proj*]]

lemma *extensional-proj*[*simp*, *intro*]: $(P)_F \in$ *extensional* (*domain P*)
by *transfer* (*auto simp*: *extensional-def*)

lemma *proj-undefined*[*simp*, *intro*]: $i \notin$ *domain* $P \Longrightarrow P\ i =$ *undefined*
using *extensional-proj*[*of P*] **unfolding** *extensional-def* **by** *auto*

lemma *finmap-eq-iff*: $P = Q \longleftrightarrow$ (*domain* $P =$ *domain* $Q \wedge$ ($\forall\, i{\in}$*domain* $P.\ P\ i = Q\ i$))
apply *transfer*
apply (*safe intro*!: *ext*)
subgoal for *P Q x*
by (*cases* $x \in$ *dom P*; *cases P x*) (*auto dest*!: *bspec*[**where** $x{=}x$])
done

22.2 Constructor of Finite Maps

lift-definition *finmap-of*::'*i set* $\Rightarrow$ ('*i* $\Rightarrow$ '*a*) $\Rightarrow$ ('*i* $\Rightarrow_F$ '*a*) **is**
$\lambda I\ f\ x.$ *if* $x \in I \wedge$ *finite I then Some* (*f x*) *else None*
by (*simp add*: *dom-def*)

lemma *proj-finmap-of*[*simp*]:
assumes *finite inds*
shows (*finmap-of inds f*)$_F$ = *restrict f inds*
using *assms*
by *transfer force*

lemma *domain-finmap-of*[*simp*]:
assumes *finite inds*
shows *domain* (*finmap-of inds f*) = *inds*
using *assms*
by *transfer* (*auto split*: *if-splits*)

lemma *finmap-of-eq-iff*[*simp*]:
assumes *finite i finite j*
shows *finmap-of i m* = *finmap-of j n* ⟷ *i* = *j* ∧ (∀ *k*∈*i*. *m k*= *n k*)
using *assms* **by** (*auto simp*: *finmap-eq-iff*)

lemma *finmap-of-inj-on-extensional-finite*:
assumes *finite K*
assumes *S* ⊆ *extensional K*
shows *inj-on* (*finmap-of K*) *S*
proof (*rule inj-onI*)
fix *x y*::'*a* ⇒ '*b*
assume *finmap-of K x* = *finmap-of K y*
hence (*finmap-of K x*)$_F$ = (*finmap-of K y*)$_F$ **by** *simp*
moreover
assume *x* ∈ *S y* ∈ *S* **hence** *x* ∈ *extensional K y* ∈ *extensional K* **using** *assms*
by *auto*
ultimately
show *x* = *y* **using** *assms* **by** (*simp add*: *extensional-restrict*)
qed

22.3 Product set of Finite Maps

This is *Pi* for Finite Maps, most of this is copied

definition *Pi'* :: '*i set* ⇒ ('*i* ⇒ '*a set*) ⇒ ('*i* ⇒$_F$ '*a*) *set* **where**
Pi' I A = { *P*. *domain P* = *I* ∧ (∀ *i*. *i* ∈ *I* ⟶ (*P*)$_F$ *i* ∈ *A i*) }

syntax
-*Pi'* :: [*pttrn*, '*a set*, '*b set*] => ('*a* => '*b*) *set* ((3Π'' -∈-./ -) 10)
translations
Π' *x*∈*A*. *B* == *CONST Pi' A* (λ*x*. *B*)

22.3.1 Basic Properties of *Pi'*

lemma *Pi'-I*[*intro*!]: *domain f* = *A* ⟹ (⋀*x*. *x* ∈ *A* ⟹ *f x* ∈ *B x*) ⟹ *f* ∈ *Pi' A B*
by (*simp add*: *Pi'-def*)

lemma *Pi′-I′*[*simp*]: *domain f = A ⟹ (⋀x. x ∈ A ⟶ f x ∈ B x) ⟹ f ∈ Pi′ A B*
by (*simp add*:*Pi′-def*)

lemma *Pi′-mem*: *f∈ Pi′ A B ⟹ x ∈ A ⟹ f x ∈ B x*
by (*simp add*: *Pi′-def*)

lemma *Pi′-iff*: *f ∈ Pi′ I X ⟷ domain f = I ∧ (∀ i∈I. f i ∈ X i)*
unfolding *Pi′-def* **by** *auto*

lemma *Pi′E* [*elim*]:
f ∈ Pi′ A B ⟹ (f x ∈ B x ⟹ domain f = A ⟹ Q) ⟹ (x ∉ A ⟹ Q) ⟹ Q
by(*auto simp*: *Pi′-def*)

lemma *in-Pi′-cong*:
domain f = domain g ⟹ (⋀ w. w ∈ A ⟹ f w = g w) ⟹ f ∈ Pi′ A B ⟷ g ∈ Pi′ A B
by (*auto simp*: *Pi′-def*)

lemma *Pi′-eq-empty*[*simp*]:
assumes *finite A* **shows** *(Pi′ A B) = {} ⟷ (∃ x∈A. B x = {})*
using *assms*
apply (*simp add*: *Pi′-def*, *auto*)
apply (*drule-tac x = finmap-of A* (*λu. SOME y. y ∈ B u*) **in** *spec*, *auto*)
apply (*cut-tac P= %y. y ∈ B i* **in** *some-eq-ex*, *auto*)
done

lemma *Pi′-mono*: *(⋀x. x ∈ A ⟹ B x ⊆ C x) ⟹ Pi′ A B ⊆ Pi′ A C*
by (*auto simp*: *Pi′-def*)

lemma *Pi-Pi′*: *finite A ⟹ (Pi$_E$ A B) = proj ‘ Pi′ A B*
apply (*auto simp*: *Pi′-def Pi-def extensional-def*)
apply (*rule-tac x = finmap-of A* (*restrict x A*) **in** *image-eqI*)
apply *auto*
done

22.4 Topological Space of Finite Maps

instantiation *fmap* :: (*type*, *topological-space*) *topological-space*
begin

definition *open-fmap* :: (*'a ⇒$_F$ 'b*) *set ⇒ bool* **where**
[*code del*]: *open-fmap = generate-topology {Pi′ a b|a b. ∀ i∈a. open (b i)}*

lemma *open-Pi′I*: *(⋀i. i ∈ I ⟹ open (A i)) ⟹ open (Pi′ I A)*
by (*auto intro*: *generate-topology.Basis simp*: *open-fmap-def*)

instance using *topological-space-generate-topology*

by *intro-classes* (*auto simp*: *open-fmap-def class.topological-space-def*)

end

lemma *open-restricted-space*:
shows *open* {*m*. *P* (*domain m*)}
proof −
have {*m*. *P* (*domain m*)} = (⋃ *i* ∈ *Collect P*. {*m*. *domain m* = *i*}) **by** *auto*
also have *open* ...
proof (*rule*, *safe*, *cases*)
fix *i*::'*a set*
assume *finite i*
hence {*m*. *domain m* = *i*} = *Pi'* *i* (λ-. *UNIV*) **by** (*auto simp*: *Pi'-def*)
also have *open* ... **by** (*auto intro*: *open-Pi'I simp*: ‹*finite i*›)
finally show *open* {*m*. *domain m* = *i*} **.**
next
fix *i*::'*a set*
assume ¬ *finite i* **hence** {*m*. *domain m* = *i*} = {} **by** *auto*
also have *open* ... **by** *simp*
finally show *open* {*m*. *domain m* = *i*} **.**
qed
finally show *?thesis* **.**
qed

lemma *closed-restricted-space*:
shows *closed* {*m*. *P* (*domain m*)}
using *open-restricted-space*[*of* λ*x*. ¬ *P x*]
unfolding *closed-def* **by** (*rule back-subst*) *auto*

lemma *tendsto-proj*: ((λ*x*. *x*) ⟶ *a*) *F* ⟹ ((λ*x*. (*x*)$_F$ *i*) ⟶ (*a*)$_F$ *i*) *F*
unfolding *tendsto-def*
proof *safe*
fix *S*::'*b set*
let *?S* = *Pi'* (*domain a*) (λ*x*. *if x* = *i then S else UNIV*)
assume *open S* **hence** *open ?S* **by** (*auto intro*!: *open-Pi'I*)
moreover assume ∀ *S*. *open S* ⟶ *a* ∈ *S* ⟶ *eventually* (λ*x*. *x* ∈ *S*) *F* *a i* ∈ *S*
ultimately have *eventually* (λ*x*. *x* ∈ *?S*) *F* **by** *auto*
thus *eventually* (λ*x*. (*x*)$_F$ *i* ∈ *S*) *F*
by *eventually-elim* (*insert* ‹*a i* ∈ *S*›, *force simp*: *Pi'-iff split*: *if-split-asm*)
qed

lemma *continuous-proj*:
shows *continuous-on s* (λ*x*. (*x*)$_F$ *i*)
unfolding *continuous-on-def* **by** (*safe intro*!: *tendsto-proj tendsto-ident-at*)

instance *fmap* :: (*type*, *first-countable-topology*) *first-countable-topology*
proof
fix *x*::'*a*⇒$_F$'*b*
have ∀ *i*. ∃ *A*. *countable A* ∧ (∀ *a*∈*A*. *x i* ∈ *a*) ∧ (∀ *a*∈*A*. *open a*) ∧

$(\forall S.\ open\ S \wedge x\ i \in S \longrightarrow (\exists a{\in}A.\ a \subseteq S)) \wedge (\forall a\ b.\ a \in A \longrightarrow b \in A \longrightarrow a \cap b \in A)$ (**is** $\forall i.\ ?th\ i$)
proof
fix i **from** *first-countable-basis-Int-stableE*[*of* x i]
obtain A **where**
$countable\ A$
$\bigwedge C.\ C \in A \Longrightarrow (x)_F\ i \in C$
$\bigwedge C.\ C \in A \Longrightarrow open\ C$
$\bigwedge S.\ open\ S \Longrightarrow (x)_F\ i \in S \Longrightarrow \exists A{\in}A.\ A \subseteq S$
$\bigwedge C\ D.\ C \in A \Longrightarrow D \in A \Longrightarrow C \cap D \in A$
by *auto*
thus *?th i* **by** (*intro exI*[**where** $x{=}A$]) *simp*
qed
then obtain A
where A: $\forall i.\ countable\ (A\ i) \wedge Ball\ (A\ i)\ ((\in)\ ((x)_F\ i)) \wedge Ball\ (A\ i)\ open \wedge (\forall S.\ open\ S \wedge (x)_F\ i \in S \longrightarrow (\exists a{\in}A\ i.\ a \subseteq S)) \wedge (\forall a\ b.\ a \in A\ i \longrightarrow b \in A\ i \longrightarrow a \cap b \in A\ i)$
by (*auto simp*: *choice-iff*)
hence *open-sub*: $\bigwedge i\ S.\ i{\in}domain\ x \Longrightarrow open\ (S\ i) \Longrightarrow x\ i{\in}(S\ i) \Longrightarrow (\exists a{\in}A\ i.\ a{\subseteq}(S\ i))$ **by** *auto*
have *A-notempty*: $\bigwedge i.\ i \in domain\ x \Longrightarrow A\ i \neq \{\}$ **using** *open-sub*[*of* - λ-. *UNIV*] **by** *auto*
let *?A* = $(\lambda f.\ Pi'\ (domain\ x)\ f)$ ` $(Pi_E\ (domain\ x)\ A)$
show $\exists A{::}nat \Rightarrow ('a \Rightarrow_F 'b)\ set.\ (\forall i.\ x \in (A\ i) \wedge open\ (A\ i)) \wedge (\forall S.\ open\ S \wedge x \in S \longrightarrow (\exists i.\ A\ i \subseteq S))$
proof (*rule first-countableI*[*of ?A*], *safe*)
show *countable ?A* **using** *A* **by** (*simp add*: *countable-PiE*)
next
fix $S{::}('a \Rightarrow_F 'b)\ set$ **assume** *open S x* $\in$ *S*
thus $\exists a{\in}?A.\ a \subseteq S$ **unfolding** *open-fmap-def*
proof (*induct rule*: *generate-topology.induct*)
case *UNIV* **thus** *?case* **by** (*auto simp add*: *ex-in-conv PiE-eq-empty-iff A-notempty*)
next
case (*Int a b*)
then obtain *f g* **where**
$f \in Pi_E\ (domain\ x)\ A\ Pi'\ (domain\ x)\ f \subseteq a\ g \in Pi_E\ (domain\ x)\ A\ Pi'\ (domain\ x)\ g \subseteq b$
by *auto*
thus *?case* **using** *A*
by (*auto simp*: *Pi'-iff PiE-iff extensional-def Int-stable-def intro*!: *bexI*[**where** $x{=}\lambda i.\ f\ i \cap g\ i$])
next
case (*UN B*)
then obtain *b* **where** $x \in b\ b \in B$ **by** *auto*
hence $\exists a{\in}?A.\ a \subseteq b$ **using** *UN* **by** *simp*
thus *?case* **using** ‹$b \in B$› **by** *blast*
next
case (*Basis s*)

then obtain *a b* **where** *xs*: $x \in Pi'\ a\ b\ s = Pi'\ a\ b \bigwedge i.\ i \in a \Longrightarrow open\ (b\ i)$
by *auto*
have $\forall i.\ \exists a.\ (i \in domain\ x \wedge open\ (b\ i) \wedge (x)_F\ i \in b\ i) \longrightarrow (a \in A\ i \wedge a \subseteq b\ i)$
using *open-sub*[*of* - *b*] **by** *auto*
then obtain b'
where $\bigwedge i.\ i \in domain\ x \Longrightarrow open\ (b\ i) \Longrightarrow (x)_F\ i \in b\ i \Longrightarrow (b'\ i \in A\ i \wedge b'\ i \subseteq b\ i)$
unfolding *choice-iff* **by** *auto*
with *xs* **have** $\bigwedge i.\ i \in a \Longrightarrow (b'\ i \in A\ i \wedge b'\ i \subseteq b\ i)\ Pi'\ a\ b' \subseteq Pi'\ a\ b$
by (*auto simp*: *Pi'-iff intro*!: *Pi'-mono*)
thus *?case* **using** *xs*
by (*intro bexI*[**where** x=$Pi'\ a\ b'$])
(*auto simp*: *Pi'-iff intro*!: *image-eqI*[**where** x=*restrict* b' (*domain* x)])
qed
qed (*insert A*,*auto simp*: *PiE-iff intro*!: *open-Pi'I*)
qed

22.5 Metric Space of Finite Maps

instantiation *fmap* :: (*type*, *metric-space*) *dist*
begin

definition *dist-fmap* **where**
dist P Q = *Max* (*range* ($\lambda i.$ *dist* ($(P)_F$ *i*) ($(Q)_F$ *i*))) + (*if domain P* = *domain Q then 0 else 1*)

instance ..
end

instantiation *fmap* :: (*type*, *metric-space*) *uniformity-dist*
begin

definition [*code del*]:
(*uniformity* :: (('*a*, '*b*) *fmap* × ('*a* ⇒$_F$ '*b*)) *filter*) =
(*INF e*∈{*0* <..}. *principal* {(*x*, *y*). *dist x y* < *e*})

instance
by *standard* (*rule uniformity-fmap-def*)
end

declare *uniformity-Abort*[**where** '*a*=('*a* ⇒$_F$ '*b*::*metric-space*), *code*]

instantiation *fmap* :: (*type*, *metric-space*) *metric-space*
begin

lemma *finite-proj-image'*: $x \notin domain\ P \Longrightarrow finite\ ((P)_F\ `\ S)$
by (*rule finite-subset*[*of* - *proj P* ` (*domain P* ∩ *S* ∪ {*x*})]) *auto*

lemma *finite-proj-image*: *finite* $((P)_F$ ‘ $S)$
by (*cases* $\exists x.\ x \notin$ *domain* P) (*auto intro*: *finite-proj-image′ finite-subset*[**where** B=*domain* P])

lemma *finite-proj-diag*: *finite* $((\lambda i.\ d\ ((P)_F\ i)\ ((Q)_F\ i))$ ‘ $S)$
proof −
have $(\lambda i.\ d\ ((P)_F\ i)\ ((Q)_F\ i))$ ‘ $S = (\lambda(i,\ j).\ d\ i\ j)$ ‘ $((\lambda i.\ ((P)_F\ i,\ (Q)_F\ i))$ ‘ $S)$ **by** *auto*
moreover have $((\lambda i.\ ((P)_F\ i,\ (Q)_F\ i))$ ‘ $S) \subseteq (\lambda i.\ (P)_F\ i)$ ‘ $S \times (\lambda i.\ (Q)_F\ i)$ ‘ S **by** *auto*
moreover have *finite* ... **using** *finite-proj-image*[*of P S*] *finite-proj-image*[*of Q S*]
by (*intro finite-cartesian-product*) *simp-all*
ultimately show *?thesis* **by** (*simp add*: *finite-subset*)
qed

lemma *dist-le-1-imp-domain-eq*:
shows *dist P Q* < *1* ⟹ *domain P* = *domain Q*
by (*simp add*: *dist-fmap-def finite-proj-diag split*: *if-split-asm*)

lemma *dist-proj*:
shows *dist* $((x)_F\ i)\ ((y)_F\ i) \leq$ *dist x y*
proof −
have *dist* $(x\ i)\ (y\ i) \leq$ *Max* $(range\ (\lambda i.\ dist\ (x\ i)\ (y\ i)))$
by (*simp add*: *Max-ge-iff finite-proj-diag*)
also have ... ≤ *dist x y* **by** (*simp add*: *dist-fmap-def*)
finally show *?thesis* **.**
qed

lemma *dist-finmap-lessI*:
assumes *domain P* = *domain Q*
assumes *0* < *e*
assumes ⋀*i*. *i* ∈ *domain P* ⟹ *dist* $(P\ i)\ (Q\ i) < e$
shows *dist P Q* < *e*
proof −
have *dist P Q* = *Max* $(range\ (\lambda i.\ dist\ (P\ i)\ (Q\ i)))$
using *assms* **by** (*simp add*: *dist-fmap-def finite-proj-diag*)
also have ... < *e*
proof (*subst Max-less-iff*, *safe*)
fix *i*
show *dist* $((P)_F\ i)\ ((Q)_F\ i) < e$ **using** *assms*
by (*cases i* ∈ *domain P*) *simp-all*
qed (*simp add*: *finite-proj-diag*)
finally show *?thesis* **.**
qed

instance
proof
fix S::$('a \Rightarrow_F 'b)$ *set*

```
  have ∗: open S = (∀ x∈S. ∃ e>0. ∀ y. dist y x < e ⟶ y ∈ S) (is - = ?od)
  proof
    assume open S
    thus ?od
      unfolding open-fmap-def
    proof (induct rule: generate-topology.induct)
      case UNIV thus ?case by (auto intro: zero-less-one)
    next
      case (Int a b)
      show ?case
      proof safe
        fix x assume x: x ∈ a x ∈ b
        with Int x obtain e1 e2 where
          e1>0 ∀ y. dist y x < e1 ⟶ y ∈ a e2>0 ∀ y. dist y x < e2 ⟶ y ∈ b by
force
        thus ∃ e>0. ∀ y. dist y x < e ⟶ y ∈ a ∩ b
          by (auto intro!: exI[where x=min e1 e2])
      qed
    next
      case (UN K)
      show ?case
      proof safe
        fix x X assume x ∈ X and X: X ∈ K
        with UN obtain e where e>0 ⋀y. dist y x < e ⟶ y ∈ X by force
        with X show ∃ e>0. ∀ y. dist y x < e ⟶ y ∈ ⋃K by auto
      qed
    next
      case (Basis s) then obtain a b where s: s = Pi' a b and b: ⋀i. i∈a ⟹
open (b i) by auto
      show ?case
      proof safe
        fix x assume x ∈ s
        hence [simp]: finite a and a-dom: a = domain x using s by (auto simp:
Pi'-iff)
        obtain es where es: ∀ i ∈ a. es i > 0 ∧ (∀ y. dist y (proj x i) < es i ⟶ y
∈ b i)
          using b ‹x ∈ s› by atomize-elim (intro bchoice, auto simp: open-dist s)
        hence in-b: ⋀i y. i ∈ a ⟹ dist y (proj x i) < es i ⟹ y ∈ b i by auto
        show ∃ e>0. ∀ y. dist y x < e ⟶ y ∈ s
        proof (cases, rule, safe)
          assume a ≠ {}
          show 0 < min 1 (Min (es ‘ a)) using es by (auto simp: ‹a ≠ {}›)
          fix y assume d: dist y x < min 1 (Min (es ‘ a))
          show y ∈ s unfolding s
          proof
        show domain y = a using d s ‹a ≠ {}› by (auto simp: dist-le-1-imp-domain-eq
a-dom)
            fix i assume i: i ∈ a
            hence dist ((y)_F i) ((x)_F i) < es i using d
```

by (*auto simp*: *dist-fmap-def* ‹$a \neq \{\}$› *intro*!: *le-less-trans*[*OF dist-proj*])
with *i* **show** $y\ i \in b\ i$ **by** (*rule in-b*)
qed
next
assume $\neg a \neq \{\}$
thus $\exists\, e{>}0.\ \forall\, y.\ dist\ y\ x < e \longrightarrow y \in s$
using *s* ‹$x \in s$› **by** (*auto simp*: *Pi'-def dist-le-1-imp-domain-eq intro*!: *exI*[**where** *x=1*])
qed
qed
qed
next
assume $\forall\, x{\in}S.\ \exists\, e{>}0.\ \forall\, y.\ dist\ y\ x < e \longrightarrow y \in S$
then obtain *e* **where** *e-pos*: $\bigwedge x.\ x \in S \Longrightarrow e\ x > 0$ **and**
e-in: $\bigwedge x\ y\ .\ x \in S \Longrightarrow dist\ y\ x < e\ x \Longrightarrow y \in S$
unfolding *bchoice-iff*
by *auto*
have *S-eq*: $S = \bigcup \{Pi'\ a\ b |\ a\ b.\ \exists\, x{\in}S.\ domain\ x = a \wedge b = (\lambda i.\ ball\ (x\ i)\ (e\ x))\}$
proof *safe*
fix *x* **assume** $x \in S$
thus $x \in \bigcup \{Pi'\ a\ b |\ a\ b.\ \exists\, x{\in}S.\ domain\ x = a \wedge b = (\lambda i.\ ball\ (x\ i)\ (e\ x))\}$
using *e-pos* **by** (*auto intro*!: *exI*[**where** *x=Pi'* (*domain x*) (λ*i*. *ball* (*x i*) (*e x*))])
next
fix *x y*
assume $y \in S$
moreover
assume $x \in (\Pi'\ i{\in}domain\ y.\ ball\ (y\ i)\ (e\ y))$
hence $dist\ x\ y < e\ y$ **using** *e-pos* ‹$y \in S$›
by (*auto simp*: *dist-fmap-def Pi'-iff finite-proj-diag dist-commute*)
ultimately show $x \in S$ **by** (*rule e-in*)
qed
also have *open* ...
unfolding *open-fmap-def*
by (*intro generate-topology.UN*) (*auto intro*: *generate-topology.Basis*)
finally show *open S* **.**
qed
show $open\ S = (\forall\, x{\in}S.\ \forall_F\ (x', y)\ in\ uniformity.\ x' = x \longrightarrow y \in S)$
unfolding ∗ *eventually-uniformity-metric*
by (*simp del*: *split-paired-All add*: *dist-fmap-def dist-commute eq-commute*)
next
fix $P\ Q{::}'a \Rightarrow_F 'b$
have *Max-eq-iff*: $\bigwedge A\ m.\ finite\ A \Longrightarrow A \neq \{\} \Longrightarrow (Max\ A = m) = (m \in A \wedge (\forall\, a{\in}A.\ a \leq m))$
by (*auto intro*: *Max-in Max-eqI*)
show $dist\ P\ Q = 0 \longleftrightarrow P = Q$
by (*auto simp*: *finmap-eq-iff dist-fmap-def Max-ge-iff finite-proj-diag Max-eq-iff add-nonneg-eq-0-iff*

intro!: *Max-eqI image-eqI*[**where** *x*=*undefined*])
next
fix *P Q R*::*'a* ⇒$_F$ *'b*
let *?dists* = λ*P Q i*. *dist* ((*P*)$_F$ *i*) ((*Q*)$_F$ *i*)
let *?dpq* = *?dists P Q* **and** *?dpr* = *?dists P R* **and** *?dqr* = *?dists Q R*
let *?dom* = λ*P Q*. (*if domain P* = *domain Q then 0 else 1*::*real*)
have *dist P Q* = *Max* (*range ?dpq*) + *?dom P Q*
by (*simp add*: *dist-fmap-def*)
also obtain *t* **where** *t* ∈ *range ?dpq t* = *Max* (*range ?dpq*) **by** (*simp add*: *finite-proj-diag*)
then obtain *i* **where** *Max* (*range ?dpq*) = *?dpq i* **by** *auto*
also have *?dpq i* ≤ *?dpr i* + *?dqr i* **by** (*rule dist-triangle2*)
also have *?dpr i* ≤ *Max* (*range ?dpr*) **by** (*simp add*: *finite-proj-diag*)
also have *?dqr i* ≤ *Max* (*range ?dqr*) **by** (*simp add*: *finite-proj-diag*)
also have *?dom P Q* ≤ *?dom P R* + *?dom Q R* **by** *simp*
finally show *dist P Q* ≤ *dist P R* + *dist Q R* **by** (*simp add*: *dist-fmap-def ac-simps*)
qed

end

22.6 Complete Space of Finite Maps

lemma *tendsto-finmap*:
fixes *f*::*nat* ⇒ (*'i* ⇒$_F$ (*'a*::*metric-space*))
assumes *ind-f*: ⋀*n*. *domain* (*f n*) = *domain g*
assumes *proj-g*: ⋀*i*. *i* ∈ *domain g* ⟹ (λ*n*. (*f n*) *i*) ⟶ *g i*
shows *f* ⟶ *g*
unfolding *tendsto-iff*
proof *safe*
fix *e*::*real* **assume** *0* < *e*
let *?dists* = λ*x i*. *dist* ((*f x*)$_F$ *i*) ((*g*)$_F$ *i*)
have *eventually* (λ*x*. ∀ *i*∈*domain g*. *?dists x i* < *e*) *sequentially*
using *finite-domain*[*of g*] *proj-g*
proof *induct*
case (*insert i G*)
with ‹*0* < *e*› **have** *eventually* (λ*x*. *?dists x i* < *e*) *sequentially* **by** (*auto simp add*: *tendsto-iff*)
moreover
from *insert* **have** *eventually* (λ*x*. ∀ *i*∈*G*. *dist* ((*f x*)$_F$ *i*) ((*g*)$_F$ *i*) < *e*) *sequentially* **by** *simp*
ultimately show *?case* **by** *eventually-elim auto*
qed *simp*
thus *eventually* (λ*x*. *dist* (*f x*) *g* < *e*) *sequentially*
by *eventually-elim* (*auto simp add*: *dist-fmap-def finite-proj-diag ind-f* ‹*0* < *e*›)
qed

instance *fmap* :: (*type*, *complete-space*) *complete-space*
proof

fix P::$nat \Rightarrow 'a \Rightarrow_F 'b$
assume *Cauchy P*
then obtain *Nd* **where** *Nd*: $\bigwedge n.\ n \geq Nd \Longrightarrow dist\ (P\ n)\ (P\ Nd) < 1$
by (*force simp*: *Cauchy-altdef2*)
define d **where** $d = domain\ (P\ Nd)$
with *Nd* **have** *dim*: $\bigwedge n.\ n \geq Nd \Longrightarrow domain\ (P\ n) = d$ **using** *dist-le-1-imp-domain-eq*
by *auto*
have [*simp*]: *finite d* **unfolding** *d-def* **by** *simp*
define p **where** $p\ i\ n = P\ n\ i$ **for** $i\ n$
define q **where** $q\ i = lim\ (p\ i)$ **for** i
define Q **where** $Q = finmap\text{-}of\ d\ q$
have q: $\bigwedge i.\ i \in d \Longrightarrow q\ i = Q\ i$ **by** (*auto simp add*: *Q-def Abs-fmap-inverse*)
{
fix i **assume** $i \in d$
have *Cauchy* $(p\ i)$ **unfolding** *Cauchy-altdef2 p-def*
proof *safe*
fix e::*real* **assume** $0 < e$
with ‹*Cauchy P*› **obtain** N **where** N: $\bigwedge n.\ n \geq N \Longrightarrow dist\ (P\ n)\ (P\ N) <$
min e 1
by (*force simp*: *Cauchy-altdef2 min-def*)
hence $\bigwedge n.\ n \geq N \Longrightarrow domain\ (P\ n) = domain\ (P\ N)$ **using** *dist-le-1-imp-domain-eq*
by *auto*
with *dim* **have** *dim*: $\bigwedge n.\ n \geq N \Longrightarrow domain\ (P\ n) = d$ **by** (*metis nat-le-linear*)
show $\exists N.\ \forall n \geq N.\ dist\ ((P\ n)\ i)\ ((P\ N)\ i) < e$
proof (*safe intro*!: *exI*[**where** x=N])
fix n **assume** $N \leq n$ **have** $N \leq N$ **by** *simp*
have *dist* $((P\ n)\ i)\ ((P\ N)\ i) \leq dist\ (P\ n)\ (P\ N)$
using *dim*[*OF* ‹$N \leq n$›] *dim*[*OF* ‹$N \leq N$›] ‹$i \in d$›
by (*auto intro*!: *dist-proj*)
also have $\ldots < e$ **using** N[*OF* ‹$N \leq n$›] **by** *simp*
finally show *dist* $((P\ n)\ i)\ ((P\ N)\ i) < e$ **.**
qed
qed
hence *convergent* $(p\ i)$ **by** (*metis Cauchy-convergent-iff*)
hence $p\ i \longrightarrow q\ i$ **unfolding** *q-def convergent-def* **by** (*metis limI*)
} **note** $p = this$
have $P \longrightarrow Q$
proof (*rule metric-LIMSEQ-I*)
fix e::*real* **assume** $0 < e$
have $\exists ni.\ \forall i \in d.\ \forall n \geq ni\ i.\ dist\ (p\ i\ n)\ (q\ i) < e$
proof (*safe intro*!: *bchoice*)
fix i **assume** $i \in d$
from p[*OF* ‹$i \in d$›, *THEN metric-LIMSEQ-D*, *OF* ‹$0 < e$›]
show $\exists no.\ \forall n \geq no.\ dist\ (p\ i\ n)\ (q\ i) < e$ **.**
qed
then obtain *ni* **where** *ni*: $\forall i \in d.\ \forall n \geq ni\ i.\ dist\ (p\ i\ n)\ (q\ i) < e$ **..**
define N **where** $N = max\ Nd\ (Max\ (ni\ `\ d))$
show $\exists N.\ \forall n \geq N.\ dist\ (P\ n)\ Q < e$
proof (*safe intro*!: *exI*[**where** x=N])

```
    fix n assume N ≤ n
    hence dom: domain (P n) = d domain Q = d domain (P n) = domain Q
      using dim by (simp-all add: N-def Q-def dim-def Abs-fmap-inverse)
    show dist (P n) Q < e
    proof (rule dist-finmap-lessI[OF dom(3) ‹0 < e›])
      fix i
      assume i ∈ domain (P n)
      hence ni i ≤ Max (ni ` d) using dom by simp
      also have ... ≤ N by (simp add: N-def)
     finally show dist ((P n)_F i) ((Q)_F i) < e using ni ‹i ∈ domain (P n)› ‹N
≤ n› dom
        by (auto simp: p-def q N-def less-imp-le)
    qed
   qed
  qed
  thus convergent P by (auto simp: convergent-def)
qed
```

22.7 Second Countable Space of Finite Maps

```
instantiation fmap :: (countable, second-countable-topology) second-countable-topology
begin

definition basis-proj::'b set set
  where basis-proj = (SOME B. countable B ∧ topological-basis B)

lemma countable-basis-proj: countable basis-proj and basis-proj: topological-basis
basis-proj
  unfolding basis-proj-def by (intro is-basis countable-basis)+

definition basis-finmap::('a ⇒_F 'b) set set
  where basis-finmap = {Pi' I S|I S. finite I ∧ (∀ i ∈ I. S i ∈ basis-proj)}

lemma in-basis-finmapI:
  assumes finite I assumes ⋀i. i ∈ I ⟹ S i ∈ basis-proj
  shows Pi' I S ∈ basis-finmap
  using assms unfolding basis-finmap-def by auto

lemma basis-finmap-eq:
  assumes basis-proj ≠ {}
  shows basis-finmap = (λf. Pi' (domain f) (λi. from-nat-into basis-proj ((f)_F
i))) `
    (UNIV::('a ⇒_F nat) set) (is - = ?f ` -)
  unfolding basis-finmap-def
proof safe
  fix I::'a set and S::'a ⇒ 'b set
  assume finite I ∀ i∈I. S i ∈ basis-proj
  hence Pi' I S = ?f (finmap-of I (λx. to-nat-on basis-proj (S x)))
    by (force simp: Pi'-def countable-basis-proj)
```

thus $Pi'\ I\ S \in range\ ?f$ **by** *simp*
next
fix x **and** $f::'a \Rightarrow_F nat$
show $\exists I\ S.\ (\Pi'\ i{\in}domain\ f.\ from\text{-}nat\text{-}into\ basis\text{-}proj\ ((f)_F\ i)) = Pi'\ I\ S \wedge$
$finite\ I \wedge (\forall i{\in}I.\ S\ i \in basis\text{-}proj)$
using *assms* **by** (*auto intro*: *from-nat-into*)
qed

lemma *basis-finmap-eq-empty*: $basis\text{-}proj = \{\} \Longrightarrow basis\text{-}finmap = \{Pi'\ \{\}\ undefined\}$
by (*auto simp*: *Pi'-iff basis-finmap-def*)

lemma *countable-basis-finmap*: *countable basis-finmap*
by (*cases* $basis\text{-}proj = \{\}$) (*auto simp*: *basis-finmap-eq basis-finmap-eq-empty*)

lemma *finmap-topological-basis*:
topological-basis basis-finmap
proof (*subst topological-basis-iff*, *safe*)
fix B' **assume** $B' \in basis\text{-}finmap$
thus *open* B'
by (*auto intro*!: *open-Pi'I topological-basis-open*[*OF basis-proj*]
simp: *topological-basis-def basis-finmap-def Let-def*)
next
fix $O'::('a \Rightarrow_F 'b)\ set$ **and** x
assume O': *open* $O'\ x \in O'$
then obtain a **where** a:
$x \in Pi'\ (domain\ x)\ a\ Pi'\ (domain\ x)\ a \subseteq O' \bigwedge i.\ i{\in}domain\ x \Longrightarrow open\ (a\ i)$
unfolding *open-fmap-def*
proof (*atomize-elim*, *induct rule*: *generate-topology.induct*)
case (*Int a b*)
let $?p{=}\lambda a\ f.\ x \in Pi'\ (domain\ x)\ f \wedge Pi'\ (domain\ x)\ f \subseteq a \wedge (\forall i.\ i \in domain\ x \longrightarrow open\ (f\ i))$
from *Int* **obtain** *f g* **where** *?p a f ?p b g* **by** *auto*
thus *?case* **by** (*force intro*!: *exI*[**where** $x{=}\lambda i.\ f\ i \cap g\ i$] *simp*: *Pi'-def*)
next
case (*UN k*)
then obtain *kk a* **where** $x \in kk\ kk \in k\ x \in Pi'\ (domain\ x)\ a\ Pi'\ (domain\ x)\ a \subseteq kk$
$\bigwedge i.\ i{\in}domain\ x \Longrightarrow open\ (a\ i)$
by *force*
thus *?case* **by** *blast*
qed (*auto simp*: *Pi'-def*)
have $\exists B.$
$(\forall i{\in}domain\ x.\ x\ i \in B\ i \wedge B\ i \subseteq a\ i \wedge B\ i \in basis\text{-}proj)$
proof (*rule bchoice*, *safe*)
fix i **assume** $i \in domain\ x$
hence *open* $(a\ i)\ x\ i \in a\ i$ **using** a **by** *auto*
from *topological-basisE*[*OF basis-proj this*] **obtain** b'
where $b' \in basis\text{-}proj\ (x)_F\ i \in b'\ b' \subseteq a\ i$

by *blast*
thus $\exists\, y.\ x\ i \in y \wedge y \subseteq a\ i \wedge y \in$ *basis-proj* **by** *auto*
qed
then obtain B **where** B: $\forall\, i{\in}$*domain* $x.\ (x)_F\ i \in B\ i \wedge B\ i \subseteq a\ i \wedge B\ i \in$ *basis-proj*
by *auto*
define B' **where** $B' = Pi'$ (*domain* x) ($\lambda i.\ (B\ i)::'b$ *set*)
have $B' \subseteq Pi'$ (*domain* x) a **using** B **by** (*auto intro*!: *Pi'-mono simp*: B'*-def*)
also note ‹$\ldots \subseteq O'$›
finally show $\exists\, B'{\in}$*basis-finmap*. $x \in B' \wedge B' \subseteq O'$ **using** B
by (*auto intro*!: *bexI*[**where** $x{=}B'$] *Pi'-mono in-basis-finmapI simp*: B'*-def*)
qed

lemma *range-enum-basis-finmap-imp-open*:
assumes $x \in$ *basis-finmap*
shows *open* x
using *finmap-topological-basis assms* **by** (*auto simp*: *topological-basis-def*)

instance proof qed (*blast intro*: *finmap-topological-basis countable-basis-finmap topological-basis-imp-subbasis*)

end

22.8 Polish Space of Finite Maps

instance *fmap* :: (*countable*, *polish-space*) *polish-space* **proof qed**

22.9 Product Measurable Space of Finite Maps

definition *PiF I M* ≡
sigma ($\bigcup J \in I.$ ($\Pi'\ j{\in}J.$ *space* (*M j*))) {($\Pi'\ j{\in}J.$ *X j*) |*X J*. *J* ∈ *I* ∧ *X* ∈ (Π *j*∈*J*. *sets* (*M j*))}

abbreviation
Pi_F *I M* ≡ *PiF I M*

syntax
-PiF :: *pttrn* ⇒ *'i set* ⇒ *'a measure* ⇒ (*'i* => *'a*) *measure* ((*3*Π_F *-*∈*-./ -*) *10*)
translations
$\Pi_F\ x{\in}I.\ M$ == *CONST PiF I* (%*x*. *M*)

lemma *PiF-gen-subset*: {($\Pi'\ j{\in}J.\ X\ j$) |*X J*. *J* ∈ *I* ∧ *X* ∈ (Π *j*∈*J*. *sets* (*M j*))} ⊆
Pow ($\bigcup J \in I.$ ($\Pi'\ j{\in}J.$ *space* (*M j*)))
by (*auto simp*: *Pi'-def*) (*blast dest*: *sets.sets-into-space*)

lemma *space-PiF*: *space* (*PiF I M*) = ($\bigcup J \in I.$ ($\Pi'\ j{\in}J.$ *space* (*M j*)))
unfolding *PiF-def* **using** *PiF-gen-subset* **by** (*rule space-measure-of*)

lemma *sets-PiF*:

sets (*PiF I M*) = *sigma-sets* ($\bigcup J \in I.$ ($\Pi' j{\in}J.$ *space* (*M j*)))
{($\Pi' j{\in}J.$ *X j*) |*X J*. *J* ∈ *I* ∧ *X* ∈ (Π *j*∈*J*. *sets* (*M j*))}
unfolding *PiF-def* **using** *PiF-gen-subset* **by** (*rule sets-measure-of*)

lemma *sets-PiF-singleton*:
sets (*PiF* {*I*} *M*) = *sigma-sets* ($\Pi' j{\in}I.$ *space* (*M j*))
{($\Pi' j{\in}I.$ *X j*) |*X*. *X* ∈ (Π *j*∈*I*. *sets* (*M j*))}
unfolding *sets-PiF* **by** *simp*

lemma *in-sets-PiFI*:
assumes *X* = (*Pi' J S*) *J* ∈ *I* $\bigwedge i.$ *i*∈*J* ⟹ *S i* ∈ *sets* (*M i*)
shows *X* ∈ *sets* (*PiF I M*)
unfolding *sets-PiF*
using *assms* **by** *blast*

lemma *product-in-sets-PiFI*:
assumes *J* ∈ *I* $\bigwedge i.$ *i*∈*J* ⟹ *S i* ∈ *sets* (*M i*)
shows (*Pi' J S*) ∈ *sets* (*PiF I M*)
unfolding *sets-PiF*
using *assms* **by** *blast*

lemma *singleton-space-subset-in-sets*:
fixes *J*
assumes *J* ∈ *I*
assumes *finite J*
shows *space* (*PiF* {*J*} *M*) ∈ *sets* (*PiF I M*)
using *assms*
by (*intro in-sets-PiFI*[**where** *J*=*J* **and** *S*=λ*i*. *space* (*M i*)])
(*auto simp*: *product-def space-PiF*)

lemma *singleton-subspace-set-in-sets*:
assumes *A*: *A* ∈ *sets* (*PiF* {*J*} *M*)
assumes *finite J*
assumes *J* ∈ *I*
shows *A* ∈ *sets* (*PiF I M*)
using *A*[*unfolded sets-PiF*]
apply (*induct A*)
unfolding *sets-PiF*[*symmetric*] **unfolding** *space-PiF*[*symmetric*]
using *assms*
by (*auto intro*: *in-sets-PiFI intro*!: *singleton-space-subset-in-sets*)

lemma *finite-measurable-singletonI*:
assumes *finite I*
assumes $\bigwedge J.$ *J* ∈ *I* ⟹ *finite J*
assumes *MN*: $\bigwedge J.$ *J* ∈ *I* ⟹ *A* ∈ *measurable* (*PiF* {*J*} *M*) *N*
shows *A* ∈ *measurable* (*PiF I M*) *N*
unfolding *measurable-def*
proof *safe*
fix *y* **assume** *y* ∈ *sets N*

```
  have A -‘ y ∩ space (PiF I M) = (⋃J∈I. A -‘ y ∩ space (PiF {J} M))
    by (auto simp: space-PiF)
  also have ... ∈ sets (PiF I M)
  proof (rule sets.finite-UN)
    show finite I by fact
    fix J assume J ∈ I
    with assms have finite J by simp
    show A -‘ y ∩ space (PiF {J} M) ∈ sets (PiF I M)
      by (rule singleton-subspace-set-in-sets[OF measurable-sets[OF assms(3)]])
fact+
  qed
  finally show A -‘ y ∩ space (PiF I M) ∈ sets (PiF I M) .
next
  fix x assume x ∈ space (PiF I M) thus A x ∈ space N
    using MN[of domain x]
    by (auto simp: space-PiF measurable-space Pi'-def)
qed

lemma countable-finite-comprehension:
  fixes f :: 'a::countable set ⇒ -
  assumes ⋀s. P s ⟹ finite s
  assumes ⋀s. P s ⟹ f s ∈ sets M
  shows ⋃{f s|s. P s} ∈ sets M
proof -
  have ⋃{f s|s. P s} = (⋃n::nat. let s = set (from-nat n) in if P s then f s else
{})
  proof safe
    fix x X s assume ∗: x ∈ f s P s
    with assms obtain l where s = set l using finite-list by blast
   with ∗ show x ∈ (⋃n. let s = set (from-nat n) in if P s then f s else {}) using
‹P s›
      by (auto intro!: exI[where x=to-nat l])
  next
    fix x n assume x ∈ (let s = set (from-nat n) in if P s then f s else {})
    thus x ∈ ⋃{f s|s. P s} using assms by (auto simp: Let-def split: if-split-asm)
  qed
  hence ⋃{f s|s. P s} = (⋃n. let s = set (from-nat n) in if P s then f s else {})
by simp
  also have ... ∈ sets M using assms by (auto simp: Let-def)
  finally show ?thesis .
qed

lemma space-subset-in-sets:
  fixes J::'a::countable set set
  assumes J ⊆ I
  assumes ⋀j. j ∈ J ⟹ finite j
  shows space (PiF J M) ∈ sets (PiF I M)
proof -
  have space (PiF J M) = ⋃{space (PiF {j} M)|j. j ∈ J}
```

```
    unfolding space-PiF by blast
  also have ... ∈ sets (PiF I M) using assms
   by (intro countable-finite-comprehension) (auto simp: singleton-space-subset-in-sets)
  finally show ?thesis .
qed

lemma subspace-set-in-sets:
  fixes J::'a::countable set set
  assumes A: A ∈ sets (PiF J M)
  assumes J ⊆ I
  assumes ⋀j. j ∈ J ⟹ finite j
  shows A ∈ sets (PiF I M)
  using A[unfolded sets-PiF]
  apply (induct A)
  unfolding sets-PiF[symmetric] unfolding space-PiF[symmetric]
  using assms
  by (auto intro: in-sets-PiFI intro!: space-subset-in-sets)

lemma countable-measurable-PiFI:
  fixes I::'a::countable set set
  assumes MN: ⋀J. J ∈ I ⟹ finite J ⟹ A ∈ measurable (PiF {J} M) N
  shows A ∈ measurable (PiF I M) N
  unfolding measurable-def
proof safe
  fix y assume y ∈ sets N
  have A -` y = (⋃{A -` y ∩ {x. domain x = J}|J. finite J}) by auto
  { fix x::'a ⇒F 'b
    from finite-list[of domain x] obtain xs where set xs = domain x by auto
    hence ∃n. domain x = set (from-nat n)
      by (intro exI[where x=to-nat xs]) auto }
  hence A -` y ∩ space (PiF I M) = (⋃n. A -` y ∩ space (PiF ({set (from-nat
n)}∩I) M))
    by (auto simp: space-PiF Pi'-def)
  also have ... ∈ sets (PiF I M)
    apply (intro sets.Int sets.countable-nat-UN subsetI, safe)
    apply (case-tac set (from-nat i) ∈ I)
    apply simp-all
    apply (rule singleton-subspace-set-in-sets[OF measurable-sets[OF MN]])
    using assms ‹y ∈ sets N›
    apply (auto simp: space-PiF)
    done
  finally show A -` y ∩ space (PiF I M) ∈ sets (PiF I M) .
next
  fix x assume x ∈ space (PiF I M) thus A x ∈ space N
    using MN[of domain x] by (auto simp: space-PiF measurable-space Pi'-def)
qed

lemma measurable-PiF:
  assumes f: ⋀x. x ∈ space N ⟹ domain (f x) ∈ I ∧ (∀i∈domain (f x). (f x) i
```

$\in$ *space* (*M i*))
assumes *S*: $\bigwedge J\ S.\ J \in I \Longrightarrow (\bigwedge i.\ i \in J \Longrightarrow S\ i \in sets\ (M\ i)) \Longrightarrow$
$f\ -`\ (Pi'\ J\ S) \cap space\ N \in sets\ N$
shows $f \in measurable\ N\ (PiF\ I\ M)$
unfolding *PiF-def*
using *PiF-gen-subset*
apply (*rule measurable-measure-of*)
using *f* **apply** *force*
apply (*insert S*, *auto*)
done

lemma *restrict-sets-measurable*:
assumes *A*: $A \in sets\ (PiF\ I\ M)$ **and** $J \subseteq I$
shows $A \cap \{m.\ domain\ m \in J\} \in sets\ (PiF\ J\ M)$
using *A*[*unfolded sets-PiF*]
proof (*induct A*)
case (*Basic a*)
then obtain *K S* **where** *S*: $a = Pi'\ K\ S\ K \in I\ (\forall i{\in}K.\ S\ i \in sets\ (M\ i))$
by *auto*
show *?case*
proof *cases*
assume $K \in J$
hence $a \cap \{m.\ domain\ m \in J\} \in \{Pi'\ K\ X\ |X\ K.\ K \in J \wedge X \in (\Pi\ j{\in}K.\ sets\ (M\ j))\}$ **using** *S*
by (*auto intro*!: *exI*[**where** *x*=*K*] *exI*[**where** *x*=*S*] *simp*: *Pi'-def*)
also have ... $\subseteq$ *sets* (*PiF J M*) **unfolding** *sets-PiF* **by** *auto*
finally show *?thesis* **.**
next
assume $K \notin J$
hence $a \cap \{m.\ domain\ m \in J\} = \{\}$ **using** *S* **by** (*auto simp*: *Pi'-def*)
also have ... $\in$ *sets* (*PiF J M*) **by** *simp*
finally show *?thesis* **.**
qed
next
case (*Union a*)
have $\bigcup (a\ `\ UNIV) \cap \{m.\ domain\ m \in J\} = (\bigcup i.\ (a\ i \cap \{m.\ domain\ m \in J\}))$
by *simp*
also have ... $\in$ *sets* (*PiF J M*) **using** *Union* **by** (*intro sets.countable-nat-UN*) *auto*
finally show *?case* **.**
next
case (*Compl a*)
have (*space* (*PiF I M*) $-$ *a*) $\cap \{m.\ domain\ m \in J\} = (space\ (PiF\ J\ M) - (a \cap \{m.\ domain\ m \in J\}))$
using $\langle J \subseteq I \rangle$ **by** (*auto simp*: *space-PiF Pi'-def*)
also have ... $\in$ *sets* (*PiF J M*) **using** *Compl* **by** *auto*
finally show *?case* **by** (*simp add*: *space-PiF*)
qed *simp*

lemma *measurable-finmap-of*:
assumes f: $\bigwedge i.\ (\exists\, x \in \mathit{space}\ N.\ i \in J\ x) \Longrightarrow (\lambda x.\ f\ x\ i) \in \mathit{measurable}\ N\ (M\ i)$
assumes J: $\bigwedge x.\ x \in \mathit{space}\ N \Longrightarrow J\ x \in I\ \bigwedge x.\ x \in \mathit{space}\ N \Longrightarrow \mathit{finite}\ (J\ x)$
assumes JN: $\bigwedge S.\ \{x.\ J\ x = S\} \cap \mathit{space}\ N \in \mathit{sets}\ N$
shows $(\lambda x.\ \mathit{finmap\text{-}of}\ (J\ x)\ (f\ x)) \in \mathit{measurable}\ N\ (\mathit{PiF}\ I\ M)$
proof (*rule measurable-PiF*)
fix x **assume** $x \in \mathit{space}\ N$
with J[*of* x] *measurable-space*[*OF* f]
show $\mathit{domain}\ (\mathit{finmap\text{-}of}\ (J\ x)\ (f\ x)) \in I \wedge$
$(\forall\, i{\in}\mathit{domain}\ (\mathit{finmap\text{-}of}\ (J\ x)\ (f\ x)).\ (\mathit{finmap\text{-}of}\ (J\ x)\ (f\ x))\ i \in \mathit{space}\ (M\ i))$
by *auto*
next
fix $K\ S$ **assume** $K \in I$ **and** $*$: $\bigwedge i.\ i \in K \Longrightarrow S\ i \in \mathit{sets}\ (M\ i)$
with J **have** *eq*: $(\lambda x.\ \mathit{finmap\text{-}of}\ (J\ x)\ (f\ x)) -` \mathit{Pi}'\ K\ S \cap \mathit{space}\ N =$
$(\mathit{if}\ \exists\, x \in \mathit{space}\ N.\ K = J\ x \wedge \mathit{finite}\ K\ \mathit{then}\ \mathit{if}\ K = \{\}\ \mathit{then}\ \{x \in \mathit{space}\ N.\ J\ x = K\}$
$\mathit{else}\ (\bigcap i{\in}K.\ (\lambda x.\ f\ x\ i) -` S\ i \cap \{x \in \mathit{space}\ N.\ J\ x = K\})\ \mathit{else}\ \{\})$
by (*auto simp*: *Pi'-def*)
have r: $\{x \in \mathit{space}\ N.\ J\ x = K\} = \mathit{space}\ N \cap (\{x.\ J\ x = K\} \cap \mathit{space}\ N)$ **by** *auto*
show $(\lambda x.\ \mathit{finmap\text{-}of}\ (J\ x)\ (f\ x)) -` \mathit{Pi}'\ K\ S \cap \mathit{space}\ N \in \mathit{sets}\ N$
unfolding *eq r*
apply (*simp del*: *INT-simps add*:)
apply (*intro conjI impI sets.finite-INT JN sets.Int*[*OF sets.top*])
apply *simp* **apply** *assumption*
apply (*subst Int-assoc*[*symmetric*])
apply (*rule sets.Int*)
apply (*intro measurable-sets*[*OF* f] $*$) **apply** *force* **apply** *assumption*
apply (*intro JN*)
done
qed

lemma *measurable-PiM-finmap-of*:
assumes *finite* J
shows *finmap-of* $J \in \mathit{measurable}\ (\mathit{Pi}_M\ J\ M)\ (\mathit{PiF}\ \{J\}\ M)$
apply (*rule measurable-finmap-of*)
apply (*rule measurable-component-singleton*)
apply *simp*
apply *rule*
apply (*rule* ‹*finite* J›)
apply *simp*
done

lemma *proj-measurable-singleton*:
assumes $A \in \mathit{sets}\ (M\ i)$
shows $(\lambda x.\ (x)_F\ i) -` A \cap \mathit{space}\ (\mathit{PiF}\ \{I\}\ M) \in \mathit{sets}\ (\mathit{PiF}\ \{I\}\ M)$
proof *cases*
assume $i \in I$

hence $(\lambda x.\ (x)_F\ i) -` A \cap space\ (PiF\ \{I\}\ M) =$
Pi' I $(\lambda x.\ if\ x = i\ then\ A\ else\ space\ (M\ x))$
using *sets.sets-into-space*[*OF*] ‹$A \in sets\ (M\ i)$› *assms*
by (*auto simp*: *space-PiF Pi'-def*)
thus *?thesis* **using** *assms* ‹$A \in sets\ (M\ i)$›
by (*intro in-sets-PiFI*) *auto*
next
assume $i \notin I$
hence $(\lambda x.\ (x)_F\ i) -` A \cap space\ (PiF\ \{I\}\ M) =$
(*if undefined* $\in A$ *then space* (*PiF* $\{I\}$ *M*) *else* $\{\}$) **by** (*auto simp*: *space-PiF Pi'-def*)
thus *?thesis* **by** *simp*
qed

lemma *measurable-proj-singleton*:
assumes $i \in I$
shows $(\lambda x.\ (x)_F\ i) \in measurable\ (PiF\ \{I\}\ M)\ (M\ i)$
by (*unfold measurable-def*, *intro CollectI conjI ballI proj-measurable-singleton assms*)
(*insert* ‹$i \in I$›, *auto simp*: *space-PiF*)

lemma *measurable-proj-countable*:
fixes *I*::*'a*::*countable set set*
assumes $y \in space\ (M\ i)$
shows $(\lambda x.\ if\ i \in domain\ x\ then\ (x)_F\ i\ else\ y) \in measurable\ (PiF\ I\ M)\ (M\ i)$
proof (*rule countable-measurable-PiFI*)
fix *J* **assume** $J \in I$ *finite J*
show $(\lambda x.\ if\ i \in domain\ x\ then\ x\ i\ else\ y) \in measurable\ (PiF\ \{J\}\ M)\ (M\ i)$
unfolding *measurable-def*
proof *safe*
fix *z* **assume** $z \in sets\ (M\ i)$
have $(\lambda x.\ if\ i \in domain\ x\ then\ x\ i\ else\ y) -` z \cap space\ (PiF\ \{J\}\ M) =$
$(\lambda x.\ if\ i \in J\ then\ (x)_F\ i\ else\ y) -` z \cap space\ (PiF\ \{J\}\ M)$
by (*auto simp*: *space-PiF Pi'-def*)
also have $\ldots \in sets\ (PiF\ \{J\}\ M)$ **using** ‹$z \in sets\ (M\ i)$› ‹*finite J*›
by (*cases* $i \in J$) (*auto intro*!: *measurable-sets*[*OF measurable-proj-singleton*])
finally show $(\lambda x.\ if\ i \in domain\ x\ then\ x\ i\ else\ y) -` z \cap space\ (PiF\ \{J\}\ M) \in sets\ (PiF\ \{J\}\ M)$ **.**
qed (*insert* ‹$y \in space\ (M\ i)$›, *auto simp*: *space-PiF Pi'-def*)
qed

lemma *measurable-restrict-proj*:
assumes $J \in II$ *finite J*
shows *finmap-of J* $\in measurable\ (PiM\ J\ M)\ (PiF\ II\ M)$
using *assms*
by (*intro measurable-finmap-of measurable-component-singleton*) *auto*

lemma *measurable-proj-PiM*:
fixes *J K* ::*'a*::*countable set* **and** *I*::*'a set set*

assumes *finite J* $J \in I$
assumes $x \in$ *space* (*PiM J M*)
shows *proj* $\in$ *measurable* (*PiF* {*J*} *M*) (*PiM J M*)
proof (*rule measurable-PiM-single*)
show *proj* $\in$ *space* (*PiF* {*J*} *M*) $\to$ (Π_E $i \in J$. *space* (*M i*))
using *assms* **by** (*auto simp add*: *space-PiM space-PiF extensional-def sets-PiF Pi'-def*)
next
fix *A i* **assume** *A*: $i \in J$ $A \in$ *sets* (*M i*)
show {$\omega \in$ *space* (*PiF* {*J*} *M*). (ω)$_F$ *i* $\in$ *A*} $\in$ *sets* (*PiF* {*J*} *M*)
proof
have {$\omega \in$ *space* (*PiF* {*J*} *M*). (ω)$_F$ *i* $\in$ *A*} =
($\lambda\omega$. (ω)$_F$ *i*) −‘ *A* ∩ *space* (*PiF* {*J*} *M*) **by** *auto*
also have ... $\in$ *sets* (*PiF* {*J*} *M*)
using *assms A* **by** (*auto intro*: *measurable-sets*[*OF measurable-proj-singleton*] *simp*: *space-PiM*)
finally show *?thesis* **.**
qed *simp*
qed

lemma *space-PiF-singleton-eq-product*:
assumes *finite I*
shows *space* (*PiF* {*I*} *M*) = (Π' *i*∈*I*. *space* (*M i*))
by (*auto simp*: *product-def space-PiF assms*)

adapted from *sets* (*Pi$_M$?I ?M*) = *sigma-sets* (Π_E *i*∈*?I*. *space* (*?M i*)) {{*f* $\in$ Π_E *i*∈*?I*. *space* (*?M i*). *f i* $\in$ *A*} |*i A*. *i* $\in$ *?I* ∧ *A* $\in$ *sets* (*?M i*)}

lemma *sets-PiF-single*:
assumes *finite I I* ≠ {}
shows *sets* (*PiF* {*I*} *M*) =
sigma-sets (Π' *i*∈*I*. *space* (*M i*))
{{*f*∈Π' *i*∈*I*. *space* (*M i*). *f i* $\in$ *A*} | *i A*. *i* $\in$ *I* ∧ *A* $\in$ *sets* (*M i*)}
(**is** - = *sigma-sets ?Ω ?R*)
unfolding *sets-PiF-singleton*
proof (*rule sigma-sets-eqI*)
interpret *R*: *sigma-algebra ?Ω sigma-sets ?Ω ?R* **by** (*rule sigma-algebra-sigma-sets*) *auto*
fix *A* **assume** *A* $\in$ {*Pi' I X* |*X*. *X* $\in$ (Π *j*∈*I*. *sets* (*M j*))}
then obtain *X* **where** *X*: *A* = *Pi' I X X* $\in$ (Π *j*∈*I*. *sets* (*M j*)) **by** *auto*
show *A* $\in$ *sigma-sets ?Ω ?R*
proof −
from ‹*I* ≠ {}› *X* **have** *A* = ($\bigcap$*j*∈*I*. {*f*∈*space* (*PiF* {*I*} *M*). *f j* $\in$ *X j*})
using *sets.sets-into-space*
by (*auto simp*: *space-PiF product-def*) *blast*
also have ... $\in$ *sigma-sets ?Ω ?R*
using *X* ‹*I* ≠ {}› *assms* **by** (*intro R.finite-INT*) (*auto simp*: *space-PiF*)
finally show *A* $\in$ *sigma-sets ?Ω ?R* **.**
qed
next

fix *A* **assume** *A* ∈ *?R*
then obtain *i B* **where** *A*: *A* = {*f*∈Π′ *i*∈*I*. *space* (*M i*). *f i* ∈ *B*} *i* ∈ *I B* ∈ *sets* (*M i*)
by *auto*
then have *A* = (Π′ *j* ∈ *I*. *if j* = *i then B else space* (*M j*))
using *sets.sets-into-space*[*OF A*(*3*)]
apply (*auto simp*: *Pi′-iff split*: *if-split-asm*)
apply *blast*
done
also have ... ∈ *sigma-sets ?*Ω {*Pi′ I X* |*X*. *X* ∈ (Π *j*∈*I*. *sets* (*M j*))}
using *A*
by (*intro sigma-sets.Basic*)
(*auto intro*: *exI*[**where** *x*=λ*j*. *if j* = *i then B else space* (*M j*)])
finally show *A* ∈ *sigma-sets ?*Ω {*Pi′ I X* |*X*. *X* ∈ (Π *j*∈*I*. *sets* (*M j*))} .
qed

adapted from (⋀*i*. *i* ∈ *?I* ⟹ *?A i* = *?B i*) ⟹ *Pi*$_E$ *?I ?A* = *Pi*$_E$ *?I ?B*

lemma *Pi′-cong*:
assumes *finite I*
assumes ⋀*i*. *i* ∈ *I* ⟹ *f i* = *g i*
shows *Pi′ I f* = *Pi′ I g*
using *assms* **by** (*auto simp*: *Pi′-def*)

adapted from ⟦*finite ?I*; ⋀*i n m*. ⟦*i* ∈ *?I*; *n* ≤ *m*⟧ ⟹ *?A n i* ⊆ *?A m i*⟧ ⟹ (⋃$_n$ *Pi ?I* (*?A n*)) = (Π *i*∈*?I*. ⋃$_n$ *?A n i*)

lemma *Pi′-UN*:
fixes *A* :: *nat* ⇒ *′i* ⇒ *′a set*
assumes *finite I*
assumes *mono*: ⋀*i n m*. *i* ∈ *I* ⟹ *n* ≤ *m* ⟹ *A n i* ⊆ *A m i*
shows (⋃*n*. *Pi′ I* (*A n*)) = *Pi′ I* (λ*i*. ⋃*n*. *A n i*)
proof (*intro set-eqI iffI*)
fix *f* **assume** *f* ∈ *Pi′ I* (λ*i*. ⋃*n*. *A n i*)
then have ∀ *i*∈*I*. ∃ *n*. *f i* ∈ *A n i domain f* = *I* **by** (*auto simp*: ‹*finite I*› *Pi′-def*)
from *bchoice*[*OF this*(*1*)] **obtain** *n* **where** *n*: ⋀*i*. *i* ∈ *I* ⟹ *f i* ∈ (*A* (*n i*) *i*) **by** *auto*
obtain *k* **where** *k*: ⋀*i*. *i* ∈ *I* ⟹ *n i* ≤ *k*
using ‹*finite I*› *finite-nat-set-iff-bounded-le*[*of n*‘*I*] **by** *auto*
have *f* ∈ *Pi′ I* (λ*i*. *A k i*)
proof
fix *i* **assume** *i* ∈ *I*
from *mono*[*OF this*, *of n i k*] *k*[*OF this*] *n*[*OF this*] ‹*domain f* = *I*› ‹*i* ∈ *I*›
show *f i* ∈ *A k i* **by** (*auto simp*: ‹*finite I*›)
qed (*simp add*: ‹*domain f* = *I*› ‹*finite I*›)
then show *f* ∈ (⋃*n*. *Pi′ I* (*A n*)) **by** *auto*
qed (*auto simp*: *Pi′-def* ‹*finite I*›)

adapted from ⟦⋀*i*. *i* ∈ *?I* ⟹ ∃ *S*⊆*?E i*. *countable S* ∧ *?*Ω *i* = ⋃ *S*; ⋀*i*. *i* ∈ *?I* ⟹ *?E i* ⊆ *Pow* (*?*Ω *i*); ⋀*j*. *j* ∈ *?J* ⟹ *finite j*; ⋃ *?J* = *?I*⟧ ⟹ *sets* (*Pi*$_M$ *?I* (λ*i*. *sigma* (*?*Ω *i*) (*?E i*))) = *sets* (*sigma* (*Pi*$_E$ *?I ?*Ω) {{*f* ∈

$Pi_E\ ?I\ ?\Omega.\ \forall\, i{\in}j.\ f\ i \in A\ i\}\ |A\ j.\ j \in\ ?J \wedge A \in Pi\ j\ ?E\})$

lemma *sigma-fprod-algebra-sigma-eq*:
fixes $E :: 'i \Rightarrow 'a\ set\ set$ **and** $S :: 'i \Rightarrow nat \Rightarrow 'a\ set$
assumes [*simp*]: *finite I* $I \neq \{\}$
and *S-union*: $\bigwedge i.\ i \in I \Longrightarrow (\bigcup j.\ S\ i\ j) = space\ (M\ i)$
and *S-in-E*: $\bigwedge i.\ i \in I \Longrightarrow range\ (S\ i) \subseteq E\ i$
assumes *E-closed*: $\bigwedge i.\ i \in I \Longrightarrow E\ i \subseteq Pow\ (space\ (M\ i))$
and *E-generates*: $\bigwedge i.\ i \in I \Longrightarrow sets\ (M\ i) = sigma\text{-}sets\ (space\ (M\ i))\ (E\ i)$
defines $P == \{\ Pi'\ I\ F \mid F.\ \forall\, i{\in}I.\ F\ i \in E\ i\ \}$
shows $sets\ (PiF\ \{I\}\ M) = sigma\text{-}sets\ (space\ (PiF\ \{I\}\ M))\ P$
proof
let $?P = sigma\ (space\ (Pi_F\ \{I\}\ M))\ P$
from ‹*finite I*›[*THEN ex-bij-betw-finite-nat*] **obtain** T **where** *bij-betw T I* $\{0..<card\ I\}$ **..**
then have T: $\bigwedge i.\ i \in I \Longrightarrow T\ i < card\ I\ \bigwedge i.\ i{\in}I \Longrightarrow the\text{-}inv\text{-}into\ I\ T\ (T\ i) = i$
by (*auto simp add*: *bij-betw-def set-eq-iff image-iff the-inv-into-f-f simp del*: ‹*finite I*›)
have *P-closed*: $P \subseteq Pow\ (space\ (Pi_F\ \{I\}\ M))$
using *E-closed* **by** (*auto simp*: *space-PiF P-def Pi'-iff subset-eq*)
then have *space-P*: $space\ ?P = (\Pi'\ i{\in}I.\ space\ (M\ i))$
by (*simp add*: *space-PiF*)
have $sets\ (PiF\ \{I\}\ M) =$
$sigma\text{-}sets\ (space\ ?P)\ \{\{f \in \Pi'\ i{\in}I.\ space\ (M\ i).\ f\ i \in A\}\ |i\ A.\ i \in I \wedge A \in sets\ (M\ i)\}$
using *sets-PiF-single*[*of I M*] **by** (*simp add*: *space-P*)
also have $\ldots \subseteq sets\ (sigma\ (space\ (PiF\ \{I\}\ M))\ P)$
proof (*safe intro*!: *sets.sigma-sets-subset*)
fix *i A* **assume** $i \in I$ **and** A: $A \in sets\ (M\ i)$
have $(\lambda x.\ (x)_F\ i) \in measurable\ ?P\ (sigma\ (space\ (M\ i))\ (E\ i))$
proof (*subst measurable-iff-measure-of*)
show $E\ i \subseteq Pow\ (space\ (M\ i))$ **using** ‹$i \in I$› **by** *fact*
from *space-P* ‹$i \in I$› **show** $(\lambda x.\ (x)_F\ i) \in space\ ?P \to space\ (M\ i)$
by *auto*
show $\forall\, A{\in}E\ i.\ (\lambda x.\ (x)_F\ i) -` A \cap space\ ?P \in sets\ ?P$
proof
fix A **assume** A: $A \in E\ i$
from T **have** $*$: $\exists xs.\ length\ xs = card\ I \wedge (\forall\, j{\in}I.\ (g)_F\ j \in (if\ i = j\ then\ A\ else\ S\ j\ (xs\ !\ T\ j)))$
if *domain* $g = I$ **and** $\forall\, j{\in}I.\ (g)_F\ j \in (if\ i = j\ then\ A\ else\ S\ j\ (f\ j))$ **for** *g f*
using *that* **by** (*auto intro*!: *exI* [*of - map* $(\lambda n.\ f\ (the\text{-}inv\text{-}into\ I\ T\ n))$ $[0..<card\ I]$])
from *A* **have** $(\lambda x.\ (x)_F\ i) -` A \cap space\ ?P = (\Pi'\ j{\in}I.\ if\ i = j\ then\ A\ else\ space\ (M\ j))$
using *E-closed* ‹$i \in I$› **by** (*auto simp*: *space-P Pi-iff subset-eq split*: *if-split-asm*)
also have $\ldots = (\Pi'\ j{\in}I.\ \bigcup n.\ if\ i = j\ then\ A\ else\ S\ j\ n)$
by (*intro Pi'-cong*) (*simp-all add*: *S-union*)
also have $\ldots = \{g.\ domain\ g = I \wedge (\exists f.\ \forall\, j{\in}I.\ (g)_F\ j \in (if\ i = j\ then\ A$

else S j (f j)))}
by (*rule set-eqI*) (*simp del*: *if-image-distrib add*: *Pi'-iff bchoice-iff*)
also have ... = (⋃ *xs*∈{*xs*. *length xs* = *card I*}. Π′ *j*∈*I*. *if i* = *j then A else S j* (*xs* ! *T j*))
using ∗ **by** (*auto simp add*: *Pi'-iff split del*: *if-split*)
also have ... ∈ *sets ?P*
proof (*safe intro*!: *sets.countable-UN*)
fix *xs* **show** (Π′ *j*∈*I*. *if i* = *j then A else S j* (*xs* ! *T j*)) ∈ *sets ?P*
using *A S-in-E*
by (*simp add*: *P-closed*)
(*auto simp*: *P-def subset-eq intro*!: *exI*[*of* - λ*j*. *if i* = *j then A else S j* (*xs* ! *T j*)])
qed
finally show (λ*x*. (*x*)$_F$ *i*) −‘ *A* ∩ *space ?P* ∈ *sets ?P*
using *P-closed* **by** *simp*
qed
qed
from *measurable-sets*[*OF this*, *of A*] *A* ‹*i* ∈ *I*› *E-closed*
have (λ*x*. (*x*)$_F$ *i*) −‘ *A* ∩ *space ?P* ∈ *sets ?P*
by (*simp add*: *E-generates*)
also have (λ*x*. (*x*)$_F$ *i*) −‘ *A* ∩ *space ?P* = {*f* ∈ Π′ *i*∈*I*. *space* (*M i*). *f i* ∈ *A*}
using *P-closed* **by** (*auto simp*: *space-PiF*)
finally show ... ∈ *sets ?P* **.**
qed
finally show *sets* (*PiF* {*I*} *M*) ⊆ *sigma-sets* (*space* (*PiF* {*I*} *M*)) *P*
by (*simp add*: *P-closed*)
show *sigma-sets* (*space* (*PiF* {*I*} *M*)) *P* ⊆ *sets* (*PiF* {*I*} *M*)
using ‹*finite I*› ‹*I* ≠ {}›
by (*auto intro*!: *sets.sigma-sets-subset product-in-sets-PiFI simp*: *E-generates P-def*)
qed

lemma *product-open-generates-sets-PiF-single*:
assumes *I* ≠ {}
assumes [*simp*]: *finite I*
shows *sets* (*PiF* {*I*} (λ-. *borel*::′*b*::*second-countable-topology measure*)) =
sigma-sets (*space* (*PiF* {*I*} (λ-. *borel*))) {*Pi′ I F* |*F*. (∀ *i*∈*I*. *F i* ∈ *Collect open*)}
proof −
from *open-countable-basisE*[*OF open-UNIV*] **obtain** *S*::′*b set set*
where *S*: *S* ⊆ (*SOME B*. *countable B* ∧ *topological-basis B*) *UNIV* = ⋃ *S*
by *auto*
show *?thesis*
proof (*rule sigma-fprod-algebra-sigma-eq*)
show *finite I* **by** *simp*
show *I* ≠ {} **by** *fact*
define *S′* **where** *S′* = *from-nat-into S*
show (⋃ *j*. *S′ j*) = *space borel*
using *S*

apply (*auto simp add*: *from-nat-into countable-basis-proj S'-def basis-proj-def*)
apply (*metis* (*lifting*, *mono-tags*) *UNIV-I UnionE basis-proj-def countable-basis-proj countable-subset from-nat-into-surj*)
done
show *range S'* ⊆ *Collect open*
using *S*
apply (*auto simp add*: *from-nat-into countable-basis-proj S'-def*)
apply (*metis UNIV-not-empty Union-empty from-nat-into subsetD topological-basis-open*[*OF basis-proj*] *basis-proj-def*)
done
show *Collect open* ⊆ *Pow* (*space borel*) **by** *simp*
show *sets borel* = *sigma-sets* (*space borel*) (*Collect open*)
by (*simp add*: *borel-def*)
qed
qed

lemma *finmap-UNIV*[*simp*]: (⋃ *J*∈*Collect finite*. Π′ *j*∈*J*. *UNIV*) = *UNIV* **by** *auto*

lemma *borel-eq-PiF-borel*:
shows (*borel* :: ('*i*::*countable* ⇒$_F$ '*a*::*polish-space*) *measure*) =
PiF (*Collect finite*) (λ-. *borel* :: '*a measure*)
unfolding *borel-def PiF-def*
proof (*rule measure-eqI*, *clarsimp*, *rule sigma-sets-eqI*)
fix *a*::('*i* ⇒$_F$ '*a*) *set* **assume** *a* ∈ *Collect open* **hence** *open a* **by** *simp*
then obtain *B'* **where** *B'*: *B'*⊆*basis-finmap a* = ⋃ *B'*
using *finmap-topological-basis* **by** (*force simp add*: *topological-basis-def*)
have *a* ∈ *sigma UNIV* {*Pi' J X* |*X J*. *finite J* ∧ *X* ∈ *J* → *sigma-sets UNIV* (*Collect open*)}
unfolding ‹*a* = ⋃ *B'*›
proof (*rule sets.countable-Union*)
from *B' countable-basis-finmap* **show** *countable B'* **by** (*metis countable-subset*)
next
show *B'* ⊆ *sets* (*sigma UNIV*
{*Pi' J X* |*X J*. *finite J* ∧ *X* ∈ *J* → *sigma-sets UNIV* (*Collect open*)}) (**is** - ⊆ *sets ?s*)
proof
fix *x* **assume** *x* ∈ *B'* **with** *B'* **have** *x* ∈ *basis-finmap* **by** *auto*
then obtain *J X* **where** *x* = *Pi' J X finite J X* ∈ *J* → *sigma-sets UNIV* (*Collect open*)
by (*auto simp*: *basis-finmap-def topological-basis-open*[*OF basis-proj*])
thus *x* ∈ *sets ?s* **by** *auto*
qed
qed
thus *a* ∈ *sigma-sets UNIV* {*Pi' J X* |*X J*. *finite J* ∧ *X* ∈ *J* → *sigma-sets UNIV* (*Collect open*)}
by *simp*
next
fix *b*::('*i* ⇒$_F$ '*a*) *set*

assume $b \in \{Pi'\ J\ X\ |X\ J.\ finite\ J \wedge X \in J \to sigma\text{-}sets\ UNIV\ (Collect\ open)\}$
hence b': $b \in sets\ (Pi_F\ (Collect\ finite)\ (\lambda\text{-}.\ borel))$ **by** (*auto simp*: *sets-PiF borel-def*)
let *?b* $= \lambda J.\ b \cap \{x.\ domain\ x = J\}$
have $b = \bigcup((\lambda J.\ ?b\ J)\ `\ Collect\ finite)$ **by** *auto*
also have $\ldots \in sets\ borel$
proof (*rule sets.countable-Union*, *safe*)
fix $J::'i\ set$ **assume** *finite J*
{ assume *ef*: $J = \{\}$
have *?b J* $\in$ *sets borel*
proof *cases*
assume *?b J* $\neq \{\}$
then obtain *f* **where** $f \in b$ *domain f* $= \{\}$ **using** *ef* **by** *auto*
hence *?b J* $= \{f\}$ **using** ‹$J = \{\}$›
by (*auto simp*: *finmap-eq-iff*)
also have $\{f\} \in sets\ borel$ **by** *simp*
finally show *?thesis* **.**
qed *simp*
} moreover {
assume $J \neq (\{\}::'i\ set)$
have (*?b J*) $= b \cap \{m.\ domain\ m \in \{J\}\}$ **by** *auto*
also have $\ldots \in sets\ (PiF\ \{J\}\ (\lambda\text{-}.\ borel))$
using *b'* **by** (*rule restrict-sets-measurable*) (*auto simp*: ‹*finite J*›)
also have $\ldots = sigma\text{-}sets\ (space\ (PiF\ \{J\}\ (\lambda\text{-}.\ borel)))$
$\{Pi'\ (J)\ F\ |F.\ (\forall j \in J.\ F\ j \in Collect\ open)\}$
(**is** - = *sigma-sets* - *?P*)
by (*rule product-open-generates-sets-PiF-single*[*OF* ‹$J \neq \{\}$› ‹*finite J*›])
also have $\ldots \subseteq sigma\text{-}sets\ UNIV\ (Collect\ open)$
by (*intro sigma-sets-mono''*) (*auto intro*!: *open-Pi'I simp*: *space-PiF*)
finally have (*?b J*) $\in$ *sets borel* **by** (*simp add*: *borel-def*)
} ultimately show (*?b J*) $\in$ *sets borel* **by** *blast*
qed (*simp add*: *countable-Collect-finite*)
finally show $b \in sigma\text{-}sets\ UNIV\ (Collect\ open)$ **by** (*simp add*: *borel-def*)
qed (*simp add*: *emeasure-sigma borel-def PiF-def*)

22.10 Isomorphism between Functions and Finite Maps

lemma *measurable-finmap-compose*:
shows $(\lambda m.\ compose\ J\ m\ f) \in measurable\ (PiM\ (f\ `\ J)\ (\lambda\text{-}.\ M))\ (PiM\ J\ (\lambda\text{-}.\ M))$
unfolding *compose-def* **by** *measurable*

lemma *measurable-compose-inv*:
assumes *inj*: $\bigwedge j.\ j \in J \Longrightarrow f'\ (f\ j) = j$
shows $(\lambda m.\ compose\ (f\ `\ J)\ m\ f') \in measurable\ (PiM\ J\ (\lambda\text{-}.\ M))\ (PiM\ (f\ `\ J)\ (\lambda\text{-}.\ M))$
unfolding *compose-def* **by** (*rule measurable-restrict*) (*auto simp*: *inj*)

locale *function-to-finmap* =

fixes J::$'a$ *set* **and** f :: $'a \Rightarrow 'b$::*countable* **and** f'
assumes [*simp*]: *finite* J
assumes *inv*: $i \in J \Longrightarrow f'\ (f\ i) = i$
begin

to measure finmaps

definition $fm = (\mathit{finmap\text{-}of}\ (f\ `\ J))\ o\ (\lambda g.\ \mathit{compose}\ (f\ `\ J)\ g\ f')$

lemma *domain-fm*[*simp*]: $\mathit{domain}\ (fm\ x) = f\ `\ J$
unfolding *fm-def* **by** *simp*

lemma *fm-restrict*[*simp*]: $fm\ (\mathit{restrict}\ y\ J) = fm\ y$
unfolding *fm-def* **by** (*auto simp*: *compose-def inv intro*: *restrict-ext*)

lemma *fm-product*:
assumes $\bigwedge i.\ \mathit{space}\ (M\ i) = \mathit{UNIV}$
shows $fm\ -`\ Pi'\ (f\ `\ J)\ S \cap \mathit{space}\ (Pi_M\ J\ M) = (\Pi_E\ j \in J.\ S\ (f\ j))$
using *assms*
by (*auto simp*: *inv fm-def compose-def space-PiM Pi'-def*)

lemma *fm-measurable*:
assumes $f\ `\ J \in N$
shows $fm \in \mathit{measurable}\ (Pi_M\ J\ (\lambda\text{-}.\ M))\ (Pi_F\ N\ (\lambda\text{-}.\ M))$
unfolding *fm-def*
proof (*rule measurable-comp*, *rule measurable-compose-inv*)
show $\mathit{finmap\text{-}of}\ (f\ `\ J) \in \mathit{measurable}\ (Pi_M\ (f\ `\ J)\ (\lambda\text{-}.\ M))\ (PiF\ N\ (\lambda\text{-}.\ M))$
using *assms* **by** (*intro measurable-finmap-of measurable-component-singleton*)
auto
qed (*simp-all add*: *inv*)

lemma *proj-fm*:
assumes $x \in J$
shows $fm\ m\ (f\ x) = m\ x$
using *assms* **by** (*auto simp*: *fm-def compose-def o-def inv*)

lemma *inj-on-compose-f'*: $\mathit{inj\text{-}on}\ (\lambda g.\ \mathit{compose}\ (f\ `\ J)\ g\ f')\ (\mathit{extensional}\ J)$
proof (*rule inj-on-inverseI*)
fix x::$'a \Rightarrow 'c$ **assume** $x \in \mathit{extensional}\ J$
thus $(\lambda x.\ \mathit{compose}\ J\ x\ f)\ (\mathit{compose}\ (f\ `\ J)\ x\ f') = x$
by (*auto simp*: *compose-def inv extensional-def*)
qed

lemma *inj-on-fm*:
assumes $\bigwedge i.\ \mathit{space}\ (M\ i) = \mathit{UNIV}$
shows $\mathit{inj\text{-}on}\ fm\ (\mathit{space}\ (Pi_M\ J\ M))$
using *assms*
apply (*auto simp*: *fm-def space-PiM PiE-def*)
apply (*rule comp-inj-on*)
apply (*rule inj-on-compose-f'*)

apply (*rule finmap-of-inj-on-extensional-finite*)
apply *simp*
apply (*auto*)
done

to measure functions

definition *mf* = (λ*g*. *compose J g f*) *o proj*

lemma *mf-fm*:
assumes *x* ∈ *space* (*Pi*$_M$ *J* (λ-. *M*))
shows *mf* (*fm x*) = *x*
proof −
have *mf* (*fm x*) ∈ *extensional J*
by (*auto simp*: *mf-def extensional-def compose-def*)
moreover
have *x* ∈ *extensional J* **using** *assms sets.sets-into-space*
by (*force simp*: *space-PiM PiE-def*)
moreover
{ fix *i* **assume** *i* ∈ *J*
hence *mf* (*fm x*) *i* = *x i*
by (*auto simp*: *inv mf-def compose-def fm-def*)
}
ultimately
show *?thesis* **by** (*rule extensionalityI*)
qed

lemma *mf-measurable*:
assumes *space M* = *UNIV*
shows *mf* ∈ *measurable* (*PiF* {*f* ‘ *J*} (λ-. *M*)) (*PiM J* (λ-. *M*))
unfolding *mf-def*
proof (*rule measurable-comp*, *rule measurable-proj-PiM*)
show (λ*g*. *compose J g f*) ∈ *measurable* (*Pi*$_M$ (*f* ‘ *J*) (λ*x*. *M*)) (*Pi*$_M$ *J* (λ-. *M*))
by (*rule measurable-finmap-compose*)
qed (*auto simp add*: *space-PiM extensional-def assms*)

lemma *fm-image-measurable*:
assumes *space M* = *UNIV*
assumes *X* ∈ *sets* (*Pi*$_M$ *J* (λ-. *M*))
shows *fm* ‘ *X* ∈ *sets* (*PiF* {*f* ‘ *J*} (λ-. *M*))
proof −
have *fm* ‘ *X* = (*mf*) −‘ *X* ∩ *space* (*PiF* {*f* ‘ *J*} (λ-. *M*))
proof *safe*
fix *x* **assume** *x* ∈ *X*
with *mf-fm*[*of x*] *sets.sets-into-space*[*OF assms*(*2*)] **show** *fm x* ∈ *mf* −‘ *X* **by** *auto*
show *fm x* ∈ *space* (*PiF* {*f* ‘ *J*} (λ-. *M*)) **by** (*simp add*: *space-PiF assms*)
next
fix *y x*
assume *x*: *mf y* ∈ *X*

assume *y*: $y \in space\ (PiF\ \{f\ `\ J\}\ (\lambda\text{-}.\ M))$
thus $y \in fm\ `\ X$
by (*intro image-eqI*[*OF* - *x*], *unfold finmap-eq-iff*)
(*auto simp*: *space-PiF fm-def mf-def compose-def inv Pi'-def*)
qed
also have $\ldots \in sets\ (PiF\ \{f\ `\ J\}\ (\lambda\text{-}.\ M))$
using *assms*
by (*intro measurable-sets*[*OF mf-measurable*]) *auto*
finally show *?thesis* **.**
qed

lemma *fm-image-measurable-finite*:
assumes $space\ M = UNIV$
assumes $X \in sets\ (Pi_M\ J\ (\lambda\text{-}.\ M::'c\ measure))$
shows $fm\ `\ X \in sets\ (PiF\ (Collect\ finite)\ (\lambda\text{-}.\ M::'c\ measure))$
using *fm-image-measurable*[*OF assms*]
by (*rule subspace-set-in-sets*) (*auto simp*: *finite-subset*)

measure on finmaps

definition $mapmeasure\ M\ N = distr\ M\ (PiF\ (Collect\ finite)\ N)\ (fm)$

lemma *sets-mapmeasure*[*simp*]: $sets\ (mapmeasure\ M\ N) = sets\ (PiF\ (Collect\ finite)\ N)$
unfolding *mapmeasure-def* **by** *simp*

lemma *space-mapmeasure*[*simp*]: $space\ (mapmeasure\ M\ N) = space\ (PiF\ (Collect\ finite)\ N)$
unfolding *mapmeasure-def* **by** *simp*

lemma *mapmeasure-PiF*:
assumes *s1*: $space\ M = space\ (Pi_M\ J\ (\lambda\text{-}.\ N))$
assumes *s2*: $sets\ M = sets\ (Pi_M\ J\ (\lambda\text{-}.\ N))$
assumes $space\ N = UNIV$
assumes $X \in sets\ (PiF\ (Collect\ finite)\ (\lambda\text{-}.\ N))$
shows $emeasure\ (mapmeasure\ M\ (\lambda\text{-}.\ N))\ X = emeasure\ M\ ((fm\ -`\ X \cap extensional\ J))$
using *assms*
by (*auto simp*: *measurable-cong-sets*[*OF s2 refl*] *mapmeasure-def emeasure-distr fm-measurable space-PiM PiE-def*)

lemma *mapmeasure-PiM*:
fixes $N::'c\ measure$
assumes *s1*: $space\ M = space\ (Pi_M\ J\ (\lambda\text{-}.\ N))$
assumes *s2*: $sets\ M = (Pi_M\ J\ (\lambda\text{-}.\ N))$
assumes *N*: $space\ N = UNIV$
assumes *X*: $X \in sets\ M$
shows $emeasure\ M\ X = emeasure\ (mapmeasure\ M\ (\lambda\text{-}.\ N))\ (fm\ `\ X)$
unfolding *mapmeasure-def*
proof (*subst emeasure-distr*, *subst measurable-cong-sets*[*OF s2 refl*], *rule fm-measurable*)

have $X \subseteq$ *space* $(Pi_M\ J\ (\lambda\text{-}.\ N))$ **using** *assms* **by** (*simp add*: *sets.sets-into-space*)
from *assms inj-on-fm*[*of* $\lambda\text{-}.\ N$] *subsetD*[*OF this*] **have** $fm\ -`\ fm\ `\ X \cap$ *space* $(Pi_M\ J\ (\lambda\text{-}.\ N)) = X$
by (*auto simp*: *vimage-image-eq inj-on-def*)
thus *emeasure* $M\ X =$ *emeasure* $M\ (fm\ -`\ fm\ `\ X \cap$ *space* $M)$ **using** *s1*
by *simp*
show $fm\ `\ X \in$ *sets* $(PiF\ (Collect\ finite)\ (\lambda\text{-}.\ N))$
by (*rule fm-image-measurable-finite*[*OF N X*[*simplified s2*]])
qed *simp*

end

end

23 Projective Limit

theory *Projective-Limit*
imports
Fin-Map
Infinite-Product-Measure
HOL−Library.Diagonal-Subsequence
begin

23.1 Sequences of Finite Maps in Compact Sets

locale *finmap-seqs-into-compact* =
fixes K::$nat \Rightarrow (nat \Rightarrow_F\ 'a$::*metric-space*$)\ set$ **and** f::$nat \Rightarrow (nat \Rightarrow_F\ 'a)$ **and** M
assumes *compact*: $\bigwedge n.\ compact\ (K\ n)$
assumes *f-in-K*: $\bigwedge n.\ K\ n \neq \{\}$
assumes *domain-K*: $\bigwedge n.\ k \in K\ n \Longrightarrow domain\ k = domain\ (f\ n)$
assumes *proj-in-K*:
$\bigwedge t\ n\ m.\ m \geq n \Longrightarrow t \in domain\ (f\ n) \Longrightarrow (f\ m)_F\ t \in (\lambda k.\ (k)_F\ t)\ `\ K\ n$
begin

lemma *proj-in-K'*: $(\exists n.\ \forall m \geq n.\ (f\ m)_F\ t \in (\lambda k.\ (k)_F\ t)\ `\ K\ n)$
using *proj-in-K f-in-K*
proof *cases*
obtain k **where** $k \in K\ (Suc\ 0)$ **using** *f-in-K* **by** *auto*
assume $\forall n.\ t \notin domain\ (f\ n)$
thus *?thesis*
by (*auto intro*!: *exI*[**where** *x=1*] *image-eqI*[*OF* - ‹$k \in K\ (Suc\ 0)$›]
simp: *domain-K*[*OF* ‹$k \in K\ (Suc\ 0)$›])
qed *blast*

lemma *proj-in-KE*:
obtains n **where** $\bigwedge m.\ m \geq n \Longrightarrow (f\ m)_F\ t \in (\lambda k.\ (k)_F\ t)\ `\ K\ n$
using *proj-in-K'* **by** *blast*

lemma *compact-projset*:
shows *compact* $((\lambda k.\ (k)_F\ i)\ `\ K\ n)$
using *continuous-proj compact* **by** (*rule compact-continuous-image*)

end

lemma *compactE'*:
fixes $S :: 'a :: metric\text{-}space\ set$
assumes *compact* $S\ \forall\, n{\geq}m.\ f\ n \in S$
obtains $l\ r$ **where** $l \in S$ *strict-mono* $(r{::}nat{\Rightarrow}nat)$ $((f \circ r) \longrightarrow l)$ *sequentially*
proof *atomize-elim*
have *strict-mono* $((+)\ m)$ **by** (*simp add*: *strict-mono-def*)
have $\forall\, n.\ (f\ o\ (\lambda i.\ m + i))\ n \in S$ **using** *assms* **by** *auto*
from *seq-compactE*[*OF* ‹*compact S*›[*unfolded compact-eq-seq-compact-metric*] *this*]
obtain $l\ r$ **where** $l \in S$ *strict-mono* r $(f \circ (+)\ m \circ r) \longrightarrow l$ **by** *blast*
hence $l \in S$ *strict-mono* $((\lambda i.\ m + i)\ o\ r) \wedge (f \circ ((\lambda i.\ m + i)\ o\ r)) \longrightarrow l$
using *strict-mono-o*[*OF* ‹*strict-mono* $((+)\ m)$› ‹*strict-mono r*›] **by** (*auto simp*: *o-def*)
thus $\exists\, l\ r.\ l \in S \wedge$ *strict-mono* $r \wedge (f \circ r) \longrightarrow l$ **by** *blast*
qed

sublocale *finmap-seqs-into-compact* $\subseteq$ *subseqs* $\lambda n\ s.\ (\exists\, l.\ (\lambda i.\ ((f\ o\ s)\ i)_F\ n) \longrightarrow l)$
proof
fix n **and** $s :: nat \Rightarrow nat$
assume *strict-mono s*
from *proj-in-KE*[*of n*] **obtain** *n0* **where** *n0*: $\bigwedge m.\ n0 \leq m \Longrightarrow (f\ m)_F\ n \in (\lambda k.\ (k)_F\ n)\ `\ K\ n0$
by *blast*
have $\forall\, i \geq n0.\ ((f \circ s)\ i)_F\ n \in (\lambda k.\ (k)_F\ n)\ `\ K\ n0$
proof *safe*
fix i **assume** $n0 \leq i$
also have $\ldots \leq s\ i$ **by** (*rule seq-suble*) *fact*
finally have $n0 \leq s\ i$ **.**
with *n0* **show** $((f \circ s)\ i)_F\ n \in (\lambda k.\ (k)_F\ n)\ `\ K\ n0$
by *auto*
qed
then obtain *ls rs*
where $ls \in (\lambda k.\ (k)_F\ n)\ `\ K\ n0$ *strict-mono rs* $((\lambda i.\ ((f \circ s)\ i)_F\ n) \circ rs) \longrightarrow ls$
by (*rule compactE'*[*OF compact-projset*])
thus $\exists\, r'.$ *strict-mono* $r' \wedge (\exists\, l.\ (\lambda i.\ ((f \circ (s \circ r'))\ i)_F\ n) \longrightarrow l)$ **by** (*auto simp*: *o-def*)
qed

lemma (**in** *finmap-seqs-into-compact*) *diagonal-tendsto*: $\exists\, l.\ (\lambda i.\ (f\ (diagseq\ i))_F\ n) \longrightarrow l$
proof −
obtain l **where** $(\lambda i.\ ((f\ o\ (diagseq\ o\ (+)\ (Suc\ n)))\ i)_F\ n) \longrightarrow l$

proof (*atomize-elim*, *rule diagseq-holds*)
fix *r s n*
assume *strict-mono* (*r* :: *nat* ⇒ *nat*)
assume ∃ *l*. (λ*i*. ((*f* ∘ *s*) *i*)$_F$ *n*) ⟶ *l*
then obtain *l* **where** ((λ*i*. (*f i*)$_F$ *n*) *o s*) ⟶ *l*
by (*auto simp*: *o-def*)
hence ((λ*i*. (*f i*)$_F$ *n*) *o s o r*) ⟶ *l* **using** ‹*strict-mono r*›
by (*rule LIMSEQ-subseq-LIMSEQ*)
thus ∃ *l*. (λ*i*. ((*f* ∘ (*s* ∘ *r*)) *i*)$_F$ *n*) ⟶ *l* **by** (*auto simp add*: *o-def*)
qed
hence (λ*i*. ((*f* (*diagseq* (*i* + *Suc n*))))$_F$ *n*) ⟶ *l* **by** (*simp add*: *ac-simps*)
hence (λ*i*. (*f* (*diagseq i*))$_F$ *n*) ⟶ *l* **by** (*rule LIMSEQ-offset*)
thus *?thesis* **..**
qed

23.2 Daniell-Kolmogorov Theorem

Existence of Projective Limit

locale *polish-projective* = *projective-family I P* λ-. *borel*::′*a*::*polish-space measure*
for *I*::′*i set* **and** *P*
begin

lemma *emeasure-lim-emb*:
assumes *X*: *J* ⊆ *I finite J X* ∈ *sets* (Π$_M$ *i*∈*J*. *borel*)
shows *lim* (*emb I J X*) = *P J X*
proof (*rule emeasure-lim*)
write *mu-G* (μ*G*)
interpret *generator*: *algebra space* (*PiM I* (λ*i*. *borel*)) *generator*
by (*rule algebra-generator*)

fix *J* **and** *B* :: *nat* ⇒ (′*i* ⇒ ′*a*) *set*
assume *J*: ⋀*n*. *finite* (*J n*) ⋀*n*. *J n* ⊆ *I* ⋀*n*. *B n* ∈ *sets* (Π$_M$ *i*∈*J n*. *borel*)
incseq J
and *B*: *decseq* (λ*n*. *emb I* (*J n*) (*B n*))
and *0* < (*INF i*. *P* (*J i*) (*B i*)) (**is** *0* < *?a*)
moreover have *?a* ≤ *1*
using *J* **by** (*auto intro*!: *INF-lower2*[*of 0*] *prob-space-P*[*THEN prob-space.measure-le-1*])
ultimately obtain *r* **where** *r*: *?a* = *ennreal r 0* < *r r* ≤ *1*
by (*cases ?a*) (*auto simp*: *top-unique*)
define *Z* **where** *Z n* = *emb I* (*J n*) (*B n*) **for** *n*
have *Z-mono*: *n* ≤ *m* ⟹ *Z m* ⊆ *Z n* **for** *n m*
unfolding *Z-def* **using** *B*[*THEN antimonoD*, *of n m*] **.**
have *J-mono*: ⋀*n m*. *n* ≤ *m* ⟹ *J n* ⊆ *J m*
using ‹*incseq J*› **by** (*force simp*: *incseq-def*)
note [*simp*] = ‹⋀*n*. *finite* (*J n*)›
interpret *prob-space P* (*J i*) **for** *i* **using** *J prob-space-P* **by** *simp*

have *P-eq*[*simp*]:
sets (*P* (*J i*)) = *sets* (Π$_M$ *i*∈*J i*. *borel*) *space* (*P* (*J i*)) = *space* (Π$_M$ *i*∈*J i*.

borel) **for** *i*
using *J* **by** (*auto simp*: *sets-P space-P*)

have *Z i* ∈ *generator* **for** *i*
unfolding *Z-def* **by** (*auto intro*!: *generator.intros J*)

have *countable-UN-J*: *countable* (⋃ *n*. *J n*) **by** (*simp add*: *countable-finite*)
define *Utn* **where** *Utn* = *to-nat-on* (⋃ *n*. *J n*)
interpret *function-to-finmap J n Utn from-nat-into* (⋃ *n*. *J n*) **for** *n*
by *unfold-locales* (*auto simp*: *Utn-def intro*: *from-nat-into-to-nat-on*[*OF countable-UN-J*])
have *inj-on-Utn*: *inj-on Utn* (⋃ *n*. *J n*)
unfolding *Utn-def* **using** *countable-UN-J* **by** (*rule inj-on-to-nat-on*)
hence *inj-on-Utn-J*: ⋀*n*. *inj-on Utn* (*J n*) **by** (*rule subset-inj-on*) *auto*
define *P'* **where** *P' n* = *mapmeasure n* (*P* (*J n*)) (λ-. *borel*) **for** *n*
interpret *P'*: *prob-space P' n* **for** *n*
unfolding *P'-def mapmeasure-def* **using** *J*
by (*auto intro*!: *prob-space-distr fm-measurable simp*: *measurable-cong-sets*[*OF sets-P*])

let *?SUP* = λ*n*. *SUP K* ∈ {*K*. *K* ⊆ *fm n* ‘ (*B n*) ∧ *compact K*}. *emeasure* (*P' n*) *K*
{ fix *n*
have *emeasure* (*P* (*J n*)) (*B n*) = *emeasure* (*P' n*) (*fm n* ‘ (*B n*))
using *J* **by** (*auto simp*: *P'-def mapmeasure-PiM space-P sets-P*)
also
have ... = *?SUP n*
proof (*rule inner-regular*)
show *sets* (*P' n*) = *sets borel* **by** (*simp add*: *borel-eq-PiF-borel P'-def*)
next
show *fm n* ‘ *B n* ∈ *sets borel*
unfolding *borel-eq-PiF-borel* **by** (*auto simp*: *P'-def fm-image-measurable-finite sets-P J*(*3*))
qed *simp*
finally have ∗: *emeasure* (*P* (*J n*)) (*B n*) = *?SUP n* **.**
have *?SUP n* ≠ ∞
unfolding ∗[*symmetric*] **by** *simp*
note ∗ *this*
} note *R* = *this*
have ∀ *n*. ∃ *K*. *emeasure* (*P* (*J n*)) (*B n*) − *emeasure* (*P' n*) *K* ≤ *2 powr* (−*n*) ∗ *?a* ∧ *compact K* ∧ *K* ⊆ *fm n* ‘ *B n*
proof
fix *n* **show** ∃ *K*. *emeasure* (*P* (*J n*)) (*B n*) − *emeasure* (*P' n*) *K* ≤ *ennreal* (*2 powr* − *real n*) ∗ *?a* ∧
compact K ∧ *K* ⊆ *fm n* ‘ *B n*
unfolding *R*[*of n*]
proof (*rule ccontr*)
assume *H*: ∄ *K'*. *?SUP n* − *emeasure* (*P' n*) *K'* ≤ *ennreal* (*2 powr* − *real n*) ∗ *?a* ∧

```
      compact K' ∧ K' ⊆ fm n ‘ B n
    have ?SUP n + 0 < ?SUP n + 2 powr (−n) * ?a
    using R[of n] unfolding ennreal-add-left-cancel-less ennreal-zero-less-mult-iff
      by (auto intro: ‹0 < ?a›)
    also have ... = (SUP K∈{K. K ⊆ fm n ‘ B n ∧ compact K}. emeasure (P' n) K + 2 powr (−n) * ?a)
      by (rule ennreal-SUP-add-left[symmetric]) auto
    also have ... ≤ ?SUP n
    proof (intro SUP-least)
      fix K assume K ∈ {K. K ⊆ fm n ‘ B n ∧ compact K}
      with H have 2 powr (−n) * ?a < ?SUP n − emeasure (P' n) K
        by auto
      then show emeasure (P' n) K + (2 powr (−n)) * ?a ≤ ?SUP n
          by (subst (asm) less-diff-eq-ennreal) (auto simp: less-top[symmetric]
R(1)[symmetric] ac-simps)
    qed
    finally show False by simp
  qed
 qed
 then obtain K' where K':
  ⋀n. emeasure (P (J n)) (B n) − emeasure (P' n) (K' n) ≤ ennreal (2 powr −
real n) * ?a
  ⋀n. compact (K' n) ⋀n. K' n ⊆ fm n ‘ B n
  unfolding choice-iff by blast
 define K where K n = fm n −‘ K' n ∩ space (Pi_M (J n) (λ-. borel)) for n
 have K-sets: ⋀n. K n ∈ sets (Pi_M (J n) (λ-. borel))
  unfolding K-def
  using compact-imp-closed[OF ‹compact (K' -)›]
  by (intro measurable-sets[OF fm-measurable, of - Collect finite])
    (auto simp: borel-eq-PiF-borel[symmetric])
 have K-B: ⋀n. K n ⊆ B n
 proof
  fix x n assume x ∈ K n
  then have fm-in: fm n x ∈ fm n ‘ B n
    using K' by (force simp: K-def)
  show x ∈ B n
   using ‹x ∈ K n› K-sets sets.sets-into-space J(1,2,3)[of n] inj-on-image-mem-iff[OF
inj-on-fm]
  by (metis (no-types) Int-iff K-def fm-in space-borel)
 qed
 define Z' where Z' n = emb I (J n) (K n) for n
 have Z': ⋀n. Z' n ⊆ Z n
  unfolding Z'-def Z-def
 proof (rule prod-emb-mono, safe)
  fix n x assume x ∈ K n
  hence fm n x ∈ K' n x ∈ space (Pi_M (J n) (λ-. borel))
    by (simp-all add: K-def space-P)
  note this(1)
  also have K' n ⊆ fm n ‘ B n by (simp add: K')
```

finally have *fm n x* ∈ *fm n* ' *B n* **.**
thus *x* ∈ *B n*
proof *safe*
fix *y* **assume** *y*: *y* ∈ *B n*
hence *y* ∈ *space* (*Pi*$_M$ (*J n*) (λ-. *borel*)) **using** *J sets.sets-into-space*[*of B n P* (*J n*)]
by (*auto simp add*: *space-P sets-P*)
assume *fm n x* = *fm n y*
note *inj-onD*[*OF inj-on-fm*[*OF space-borel*],
OF ‹*fm n x* = *fm n y*› ‹*x* ∈ *space* -› ‹*y* ∈ *space* -›]
with *y* **show** *x* ∈ *B n* **by** *simp*
qed
qed
have ⋀*n*. *Z*′ *n* ∈ *generator* **using** *J K*′(*2*) **unfolding** *Z*′*-def*
by (*auto intro*!: *generator.intros measurable-sets*[*OF fm-measurable*[*of - Collect finite*]]
simp: *K-def borel-eq-PiF-borel*[*symmetric*] *compact-imp-closed*)
define *Y* **where** *Y n* = (⋂*i*∈{*1..n*}. *Z*′ *i*) **for** *n*
hence ⋀*n k*. *Y* (*n* + *k*) ⊆ *Y n* **by** (*induct-tac k*) (*auto simp*: *Y-def*)
hence *Y-mono*: ⋀*n m*. *n* ≤ *m* ⟹ *Y m* ⊆ *Y n* **by** (*auto simp*: *le-iff-add*)
have *Y-Z*′: ⋀*n*. *n* ≥ *1* ⟹ *Y n* ⊆ *Z*′ *n* **by** (*auto simp*: *Y-def*)
hence *Y-Z*: ⋀*n*. *n* ≥ *1* ⟹ *Y n* ⊆ *Z n* **using** *Z*′ **by** *auto*

have *Y-notempty*: ⋀*n*. *n* ≥ *1* ⟹ (*Y n*) ≠ {}
proof −
fix *n*::*nat* **assume** *n* ≥ *1* **hence** *Y n* ⊆ *Z n* **by** *fact*
have *Y n* = (⋂*i*∈{*1..n*}. *emb I* (*J n*) (*emb* (*J n*) (*J i*) (*K i*))) **using** *J J-mono*
by (*auto simp*: *Y-def Z*′*-def*)
also have ... = *prod-emb I* (λ-. *borel*) (*J n*) (⋂*i*∈{*1..n*}. *emb* (*J n*) (*J i*) (*K i*))
using ‹*n* ≥ *1*›
by (*subst prod-emb-INT*) *auto*
finally
have *Y-emb*:
Y n = *prod-emb I* (λ-. *borel*) (*J n*) (⋂*i*∈{*1..n*}. *prod-emb* (*J n*) (λ-. *borel*) (*J i*) (*K i*)) **.**
hence *Y n* ∈ *generator* **using** *J J-mono K-sets* ‹*n* ≥ *1*›
by (*auto simp del*: *prod-emb-INT intro*!: *generator.intros*)
have ∗: *μG* (*Z n*) = *P* (*J n*) (*B n*)
unfolding *Z-def* **using** *J* **by** (*intro mu-G-spec*) *auto*
then have *μG* (*Z n*) ≠ ∞ **by** *auto*
note ∗
moreover have ∗: *μG* (*Y n*) = *P* (*J n*) (⋂*i*∈{*Suc 0..n*}. *prod-emb* (*J n*) (λ-. *borel*) (*J i*) (*K i*))
unfolding *Y-emb* **using** *J J-mono K-sets* ‹*n* ≥ *1*› **by** (*subst mu-G-spec*) *auto*
then have *μG* (*Y n*) ≠ ∞ **by** *auto*
note ∗
moreover have *μG* (*Z n* − *Y n*) =
P (*J n*) (*B n* − (⋂*i*∈{*Suc 0..n*}. *prod-emb* (*J n*) (λ-. *borel*) (*J i*) (*K i*)))

unfolding *Z-def Y-emb prod-emb-Diff*[*symmetric*] **using** *J J-mono K-sets* ‹$n \geq 1$›
by (*subst mu-G-spec*) (*auto intro*!: *sets.Diff*)
ultimately
have $\mu G\ (Z\ n) - \mu G\ (Y\ n) = \mu G\ (Z\ n - Y\ n)$
using *J J-mono K-sets* ‹$n \geq 1$›
by (*simp only*: *emeasure-eq-measure Z-def*)
(*auto dest*!: *bspec*[**where** *x*=*n*] *intro*!: *measure-Diff*[*symmetric*] *subsetD*[*OF K-B*]
intro!: *arg-cong*[**where** *f*=*ennreal*]
simp: *extensional-restrict emeasure-eq-measure prod-emb-iff sets-P space-P*
ennreal-minus measure-nonneg)
also have *subs*: $Z\ n - Y\ n \subseteq (\bigcup i{\in}\{1..n\}.\ (Z\ i - Z'\ i))$
using ‹$n \geq 1$› **unfolding** *Y-def UN-extend-simps*(*7*) **by** (*intro UN-mono Diff-mono Z-mono order-refl*) *auto*
have $Z\ n - Y\ n \in$ *generator* $(\bigcup i{\in}\{1..n\}.\ (Z\ i - Z'\ i)) \in$ *generator*
using ‹Z' - $\in$ *generator*› ‹Z - $\in$ *generator*› ‹Y - $\in$ *generator*› **by** *auto*
hence $\mu G\ (Z\ n - Y\ n) \leq \mu G\ (\bigcup i{\in}\{1..n\}.\ (Z\ i - Z'\ i))$
using *subs generator.additive-increasing*[*OF positive-mu-G additive-mu-G*]
unfolding *increasing-def* **by** *auto*
also have $\ldots \leq (\sum\ i{\in}\{1..n\}.\ \mu G\ (Z\ i - Z'\ i))$ **using** ‹Z - $\in$ *generator*› ‹Z' - $\in$ *generator*›
by (*intro generator.subadditive*[*OF positive-mu-G additive-mu-G*]) *auto*
also have $\ldots \leq (\sum\ i{\in}\{1..n\}.\ 2\ powr\ -real\ i * ?a)$
proof (*rule sum-mono*)
fix *i* **assume** $i \in \{1..n\}$ **hence** $i \leq n$ **by** *simp*
have $\mu G\ (Z\ i - Z'\ i) = \mu G\ (prod\text{-}emb\ I\ (\lambda\text{-}.\ borel)\ (J\ i)\ (B\ i - K\ i))$
unfolding *Z'-def Z-def* **by** *simp*
also have $\ldots = P\ (J\ i)\ (B\ i - K\ i)$
using *J K-sets* **by** (*subst mu-G-spec*) *auto*
also have $\ldots = P\ (J\ i)\ (B\ i) - P\ (J\ i)\ (K\ i)$
using *K-sets J* ‹K - $\subseteq B$ -› **by** (*simp add*: *emeasure-Diff*)
also have $\ldots = P\ (J\ i)\ (B\ i) - P'\ i\ (K'\ i)$
unfolding *K-def P'-def*
by (*auto simp*: *mapmeasure-PiF borel-eq-PiF-borel*[*symmetric*]
compact-imp-closed[*OF* ‹*compact* (K' -)›] *space-PiM PiE-def*)
also have $\ldots \leq ennreal\ (2\ powr - real\ i) * ?a$ **using** $K'(1)$[*of i*] **.**
finally show $\mu G\ (Z\ i - Z'\ i) \leq (2\ powr - real\ i) * ?a$ **.**
qed
also have $\ldots = ennreal\ ((\sum\ i{\in}\{1..n\}.\ (2\ powr - enn2real\ i)) * enn2real\ ?a)$
using *r* **by** (*simp add*: *sum-distrib-right ennreal-mult*[*symmetric*])
also have $\ldots < ennreal\ (1 * enn2real\ ?a)$
proof (*intro ennreal-lessI mult-strict-right-mono*)
have $(\sum i = 1..n.\ 2\ powr - real\ i) = (\sum i = 1..{<}Suc\ n.\ (1/2)\ \hat{}\ i)$
by (*rule sum.cong*) (*auto simp*: *powr-realpow powr-divide power-divide powr-minus-divide*)
also have $\{1..{<}Suc\ n\} = \{..{<}Suc\ n\} - \{0\}$ **by** *auto*
also have *sum* ((^) (*1 / 2*::*real*)) ($\{..{<}Suc\ n\} - \{0\}$) =

```
      sum ((^) (1 / 2)) ({..<Suc n}) - 1 by (auto simp: sum-diff1)
    also have ... < 1 by (subst geometric-sum) auto
    finally show (∑ i = 1..n. 2 powr - enn2real i) < 1 by simp
  qed (auto simp: r enn2real-positive-iff)
  also have ... = ?a by (auto simp: r)
  also have ... ≤ μG (Z n)
    using J by (auto intro: INF-lower simp: Z-def mu-G-spec)
  finally have μG (Z n) - μG (Y n) < μG (Z n) .
  hence R: μG (Z n) < μG (Z n) + μG (Y n)
    using ‹μG (Y n) ≠ ∞› by (auto simp: zero-less-iff-neq-zero)
  then have μG (Y n) > 0
    by simp
  thus Y n ≠ {} using positive-mu-G by (auto simp add: positive-def)
 qed
 hence ∀ n∈{1..}. ∃ y. y ∈ Y n by auto
 then obtain y where y: ⋀n. n ≥ 1 ⟹ y n ∈ Y n unfolding bchoice-iff by
force
 {
  fix t and n m::nat
  assume 1 ≤ n n ≤ m hence 1 ≤ m by simp
  from Y-mono[OF ‹m ≥ n›] y[OF ‹1 ≤ m›] have y m ∈ Y n by auto
  also have ... ⊆ Z' n using Y-Z'[OF ‹1 ≤ n›] .
  finally
  have fm n (restrict (y m) (J n)) ∈ K' n
   unfolding Z'-def K-def prod-emb-iff by (simp add: Z'-def K-def prod-emb-iff)
   moreover have finmap-of (J n) (restrict (y m) (J n)) = finmap-of (J n) (y
m)
    using J by (simp add: fm-def)
  ultimately have fm n (y m) ∈ K' n by simp
 } note fm-in-K' = this
 interpret finmap-seqs-into-compact λn. K' (Suc n) λk. fm (Suc k) (y (Suc k))
borel
 proof
  fix n show compact (K' n) by fact
 next
  fix n
  from Y-mono[of n Suc n] y[of Suc n] have y (Suc n) ∈ Y (Suc n) by auto
  also have ... ⊆ Z' (Suc n) using Y-Z' by auto
  finally
  have fm (Suc n) (restrict (y (Suc n)) (J (Suc n))) ∈ K' (Suc n)
   unfolding Z'-def K-def prod-emb-iff by (simp add: Z'-def K-def prod-emb-iff)
  thus K' (Suc n) ≠ {} by auto
  fix k
  assume k ∈ K' (Suc n)
   with K'[of Suc n] sets.sets-into-space have k ∈ fm (Suc n) ' B (Suc n) by
auto
  then obtain b where k = fm (Suc n) b by auto
  thus domain k = domain (fm (Suc n) (y (Suc n)))
    by (simp-all add: fm-def)
```

```
next
  fix t and n m::nat
  assume n ≤ m hence Suc n ≤ Suc m by simp
  assume t ∈ domain (fm (Suc n) (y (Suc n)))
  then obtain j where j: t = Utn j j ∈ J (Suc n) by auto
  hence j ∈ J (Suc m) using J-mono[OF ‹Suc n ≤ Suc m›] by auto
  have img: fm (Suc n) (y (Suc m)) ∈ K' (Suc n) using ‹n ≤ m›
    by (intro fm-in-K') simp-all
  show (fm (Suc m) (y (Suc m)))F t ∈ (λk. (k)F t) ‘ K' (Suc n)
    apply (rule image-eqI[OF - img])
    using ‹j ∈ J (Suc n)› ‹j ∈ J (Suc m)›
    unfolding j by (subst proj-fm, auto)+
qed
have ∀ t. ∃ z. (λi. (fm (Suc (diagseq i)) (y (Suc (diagseq i))))F t) ⟶ z
  using diagonal-tendsto ..
then obtain z where z:
  ⋀t. (λi. (fm (Suc (diagseq i)) (y (Suc (diagseq i))))F t) ⟶ z t
  unfolding choice-iff by blast
{
  fix n :: nat assume n ≥ 1
  have ⋀i. domain (fm n (y (Suc (diagseq i)))) = domain (finmap-of (Utn ‘ J
n) z)
    by simp
  moreover
  {
    fix t
    assume t: t ∈ domain (finmap-of (Utn ‘ J n) z)
    hence t ∈ Utn ‘ J n by simp
    then obtain j where j: t = Utn j j ∈ J n by auto
    have (λi. (fm n (y (Suc (diagseq i))))F t) ⟶ z t
      apply (subst (2) tendsto-iff, subst eventually-sequentially)
    proof safe
      fix e :: real assume 0 < e
      { fix i and x :: 'i ⇒ 'a assume i: i ≥ n
        assume t ∈ domain (fm n x)
        hence t ∈ domain (fm i x) using J-mono[OF ‹i ≥ n›] by auto
        with i have (fm i x)F t = (fm n x)F t
          using j by (auto simp: proj-fm dest!: inj-onD[OF inj-on-Utn])
      } note index-shift = this
      have I: ⋀i. i ≥ n ⟹ Suc (diagseq i) ≥ n
        apply (rule le-SucI)
        apply (rule order-trans) apply simp
        apply (rule seq-suble[OF subseq-diagseq])
        done
      from z
      have ∃ N. ∀ i≥N. dist ((fm (Suc (diagseq i)) (y (Suc (diagseq i))))F t) (z
t) < e
        unfolding tendsto-iff eventually-sequentially using ‹0 < e› by auto
      then obtain N where N: ⋀i. i ≥ N ⟹
```

dist ((*fm* (*Suc* (*diagseq i*)) (*y* (*Suc* (*diagseq i*))))$_F$ *t*) (*z t*) < *e* **by** *auto*
show ∃ *N*. ∀ *na*≥*N*. *dist* ((*fm n* (*y* (*Suc* (*diagseq na*))))$_F$ *t*) (*z t*) < *e*
proof (*rule exI*[**where** *x*=*max N n*], *safe*)
fix *na* **assume** *max N n* ≤ *na*
hence *dist* ((*fm n* (*y* (*Suc* (*diagseq na*))))$_F$ *t*) (*z t*) =
dist ((*fm* (*Suc* (*diagseq na*)) (*y* (*Suc* (*diagseq na*))))$_F$ *t*) (*z t*) **using** *t*
by (*subst index-shift*[*OF I*]) *auto*
also have ... < *e* **using** ‹*max N n* ≤ *na*› **by** (*intro N*) *simp*
finally show *dist* ((*fm n* (*y* (*Suc* (*diagseq na*))))$_F$ *t*) (*z t*) < *e* **.**
qed
qed
hence (λ*i*. (*fm n* (*y* (*Suc* (*diagseq i*))))$_F$ *t*) ⟶ (*finmap-of* (*Utn* ‘ *J n*) *z*)$_F$ *t*
by (*simp add*: *tendsto-intros*)
} ultimately
have (λ*i*. *fm n* (*y* (*Suc* (*diagseq i*)))) ⟶ *finmap-of* (*Utn* ‘ *J n*) *z*
by (*rule tendsto-finmap*)
hence ((λ*i*. *fm n* (*y* (*Suc* (*diagseq i*)))) *o* (λ*i*. *i* + *n*)) ⟶ *finmap-of* (*Utn* ‘ *J n*) *z*
by (*rule LIMSEQ-subseq-LIMSEQ*) (*simp add*: *strict-mono-def*)
moreover
have (∀ *i*. ((λ*i*. *fm n* (*y* (*Suc* (*diagseq i*)))) *o* (λ*i*. *i* + *n*)) *i* ∈ *K*′ *n*)
apply (*auto simp add*: *o-def intro*!: *fm-in-K*′ ‹1 ≤ *n*› *le-SucI*)
apply (*rule le-trans*)
apply (*rule le-add2*)
using *seq-suble*[*OF subseq-diagseq*]
apply *auto*
done
moreover
from ‹*compact* (*K*′ *n*)› **have** *closed* (*K*′ *n*) **by** (*rule compact-imp-closed*)
ultimately
have *finmap-of* (*Utn* ‘ *J n*) *z* ∈ *K*′ *n*
unfolding *closed-sequential-limits* **by** *blast*
also have *finmap-of* (*Utn* ‘ *J n*) *z* = *fm n* (λ*i*. *z* (*Utn i*))
unfolding *finmap-eq-iff*
proof *clarsimp*
fix *i* **assume** *i*: *i* ∈ *J n*
hence *from-nat-into* (⋃ *n*. *J n*) (*Utn i*) = *i*
unfolding *Utn-def*
by (*subst from-nat-into-to-nat-on*[*OF countable-UN-J*]) *auto*
with *i* **show** *z* (*Utn i*) = (*fm n* (λ*i*. *z* (*Utn i*)))$_F$ (*Utn i*)
by (*simp add*: *finmap-eq-iff fm-def compose-def*)
qed
finally have *fm n* (λ*i*. *z* (*Utn i*)) ∈ *K*′ *n* **.**
moreover
let *?J* = ⋃ *n*. *J n*
have (*?J* ∩ *J n*) = *J n* **by** *auto*
ultimately have *restrict* (λ*i*. *z* (*Utn i*)) (*?J* ∩ *J n*) ∈ *K n*
unfolding *K-def* **by** (*auto simp*: *space-P space-PiM*)

```
    hence restrict (λi. z (Utn i)) ?J ∈ Z' n unfolding Z'-def
      using J by (auto simp: prod-emb-def PiE-def extensional-def)
    also have ... ⊆ Z n using Z' by simp
    finally have restrict (λi. z (Utn i)) ?J ∈ Z n .
  } note in-Z = this
  hence (⋂i∈{1..}. Z i) ≠ {} by auto
  thus (⋂i. Z i) ≠ {}
    using INT-decseq-offset[OF antimonoI[OF Z-mono]] by simp
qed fact+

lemma measure-lim-emb:
  J ⊆ I ⟹ finite J ⟹ X ∈ sets (Π_M i∈J. borel) ⟹ measure lim (emb I J X)
= measure (P J) X
  unfolding measure-def by (subst emeasure-lim-emb) auto

end

hide-const (open) PiF
hide-const (open) Pi_F
hide-const (open) Pi'
hide-const (open) finmap-of
hide-const (open) proj
hide-const (open) domain
hide-const (open) basis-finmap

sublocale polish-projective ⊆ P: prob-space lim
proof
  have ∗: emb I {} {λx. undefined} = space (Π_M i∈I. borel)
    by (auto simp: prod-emb-def space-PiM)
  interpret prob-space P {}
    using prob-space-P by simp
  show emeasure lim (space lim) = 1
    using emeasure-lim-emb[of {} {λx. undefined}] emeasure-space-1
    by (simp add: ∗ PiM-empty space-P)
qed

locale polish-product-prob-space =
  product-prob-space λ-. borel::('a::polish-space) measure I for I::'i set

sublocale polish-product-prob-space ⊆ P: polish-projective I λJ. PiM J (λ-. borel::('a)
measure)
  ..

lemma (in polish-product-prob-space) limP-eq-PiM: lim = PiM I (λ-. borel)
  by (rule PiM-eq) (auto simp: emeasure-PiM emeasure-lim-emb)

end
```

24 Random Permutations

theory *Random-Permutations*
imports
HOL−Combinatorics.Multiset-Permutations
Probability-Mass-Function
begin

Choosing a set permutation (i.e. a distinct list with the same elements as the set) uniformly at random is the same as first choosing the first element of the list and then choosing the rest of the list as a permutation of the remaining set.

lemma *random-permutation-of-set*:
assumes *finite A* $A \neq \{\}$
shows *pmf-of-set* (*permutations-of-set A*) =
do {
$x \leftarrow$ *pmf-of-set A*;
$xs \leftarrow$ *pmf-of-set* (*permutations-of-set* $(A - \{x\})$);
return-pmf $(x \# xs)$
} (**is** *?lhs* = *?rhs*)
proof −
from *assms* **have** *permutations-of-set A* = $(\bigcup x \in A.$ $(\#)$ x ' *permutations-of-set* $(A - \{x\}))$
by (*simp add*: *permutations-of-set-nonempty*)
also from *assms* **have** *pmf-of-set* ... = *?rhs*
by (*subst pmf-of-set-UN*[**where** n = *fact* (*card A* − *1*)])
(*auto simp*: *card-image disjoint-family-on-def map-pmf-def* [*symmetric*] *map-pmf-of-set-inj*)
finally show *?thesis* **.**
qed

A generic fold function that takes a function, an initial state, and a set and chooses a random order in which it then traverses the set in the same fashion as a left fold over a list. We first give a recursive definition.

function *fold-random-permutation* :: $('a \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'b \Rightarrow 'a$ *set* $\Rightarrow$ $'b$ *pmf*
where
fold-random-permutation f x $\{\}$ = *return-pmf x*
| ¬*finite A* $\Longrightarrow$ *fold-random-permutation f x A* = *return-pmf x*
| *finite A* $\Longrightarrow$ $A \neq \{\} \Longrightarrow$
fold-random-permutation f x A =
pmf-of-set A $\gg\!=$ $(\lambda a.$ *fold-random-permutation f* $(f\ a\ x)$ $(A - \{a\}))$
by *simp-all fastforce*
termination proof (*relation Wellfounded.measure* $(\lambda(\text{-},\text{-},A).$ *card A*))
fix *A* :: $'a$ *set* **and** *f* :: $'a \Rightarrow 'b \Rightarrow 'b$ **and** *x* :: $'b$ **and** *y* :: $'a$
assume *A*: *finite A* $A \neq \{\}$ $y \in$ *set-pmf* (*pmf-of-set A*)
then have *card A* > *0* **by** (*simp add*: *card-gt-0-iff*)
with *A* **show** $((f, f\ y\ x, A - \{y\}), f, x, A) \in$ *Wellfounded.measure* $(\lambda(\text{-}, \text{-}, A).$ *card A*)
by *simp*

qed *simp-all*

We can now show that the above recursive definition is equivalent to choosing a random set permutation and folding over it (in any direction).

lemma *fold-random-permutation-foldl*:
assumes *finite A*
shows *fold-random-permutation f x A* =
map-pmf (*foldl* (λ*x y*. *f y x*) *x*) (*pmf-of-set* (*permutations-of-set A*))
using *assms*
proof (*induction f x A rule*: *fold-random-permutation.induct* [*case-names empty infinite remove*])
case (*remove A f x*)
from *remove*
have *fold-random-permutation f x A* =
pmf-of-set A ⤜ (λ*a*. *fold-random-permutation f* (*f a x*) (*A* − {*a*})) **by** *simp*
also from *remove*
have ... = *pmf-of-set A* ⤜ (λ*a*. *map-pmf* (*foldl* (λ*x y*. *f y x*) *x*)
(*map-pmf* ((#) *a*) (*pmf-of-set* (*permutations-of-set* (*A* − {*a*})))))
by (*intro bind-pmf-cong*) (*simp-all add*: *pmf.map-comp o-def*)
also from *remove* **have** ... = *map-pmf* (*foldl* (λ*x y*. *f y x*) *x*) (*pmf-of-set* (*permutations-of-set A*))
by (*simp-all add*: *random-permutation-of-set map-bind-pmf map-pmf-def* [*symmetric*])
finally show *?case* **.**
qed (*simp-all add*: *pmf-of-set-singleton*)

lemma *fold-random-permutation-foldr*:
assumes *finite A*
shows *fold-random-permutation f x A* =
map-pmf (λ*xs*. *foldr f xs x*) (*pmf-of-set* (*permutations-of-set A*))
proof −
have *fold-random-permutation f x A* =
map-pmf (*foldl* (λ*x y*. *f y x*) *x* ∘ *rev*) (*pmf-of-set* (*permutations-of-set A*))
using *assms* **by** (*subst fold-random-permutation-foldl* [*OF assms*])
(*simp-all add*: *pmf.map-comp* [*symmetric*] *map-pmf-of-set-inj*)
also have *foldl* (λ*x y*. *f y x*) *x* ∘ *rev* = (λ*xs*. *foldr f xs x*)
by (*intro ext*) (*simp add*: *foldl-conv-foldr*)
finally show *?thesis* **.**
qed

lemma *fold-random-permutation-fold*:
assumes *finite A*
shows *fold-random-permutation f x A* =
map-pmf (λ*xs*. *fold f xs x*) (*pmf-of-set* (*permutations-of-set A*))
by (*subst fold-random-permutation-foldl* [*OF assms*], *intro map-pmf-cong*)
(*simp-all add*: *foldl-conv-fold*)

lemma *fold-random-permutation-code* [*code*]:
fold-random-permutation f x (*set xs*) =

map-pmf (*foldl* (λ*x y*. *f y x*) *x*) (*pmf-of-set* (*permutations-of-set* (*set xs*)))
by (*simp add*: *fold-random-permutation-foldl*)

We now introduce a slightly generalised version of the above fold operation that does not simply return the result in the end, but applies a monadic bind to it. This may seem somewhat arbitrary, but it is a common use case, e.g. in the Social Decision Scheme of Random Serial Dictatorship, where voters narrow down a set of possible winners in a random order and the winner is chosen from the remaining set uniformly at random.

function *fold-bind-random-permutation*
:: ('*a* ⇒ '*b* ⇒ '*b*) ⇒ ('*b* ⇒ '*c pmf*) ⇒ '*b* ⇒ '*a set* ⇒ '*c pmf* **where**
fold-bind-random-permutation f g x {} = *g x*
| ¬*finite A* ⟹ *fold-bind-random-permutation f g x A* = *g x*
| *finite A* ⟹ *A* ≠ {} ⟹
fold-bind-random-permutation f g x A =
pmf-of-set A ⤜ (λ*a*. *fold-bind-random-permutation f g* (*f a x*) (*A* − {*a*}))
by *simp-all fastforce*
termination proof (*relation Wellfounded.measure* (λ(-,-,-,*A*). *card A*))
fix *A* :: '*a set* **and** *f* :: '*a* ⇒ '*b* ⇒ '*b* **and** *x* :: '*b*
and *y* :: '*a* **and** *g* :: '*b* ⇒ '*c pmf*
assume *A*: *finite A A* ≠ {} *y* ∈ *set-pmf* (*pmf-of-set A*)
then have *card A* > *0* **by** (*simp add*: *card-gt-0-iff*)
with *A* **show** ((*f*, *g*, *f y x*, *A* − {*y*}), *f*, *g*, *x*, *A*) ∈ *Wellfounded.measure* (λ(-, -, -, *A*). *card A*)
by *simp*
qed *simp-all*

We now show that the recursive definition is equivalent to a random fold followed by a monadic bind.

lemma *fold-bind-random-permutation-altdef* [*code*]:
fold-bind-random-permutation f g x A = *fold-random-permutation f x A* ⤜ *g*
proof (*induction f x A rule*: *fold-random-permutation.induct* [*case-names empty infinite remove*])
case (*remove A f x*)
from *remove* **have** *pmf-of-set A* ⤜ (λ*a*. *fold-bind-random-permutation f g* (*f a x*) (*A* − {*a*})) =
pmf-of-set A ⤜ (λ*a*. *fold-random-permutation f* (*f a x*) (*A* − {*a*}) ⤜ *g*)
by (*intro bind-pmf-cong*) *simp-all*
with *remove* **show** *?case* **by** (*simp add*: *bind-return-pmf bind-assoc-pmf*)
qed (*simp-all add*: *bind-return-pmf*)

We can now derive the following nice monadic representations of the combined fold-and-bind:

lemma *fold-bind-random-permutation-foldl*:
assumes *finite A*
shows *fold-bind-random-permutation f g x A* =
do {*xs* ← *pmf-of-set* (*permutations-of-set A*); *g* (*foldl* (λ*x y*. *f y x*) *x xs*)}

using *assms* **by** (*simp add*: *fold-bind-random-permutation-altdef bind-assoc-pmf fold-random-permutation-foldl bind-return-pmf map-pmf-def*)

lemma *fold-bind-random-permutation-foldr*:
assumes *finite A*
shows *fold-bind-random-permutation f g x A* =
do {*xs* ← *pmf-of-set* (*permutations-of-set A*); *g* (*foldr f xs x*)}
using *assms* **by** (*simp add*: *fold-bind-random-permutation-altdef bind-assoc-pmf fold-random-permutation-foldr bind-return-pmf map-pmf-def*)

lemma *fold-bind-random-permutation-fold*:
assumes *finite A*
shows *fold-bind-random-permutation f g x A* =
do {*xs* ← *pmf-of-set* (*permutations-of-set A*); *g* (*fold f xs x*)}
using *assms* **by** (*simp add*: *fold-bind-random-permutation-altdef bind-assoc-pmf fold-random-permutation-fold bind-return-pmf map-pmf-def*)

The following useful lemma allows us to swap partitioning a set w. r. t. a predicate and drawing a random permutation of that set.

lemma *partition-random-permutations*:
assumes *finite A*
shows *map-pmf* (*partition P*) (*pmf-of-set* (*permutations-of-set A*)) =
pair-pmf (*pmf-of-set* (*permutations-of-set* {*x*∈*A*. *P x*}))
(*pmf-of-set* (*permutations-of-set* {*x*∈*A*. ¬*P x*})) (**is** *?lhs* = *?rhs*)
proof (*rule pmf-eqI*, *clarify*, *goal-cases*)
case (*1 xs ys*)
show *?case*
proof (*cases xs* ∈ *permutations-of-set* {*x*∈*A*. *P x*} ∧ *ys* ∈ *permutations-of-set* {*x*∈*A*. ¬*P x*})
case *True*
let *?n1* = *card* {*x*∈*A*. *P x*} **and** *?n2* = *card* {*x*∈*A*. ¬*P x*}
have *card-eq*: *card A* = *?n1* + *?n2*
proof −
have *?n1* + *?n2* = *card* ({*x*∈*A*. *P x*} ∪ {*x*∈*A*. ¬*P x*})
using *assms* **by** (*intro card-Un-disjoint* [*symmetric*]) *auto*
also have {*x*∈*A*. *P x*} ∪ {*x*∈*A*. ¬*P x*} = *A* **by** *blast*
finally show *?thesis* **..**
qed

from *True* **have** *lengths* [*simp*]: *length xs* = *?n1 length ys* = *?n2*
by (*auto intro*!: *length-finite-permutations-of-set*)
have *pmf ?lhs* (*xs*, *ys*) =
real (*card* (*permutations-of-set A* ∩ *partition P* −‘ {(*xs*, *ys*)})) / *fact* (*card A*)
using *assms* **by** (*auto simp*: *pmf-map measure-pmf-of-set*)
also have *partition P* −‘ {(*xs*, *ys*)} = *shuffles xs ys*
using *True* **by** (*intro inv-image-partition*) (*auto simp*: *permutations-of-set-def*)
also have *permutations-of-set A* ∩ *shuffles xs ys* = *shuffles xs ys*
using *True distinct-disjoint-shuffles*[*of xs ys*]

by (*auto simp*: *permutations-of-set-def dest*: *set-shuffles*)
also have *card* (*shuffles xs ys*) = *length xs* + *length ys choose length xs*
using *True* **by** (*intro card-disjoint-shuffles*) (*auto simp*: *permutations-of-set-def*)
also have *length xs* + *length ys* = *card A* **by** (*simp add*: *card-eq*)
also have *real* (*card A choose length xs*) = *fact* (*card A*) / (*fact ?n1* ∗ *fact* (*card A* − *?n1*))
by (*subst binomial-fact*) (*auto intro*!: *card-mono assms*)
also have ... / *fact* (*card A*) = *1* / (*fact ?n1* ∗ *fact ?n2*)
by (*simp add*: *field-split-simps card-eq*)
also have ... = *pmf ?rhs* (*xs*, *ys*) **using** *True assms* **by** (*simp add*: *pmf-pair*)
finally show *?thesis* **.**
next
case *False*
hence ∗: *xs* ∉ *permutations-of-set* {*x*∈*A*. *P x*} ∨ *ys* ∉ *permutations-of-set* {*x*∈*A*. ¬*P x*} **by** *blast*
hence *eq*: *permutations-of-set A* ∩ (*partition P* −‘ {(*xs*, *ys*)}) = {}
by (*auto simp*: *o-def permutations-of-set-def*)
from ∗ **show** *?thesis*
by (*elim disjE*) (*insert assms eq*, *simp-all add*: *pmf-pair pmf-map measure-pmf-of-set*)
qed
qed

end

25 Discrete subprobability distribution

theory *SPMF* **imports**
Probability-Mass-Function
HOL−Library.Complete-Partial-Order2
HOL−Library.Rewrite
begin

25.1 Auxiliary material

lemma *cSUP-singleton* [*simp*]: (*SUP x*∈{*x*}. *f x* :: - :: *conditionally-complete-lattice*) = *f x*
by (*metis cSup-singleton image-empty image-insert*)

25.1.1 More about extended reals

lemma [*simp*]:
shows *ennreal-max-0*: *ennreal* (*max 0 x*) = *ennreal x*
and *ennreal-max-0′*: *ennreal* (*max x 0*) = *ennreal x*
by(*simp-all add*: *max-def ennreal-eq-0-iff*)

lemma *e2ennreal-0* [*simp*]: *e2ennreal 0* = *0*
by(*simp add*: *zero-ennreal-def*)

lemma *enn2real-bot* [*simp*]: *enn2real* $\bot = 0$
by(*simp add*: *bot-ennreal-def*)

lemma *continuous-at-ennreal*[*continuous-intros*]: *continuous* $F\ f \Longrightarrow$ *continuous* F $(\lambda x.\ ennreal\ (f\ x))$
unfolding *continuous-def* **by** *auto*

lemma *ennreal-Sup*:
assumes $*$: $(SUP\ a{\in}A.\ ennreal\ a) \neq \top$
and $A \neq \{\}$
shows *ennreal* $(Sup\ A) = (SUP\ a{\in}A.\ ennreal\ a)$
proof (*rule continuous-at-Sup-mono*)
obtain r **where** r: $ennreal\ r = (SUP\ a{\in}A.\ ennreal\ a)$ $r \geq 0$
using $*$ **by**(*cases* $(SUP\ a{\in}A.\ ennreal\ a)$) *simp-all*
then show *bdd-above A*
by(*auto intro*!: *SUP-upper bdd-aboveI*[*of - r*] *simp add*: *ennreal-le-iff*[*symmetric*])
qed (*auto simp*: *mono-def continuous-at-imp-continuous-at-within continuous-at-ennreal ennreal-leI assms*)

lemma *ennreal-SUP*:
$[\![(SUP\ a{\in}A.\ ennreal\ (f\ a)) \neq \top;\ A \neq \{\}]\!] \Longrightarrow ennreal\ (SUP\ a{\in}A.\ f\ a) = (SUP\ a{\in}A.\ ennreal\ (f\ a))$
using *ennreal-Sup*[*of f ‘ A*] **by** (*auto simp add*: *image-comp*)

lemma *ennreal-lt-0*: $x < 0 \Longrightarrow ennreal\ x = 0$
by(*simp add*: *ennreal-eq-0-iff*)

25.1.2 More about *'a option*

lemma *None-in-map-option-image* [*simp*]: *None* $\in$ *map-option f ‘ A* $\longleftrightarrow$ *None* $\in$ *A*
by *auto*

lemma *Some-in-map-option-image* [*simp*]: *Some x* $\in$ *map-option f ‘ A* $\longleftrightarrow$ $(\exists\, y.\ x = f\ y \wedge Some\ y \in A)$
by(*auto intro*: *rev-image-eqI dest*: *sym*)

lemma *case-option-collapse*: *case-option x* $(\lambda\text{-}.\ x) = (\lambda\text{-}.\ x)$
by(*simp add*: *fun-eq-iff split*: *option.split*)

lemma *case-option-id*: *case-option None Some* $= id$
by(*rule ext*)(*simp split*: *option.split*)

inductive *ord-option* :: $('a \Rightarrow 'b \Rightarrow bool) \Rightarrow 'a\ option \Rightarrow 'b\ option \Rightarrow bool$
for *ord* :: $'a \Rightarrow 'b \Rightarrow bool$
where
None: *ord-option ord None x*
| *Some*: *ord x y* $\Longrightarrow$ *ord-option ord* $(Some\ x)\ (Some\ y)$

inductive-simps *ord-option-simps* [*simp*]:
ord-option ord None x
ord-option ord x None
ord-option ord (*Some x*) (*Some y*)
ord-option ord (*Some x*) *None*

inductive-simps *ord-option-eq-simps* [*simp*]:
ord-option (=) *None y*
ord-option (=) (*Some x*) *y*

lemma *ord-option-reflI*: (⋀*y*. *y* ∈ *set-option x* ⟹ *ord y y*) ⟹ *ord-option ord x x*
by(*cases x*) *simp-all*

lemma *reflp-ord-option*: *reflp ord* ⟹ *reflp* (*ord-option ord*)
by(*simp add*: *reflp-def ord-option-reflI*)

lemma *ord-option-trans*:
⟦ *ord-option ord x y*; *ord-option ord y z*;
⋀*a b c*. ⟦ *a* ∈ *set-option x*; *b* ∈ *set-option y*; *c* ∈ *set-option z*; *ord a b*; *ord b c* ⟧ ⟹ *ord a c* ⟧
⟹ *ord-option ord x z*
by(*auto elim*!: *ord-option.cases*)

lemma *transp-ord-option*: *transp ord* ⟹ *transp* (*ord-option ord*)
unfolding *transp-def* **by**(*blast intro*: *ord-option-trans*)

lemma *antisymp-ord-option*: *antisymp ord* ⟹ *antisymp* (*ord-option ord*)
by(*auto intro*!: *antisympI elim*!: *ord-option.cases dest*: *antisympD*)

lemma *ord-option-chainD*:
Complete-Partial-Order.chain (*ord-option ord*) *Y*
⟹ *Complete-Partial-Order.chain ord* {*x*. *Some x* ∈ *Y*}
by(*rule chainI*)(*auto dest*: *chainD*)

definition *lub-option* :: ('*a set* ⇒ '*b*) ⇒ '*a option set* ⇒ '*b option*
where *lub-option lub Y* = (*if Y* ⊆ {*None*} *then None else Some* (*lub* {*x*. *Some x* ∈ *Y*}))

lemma *map-lub-option*: *map-option f* (*lub-option lub Y*) = *lub-option* (*f* ∘ *lub*) *Y*
by(*simp add*: *lub-option-def*)

lemma *lub-option-upper*:
assumes *Complete-Partial-Order.chain* (*ord-option ord*) *Y x* ∈ *Y*
and *lub-upper*: ⋀*Y x*. ⟦ *Complete-Partial-Order.chain ord Y*; *x* ∈ *Y* ⟧ ⟹ *ord x* (*lub Y*)
shows *ord-option ord x* (*lub-option lub Y*)
using *assms*(*1*−*2*)
by(*cases x*)(*auto simp add*: *lub-option-def intro*: *lub-upper*[*OF ord-option-chainD*])

lemma *lub-option-least*:
assumes *Y*: *Complete-Partial-Order.chain* (*ord-option ord*) *Y*
and *upper*: ⋀*x*. *x* ∈ *Y* ⟹ *ord-option ord x y*
assumes *lub-least*: ⋀*Y y*. ⟦ *Complete-Partial-Order.chain ord Y*; ⋀*x*. *x* ∈ *Y* ⟹ *ord x y* ⟧ ⟹ *ord* (*lub Y*) *y*
shows *ord-option ord* (*lub-option lub Y*) *y*
using *Y*
by(*cases y*)(*auto 4 3 simp add*: *lub-option-def intro*: *lub-least*[*OF ord-option-chainD*] *dest*: *upper*)

lemma *lub-map-option*: *lub-option lub* (*map-option f* ‘ *Y*) = *lub-option* (*lub* ∘ (‘) *f*) *Y*
apply(*auto simp add*: *lub-option-def*)
apply(*erule notE*)
apply(*rule arg-cong*[**where** *f*=*lub*])
apply(*auto intro*: *rev-image-eqI dest*: *sym*)
done

lemma *ord-option-mono*: ⟦ *ord-option A x y*; ⋀*x y*. *A x y* ⟹ *B x y* ⟧ ⟹ *ord-option B x y*
by(*auto elim*: *ord-option.cases*)

lemma *ord-option-mono′* [*mono*]:
(⋀*x y*. *A x y* ⟶ *B x y*) ⟹ *ord-option A x y* ⟶ *ord-option B x y*
by(*blast intro*: *ord-option-mono*)

lemma *ord-option-compp*: *ord-option* (*A OO B*) = *ord-option A OO ord-option B*
by(*auto simp add*: *fun-eq-iff elim*!: *ord-option.cases intro*: *ord-option.intros*)

lemma *ord-option-inf*: *inf* (*ord-option A*) (*ord-option B*) = *ord-option* (*inf A B*) (**is** *?lhs* = *?rhs*)
proof(*rule antisym*)
show *?lhs* ≤ *?rhs* **by**(*auto elim*!: *ord-option.cases*)
qed(*auto elim*: *ord-option-mono*)

lemma *ord-option-map2*: *ord-option ord x* (*map-option f y*) = *ord-option* (λ*x y*. *ord x* (*f y*)) *x y*
by(*auto elim*: *ord-option.cases*)

lemma *ord-option-map1*: *ord-option ord* (*map-option f x*) *y* = *ord-option* (λ*x y*. *ord* (*f x*) *y*) *x y*
by(*auto elim*: *ord-option.cases*)

lemma *option-ord-Some1-iff*: *option-ord* (*Some x*) *y* ⟷ *y* = *Some x*
by(*auto simp add*: *flat-ord-def*)

25.1.3 A relator for sets that treats sets like predicates

context includes *lifting-syntax*
begin

definition *rel-pred* :: $('a \Rightarrow 'b \Rightarrow bool) \Rightarrow 'a\ set \Rightarrow 'b\ set \Rightarrow bool$
where $rel\text{-}pred\ R\ A\ B = (R ===> (=))\ (\lambda x.\ x \in A)\ (\lambda y.\ y \in B)$

lemma *rel-predI*: $(R ===> (=))\ (\lambda x.\ x \in A)\ (\lambda y.\ y \in B) \Longrightarrow rel\text{-}pred\ R\ A\ B$
by(*simp add*: *rel-pred-def*)

lemma *rel-predD*: $[\![\ rel\text{-}pred\ R\ A\ B;\ R\ x\ y\ ]\!] \Longrightarrow x \in A \longleftrightarrow y \in B$
by(*simp add*: *rel-pred-def rel-fun-def*)

lemma *Collect-parametric*: $((A ===> (=)) ===> rel\text{-}pred\ A)\ Collect\ Collect$
— Declare this rule as *transfer-rule* only locally because it blows up the search space for *transfer* (in combination with *Collect-transfer*)
by(*simp add*: *rel-funI rel-predI*)

end

25.1.4 Monotonicity rules

lemma *monotone-gfp-eadd1*: $monotone\ (\geq)\ (\geq)\ (\lambda x.\ x + y :: enat)$
by(*auto intro*!: *monotoneI*)

lemma *monotone-gfp-eadd2*: $monotone\ (\geq)\ (\geq)\ (\lambda y.\ x + y :: enat)$
by(*auto intro*!: *monotoneI*)

lemma *mono2mono-gfp-eadd*[*THEN gfp.mono2mono2*, *cont-intro*, *simp*]:
shows *monotone-eadd*: $monotone\ (rel\text{-}prod\ (\geq)\ (\geq))\ (\geq)\ (\lambda(x, y).\ x + y :: enat)$
by(*simp add*: *monotone-gfp-eadd1 monotone-gfp-eadd2*)

lemma *eadd-gfp-partial-function-mono* [*partial-function-mono*]:
$[\![\ monotone\ (fun\text{-}ord\ (\geq))\ (\geq)\ f;\ monotone\ (fun\text{-}ord\ (\geq))\ (\geq)\ g\ ]\!]$
$\Longrightarrow monotone\ (fun\text{-}ord\ (\geq))\ (\geq)\ (\lambda x.\ f\ x + g\ x :: enat)$
by(*rule mono2mono-gfp-eadd*)

lemma *mono2mono-ereal*[*THEN lfp.mono2mono*]:
shows *monotone-ereal*: $monotone\ (\leq)\ (\leq)\ ereal$
by(*rule monotoneI*) *simp*

lemma *mono2mono-ennreal*[*THEN lfp.mono2mono*]:
shows *monotone-ennreal*: $monotone\ (\leq)\ (\leq)\ ennreal$
by(*rule monotoneI*)(*simp add*: *ennreal-leI*)

25.1.5 Bijections

lemma *bi-unique-rel-set-bij-betw*:
assumes *unique*: *bi-unique R*

and *rel*: *rel-set R A B*
shows $\exists f.$ *bij-betw f A B* $\wedge$ $(\forall x{\in}A.\ R\ x\ (f\ x))$
proof −
from *assms* **obtain** *f* **where** *f*: $\bigwedge x.\ x \in A \Longrightarrow R\ x\ (f\ x)$ **and** *B*: $\bigwedge x.\ x \in A \Longrightarrow f\ x \in B$
apply(*atomize-elim*)
apply(*fold all-conj-distrib*)
apply(*subst choice-iff*[*symmetric*])
apply(*auto dest*: *rel-setD1*)
done
have *inj-on f A* **by**(*rule inj-onI*)(*auto dest*!: *f dest*: *bi-uniqueDl*[*OF unique*])
moreover have *f ' A* = *B* **using** *rel*
by(*auto 4 3 intro*: *B dest*: *rel-setD2 f bi-uniqueDr*[*OF unique*])
ultimately have *bij-betw f A B* **unfolding** *bij-betw-def* **..**
thus *?thesis* **using** *f* **by** *blast*
qed

lemma *bij-betw-rel-setD*: *bij-betw f A B* $\Longrightarrow$ *rel-set* $(\lambda x\ y.\ y = f\ x)$ *A B*
by(*rule rel-setI*)(*auto dest*: *bij-betwE bij-betw-imp-surj-on*[*symmetric*])

25.2 Subprobability mass function

type-synonym *'a spmf* = *'a option pmf*
translations (*type*) *'a spmf* ↽ (*type*) *'a option pmf*

definition *measure-spmf* :: *'a spmf* ⇒ *'a measure*
where *measure-spmf p* = *distr* (*restrict-space* (*measure-pmf p*) (*range Some*)) (*count-space UNIV*) *the*

abbreviation *spmf* :: *'a spmf* ⇒ *'a* ⇒ *real*
where *spmf p x* ≡ *pmf p* (*Some x*)

lemma *space-measure-spmf*: *space* (*measure-spmf p*) = *UNIV*
by(*simp add*: *measure-spmf-def*)

lemma *sets-measure-spmf* [*simp*, *measurable-cong*]: *sets* (*measure-spmf p*) = *sets* (*count-space UNIV*)
by(*simp add*: *measure-spmf-def*)

lemma *measure-spmf-not-bot* [*simp*]: *measure-spmf p* ≠ ⊥
proof
assume *measure-spmf p* = ⊥
hence *space* (*measure-spmf p*) = *space* ⊥ **by** *simp*
thus *False* **by**(*simp add*: *space-measure-spmf*)
qed

lemma *measurable-the-measure-pmf-Some* [*measurable*, *simp*]:
the ∈ *measurable* (*restrict-space* (*measure-pmf p*) (*range Some*)) (*count-space UNIV*)

by(*auto simp add*: *measurable-def sets-restrict-space space-restrict-space integral-restrict-space*)

lemma *measurable-spmf-measure1*[*simp*]: *measurable* (*measure-spmf M*) *N* = *UNIV* → *space N*
by(*auto simp*: *measurable-def space-measure-spmf*)

lemma *measurable-spmf-measure2*[*simp*]: *measurable N* (*measure-spmf M*) = *measurable N* (*count-space UNIV*)
by(*intro measurable-cong-sets*) *simp-all*

lemma *subprob-space-measure-spmf* [*simp*, *intro*!]: *subprob-space* (*measure-spmf p*)
proof
show *emeasure* (*measure-spmf p*) (*space* (*measure-spmf p*)) ≤ *1*
by(*simp add*: *measure-spmf-def emeasure-distr emeasure-restrict-space space-restrict-space measure-pmf.measure-le-1*)
qed(*simp add*: *space-measure-spmf*)

interpretation *measure-spmf*: *subprob-space measure-spmf p* **for** *p*
by(*rule subprob-space-measure-spmf*)

lemma *finite-measure-spmf* [*simp*]: *finite-measure* (*measure-spmf p*)
by *unfold-locales*

lemma *spmf-conv-measure-spmf*: *spmf p x* = *measure* (*measure-spmf p*) {*x*}
by(*auto simp add*: *measure-spmf-def measure-distr measure-restrict-space pmf.rep-eq space-restrict-space intro*: *arg-cong2*[**where** *f*=*measure*])

lemma *emeasure-measure-spmf-conv-measure-pmf*:
emeasure (*measure-spmf p*) *A* = *emeasure* (*measure-pmf p*) (*Some* ‘ *A*)
by(*auto simp add*: *measure-spmf-def emeasure-distr emeasure-restrict-space space-restrict-space intro*: *arg-cong2*[**where** *f*=*emeasure*])

lemma *measure-measure-spmf-conv-measure-pmf*:
measure (*measure-spmf p*) *A* = *measure* (*measure-pmf p*) (*Some* ‘ *A*)
using *emeasure-measure-spmf-conv-measure-pmf*[*of p A*]
by(*simp add*: *measure-spmf.emeasure-eq-measure measure-pmf.emeasure-eq-measure*)

lemma *emeasure-spmf-map-pmf-Some* [*simp*]:
emeasure (*measure-spmf* (*map-pmf Some p*)) *A* = *emeasure* (*measure-pmf p*) *A*
by(*auto simp add*: *measure-spmf-def emeasure-distr emeasure-restrict-space space-restrict-space intro*: *arg-cong2*[**where** *f*=*emeasure*])

lemma *measure-spmf-map-pmf-Some* [*simp*]:
measure (*measure-spmf* (*map-pmf Some p*)) *A* = *measure* (*measure-pmf p*) *A*
using *emeasure-spmf-map-pmf-Some*[*of p A*] **by**(*simp add*: *measure-spmf.emeasure-eq-measure measure-pmf.emeasure-eq-measure*)

lemma *nn-integral-measure-spmf*: (∫⁺ *x*. *f x* ∂*measure-spmf p*) = ∫⁺ *x*. *ennreal* (*spmf p x*) ∗ *f x* ∂*count-space UNIV*

(**is** *?lhs* = *?rhs*)
proof −
have *?lhs* = $\int^+ x.$ *pmf p x* ∗ *f* (*the x*) ∂*count-space* (*range Some*)
by(*simp add*: *measure-spmf-def nn-integral-distr nn-integral-restrict-space nn-integral-measure-pmf nn-integral-count-space-indicator ac-simps times-ereal.simps*(*1*)[*symmetric*] *del*: *times-ereal.simps*(*1*))
also have ... = $\int^+ x.$ *ennreal* (*spmf p* (*the x*)) ∗ *f* (*the x*) ∂*count-space* (*range Some*)
by(*rule nn-integral-cong*) *auto*
also have ... = $\int^+ x.$ *spmf p* (*the* (*Some x*)) ∗ *f* (*the* (*Some x*)) ∂*count-space UNIV*
by(*rule nn-integral-bij-count-space*[*symmetric*])(*simp add*: *bij-betw-def*)
also have ... = *?rhs* **by** *simp*
finally show *?thesis* **.**
qed

lemma *integral-measure-spmf*:
assumes *integrable* (*measure-spmf p*) *f*
shows ($\int$ *x. f x* ∂*measure-spmf p*) = $\int$ *x. spmf p x* ∗ *f x* ∂*count-space UNIV*
proof −
have *integrable* (*count-space UNIV*) (λ*x. spmf p x* ∗ *f x*)
using *assms* **by**(*simp add*: *integrable-iff-bounded nn-integral-measure-spmf abs-mult ennreal-mult''*)
then show *?thesis* **using** *assms*
by(*simp add*: *real-lebesgue-integral-def nn-integral-measure-spmf ennreal-mult'*[*symmetric*])
qed

lemma *emeasure-spmf-single*: *emeasure* (*measure-spmf p*) {*x*} = *spmf p x*
by(*simp add*: *measure-spmf.emeasure-eq-measure spmf-conv-measure-spmf*)

lemma *measurable-measure-spmf*[*measurable*]:
(λ*x. measure-spmf* (*M x*)) ∈ *measurable* (*count-space UNIV*) (*subprob-algebra* (*count-space UNIV*))
by (*auto simp*: *space-subprob-algebra*)

lemma *nn-integral-measure-spmf-conv-measure-pmf*:
assumes [*measurable*]: *f* ∈ *borel-measurable* (*count-space UNIV*)
shows *nn-integral* (*measure-spmf p*) *f* = *nn-integral* (*restrict-space* (*measure-pmf p*) (*range Some*)) (*f* ∘ *the*)
by(*simp add*: *measure-spmf-def nn-integral-distr o-def*)

lemma *measure-spmf-in-space-subprob-algebra* [*simp*]:
measure-spmf p ∈ *space* (*subprob-algebra* (*count-space UNIV*))
by(*simp add*: *space-subprob-algebra*)

lemma *nn-integral-spmf-neq-top*: ($\int^+ x.$ *spmf p x* ∂*count-space UNIV*) ≠ ⊤
using *nn-integral-measure-spmf*[**where** *f*=λ*-. 1*, *of p*, *symmetric*] **by** *simp*

lemma *SUP-spmf-neq-top'*: (*SUP p*∈*Y. ennreal* (*spmf p x*)) ≠ ⊤
proof(*rule neq-top-trans*)

show (*SUP p∈Y. ennreal* (*spmf p x*)) ≤ *1* **by**(*rule SUP-least*)(*simp add*: *pmf-le-1*)
qed *simp*

lemma *SUP-spmf-neq-top*: (*SUP i. ennreal* (*spmf* (*Y i*) *x*)) ≠ ⊤
proof(*rule neq-top-trans*)
show (*SUP i. ennreal* (*spmf* (*Y i*) *x*)) ≤ *1* **by**(*rule SUP-least*)(*simp add*: *pmf-le-1*)
qed *simp*

lemma *SUP-emeasure-spmf-neq-top*: (*SUP p∈Y. emeasure* (*measure-spmf p*) *A*) ≠ ⊤
proof(*rule neq-top-trans*)
show (*SUP p∈Y. emeasure* (*measure-spmf p*) *A*) ≤ *1*
by(*rule SUP-least*)(*simp add*: *measure-spmf.subprob-emeasure-le-1*)
qed *simp*

25.3 Support

definition *set-spmf* :: *'a spmf* ⇒ *'a set*
where *set-spmf p* = *set-pmf p* ⤜ *set-option*

lemma *set-spmf-rep-eq*: *set-spmf p* = {*x. measure* (*measure-spmf p*) {*x*} ≠ *0*}
proof −
have ⋀*x* :: *'a. the* −‘ {*x*} ∩ *range Some* = {*Some x*} **by** *auto*
then show *?thesis*
by(*auto simp add*: *set-spmf-def set-pmf.rep-eq measure-spmf-def measure-distr measure-restrict-space space-restrict-space intro*: *rev-image-eqI*)
qed

lemma *in-set-spmf*: *x* ∈ *set-spmf p* ⟷ *Some x* ∈ *set-pmf p*
by(*simp add*: *set-spmf-def*)

lemma *AE-measure-spmf-iff* [*simp*]: (*AE x in measure-spmf p. P x*) ⟷ (∀ *x*∈*set-spmf p. P x*)
by(*auto 4 3 simp add*: *measure-spmf-def AE-distr-iff AE-restrict-space-iff AE-measure-pmf-iff set-spmf-def cong del*: *AE-cong*)

lemma *spmf-eq-0-set-spmf*: *spmf p x* = *0* ⟷ *x* ∉ *set-spmf p*
by(*auto simp add*: *pmf-eq-0-set-pmf set-spmf-def intro*: *rev-image-eqI*)

lemma *in-set-spmf-iff-spmf*: *x* ∈ *set-spmf p* ⟷ *spmf p x* ≠ *0*
by(*auto simp add*: *set-spmf-def set-pmf-iff intro*: *rev-image-eqI*)

lemma *set-spmf-return-pmf-None* [*simp*]: *set-spmf* (*return-pmf None*) = {}
by(*auto simp add*: *set-spmf-def*)

lemma *countable-set-spmf* [*simp*]: *countable* (*set-spmf p*)
by(*simp add*: *set-spmf-def bind-UNION*)

lemma *spmf-eqI*:
assumes $\bigwedge i.$ *spmf p i* = *spmf q i*
shows *p* = *q*
proof(*rule pmf-eqI*)
fix *i*
show *pmf p i* = *pmf q i*
proof(*cases i*)
case (*Some i'*)
thus *?thesis* **by**(*simp add*: *assms*)
next
case *None*
have *ennreal* (*pmf p i*) = *measure* (*measure-pmf p*) {*i*} **by**(*simp add*: *pmf-def*)
also have {*i*} = *space* (*measure-pmf p*) − *range Some*
by(*auto simp add*: *None intro*: *ccontr*)
also have *measure* (*measure-pmf p*) ... = *ennreal 1* − *measure* (*measure-pmf p*) (*range Some*)
by(*simp add*: *measure-pmf.prob-compl ennreal-minus*[*symmetric*] *del*: *space-measure-pmf*)
also have *range Some* = (⋃ *x*∈*set-spmf p*. {*Some x*}) ∪ *Some* ' (− *set-spmf p*)
by *auto*
also have *measure* (*measure-pmf p*) ... = *measure* (*measure-pmf p*) (⋃ *x*∈*set-spmf p*. {*Some x*})
by(*rule measure-pmf.measure-zero-union*)(*auto simp add*: *measure-pmf.prob-eq-0 AE-measure-pmf-iff in-set-spmf-iff-spmf set-pmf-iff*)
also have *ennreal* ... = ∫⁺ *x*. *measure* (*measure-pmf p*) {*Some x*} ∂*count-space* (*set-spmf p*)
unfolding *measure-pmf.emeasure-eq-measure*[*symmetric*]
by(*simp-all add*: *emeasure-UN-countable disjoint-family-on-def*)
also have ... = ∫⁺ *x*. *spmf p x* ∂*count-space* (*set-spmf p*) **by**(*simp add*: *pmf-def*)
also have ... = ∫⁺ *x*. *spmf q x* ∂*count-space* (*set-spmf p*) **by**(*simp add*: *assms*)
also have *set-spmf p* = *set-spmf q* **by**(*auto simp add*: *in-set-spmf-iff-spmf assms*)
also have (∫⁺ *x*. *spmf q x* ∂*count-space* (*set-spmf q*)) = ∫⁺ *x*. *measure* (*measure-pmf q*) {*Some x*} ∂*count-space* (*set-spmf q*)
by(*simp add*: *pmf-def*)
also have ... = *measure* (*measure-pmf q*) (⋃ *x*∈*set-spmf q*. {*Some x*})
unfolding *measure-pmf.emeasure-eq-measure*[*symmetric*]
by(*simp-all add*: *emeasure-UN-countable disjoint-family-on-def*)
also have ... = *measure* (*measure-pmf q*) ((⋃ *x*∈*set-spmf q*. {*Some x*}) ∪ *Some* ' (− *set-spmf q*))
by(*rule ennreal-cong measure-pmf.measure-zero-union*[*symmetric*])+(*auto simp add*: *measure-pmf.prob-eq-0 AE-measure-pmf-iff in-set-spmf-iff-spmf set-pmf-iff*)
also have ((⋃ *x*∈*set-spmf q*. {*Some x*}) ∪ *Some* ' (− *set-spmf q*)) = *range Some* **by** *auto*
also have *ennreal 1* − *measure* (*measure-pmf q*) ... = *measure* (*measure-pmf q*) (*space* (*measure-pmf q*) − *range Some*)
by(*simp add*: *one-ereal-def measure-pmf.prob-compl ennreal-minus*[*symmetric*] *del*: *space-measure-pmf*)
also have *space* (*measure-pmf q*) − *range Some* = {*i*}

by(*auto simp add*: *None intro*: *ccontr*)
also have *measure* (*measure-pmf q*) ... = *pmf q i* **by**(*simp add*: *pmf-def*)
finally show *?thesis* **by** *simp*
qed
qed

lemma *integral-measure-spmf-restrict*:
fixes *f* :: *'a* ⇒ *'b* :: {*banach*, *second-countable-topology*} **shows**
(∫ *x*. *f x* ∂*measure-spmf M*) = (∫ *x*. *f x* ∂*restrict-space* (*measure-spmf M*) (*set-spmf M*))
by(*auto intro*!: *integral-cong-AE simp add*: *integral-restrict-space*)

lemma *nn-integral-measure-spmf'*:
(∫⁺ *x*. *f x* ∂*measure-spmf p*) = ∫⁺ *x*. *ennreal* (*spmf p x*) ∗ *f x* ∂*count-space* (*set-spmf p*)
by(*auto simp add*: *nn-integral-measure-spmf nn-integral-count-space-indicator in-set-spmf-iff-spmf intro*!: *nn-integral-cong split*: *split-indicator*)

25.4 Functorial structure

abbreviation *map-spmf* :: (*'a* ⇒ *'b*) ⇒ *'a spmf* ⇒ *'b spmf*
where *map-spmf f* ≡ *map-pmf* (*map-option f*)

context begin
local-setup ‹*Local-Theory.map-background-naming* (*Name-Space.mandatory-path spmf*)›

lemma *map-comp*: *map-spmf f* (*map-spmf g p*) = *map-spmf* (*f* ∘ *g*) *p*
by(*simp add*: *pmf.map-comp o-def option.map-comp*)

lemma *map-id0*: *map-spmf id* = *id*
by(*simp add*: *pmf.map-id option.map-id0*)

lemma *map-id* [*simp*]: *map-spmf id p* = *p*
by(*simp add*: *map-id0*)

lemma *map-ident* [*simp*]: *map-spmf* (λ*x*. *x*) *p* = *p*
by(*simp add*: *id-def*[*symmetric*])

end

lemma *set-map-spmf* [*simp*]: *set-spmf* (*map-spmf f p*) = *f* ‘ *set-spmf p*
by(*simp add*: *set-spmf-def image-bind bind-image o-def Option.option.set-map*)

lemma *map-spmf-cong*:
⟦ *p* = *q*; ⋀*x*. *x* ∈ *set-spmf q* ⟹ *f x* = *g x* ⟧
⟹ *map-spmf f p* = *map-spmf g q*
by(*auto intro*: *pmf.map-cong option.map-cong simp add*: *in-set-spmf*)

lemma *map-spmf-cong-simp*:
⟦ *p* = *q*; ⋀*x*. *x* ∈ *set-spmf q* =*simp*=> *f x* = *g x* ⟧
⟹ *map-spmf f p* = *map-spmf g q*
unfolding *simp-implies-def* **by**(*rule map-spmf-cong*)

lemma *map-spmf-idI*: (⋀*x*. *x* ∈ *set-spmf p* ⟹ *f x* = *x*) ⟹ *map-spmf f p* = *p*
by(*rule map-pmf-idI map-option-idI*)+(*simp add*: *in-set-spmf*)

lemma *emeasure-map-spmf*:
emeasure (*measure-spmf* (*map-spmf f p*)) *A* = *emeasure* (*measure-spmf p*) (*f* −‘ *A*)
by(*auto simp add*: *measure-spmf-def emeasure-distr measurable-restrict-space1 space-restrict-space emeasure-restrict-space intro*: *arg-cong2*[**where** *f*=*emeasure*])

lemma *measure-map-spmf*: *measure* (*measure-spmf* (*map-spmf f p*)) *A* = *measure* (*measure-spmf p*) (*f* −‘ *A*)
using *emeasure-map-spmf*[*of f p A*] **by**(*simp add*: *measure-spmf.emeasure-eq-measure*)

lemma *measure-map-spmf-conv-distr*:
measure-spmf (*map-spmf f p*) = *distr* (*measure-spmf p*) (*count-space UNIV*) *f*
by(*rule measure-eqI*)(*simp-all add*: *emeasure-map-spmf emeasure-distr*)

lemma *spmf-map-pmf-Some* [*simp*]: *spmf* (*map-pmf Some p*) *i* = *pmf p i*
by(*simp add*: *pmf-map-inj'*)

lemma *spmf-map-inj*: ⟦ *inj-on f* (*set-spmf M*); *x* ∈ *set-spmf M* ⟧ ⟹ *spmf* (*map-spmf f M*) (*f x*) = *spmf M x*
by(*subst option.map*(*2*)[*symmetric*, **where** *f*=*f*])(*rule pmf-map-inj*, *auto simp add*: *in-set-spmf inj-on-def elim*!: *option.inj-map-strong*[*rotated*])

lemma *spmf-map-inj'*: *inj f* ⟹ *spmf* (*map-spmf f M*) (*f x*) = *spmf M x*
by(*subst option.map*(*2*)[*symmetric*, **where** *f*=*f*])(*rule pmf-map-inj'*[*OF option.inj-map*])

lemma *spmf-map-outside*: *x* ∉ *f* ‘ *set-spmf M* ⟹ *spmf* (*map-spmf f M*) *x* = *0*
unfolding *spmf-eq-0-set-spmf* **by** *simp*

lemma *ennreal-spmf-map*: *ennreal* (*spmf* (*map-spmf f p*) *x*) = *emeasure* (*measure-spmf p*) (*f* −‘ {*x*})
by(*auto simp add*: *ennreal-pmf-map measure-spmf-def emeasure-distr emeasure-restrict-space space-restrict-space intro*: *arg-cong2*[**where** *f*=*emeasure*])

lemma *spmf-map*: *spmf* (*map-spmf f p*) *x* = *measure* (*measure-spmf p*) (*f* −‘ {*x*})
using *ennreal-spmf-map*[*of f p x*] **by**(*simp add*: *measure-spmf.emeasure-eq-measure*)

lemma *ennreal-spmf-map-conv-nn-integral*:
ennreal (*spmf* (*map-spmf f p*) *x*) = *integral*^{N} (*measure-spmf p*) (*indicator* (*f* −‘ {*x*}))
by(*auto simp add*: *ennreal-pmf-map measure-spmf-def emeasure-distr space-restrict-space emeasure-restrict-space intro*: *arg-cong2*[**where** *f*=*emeasure*])

25.5 Monad operations

25.5.1 Return

abbreviation *return-spmf* :: *'a* ⇒ *'a spmf*
where *return-spmf x* ≡ *return-pmf* (*Some x*)

lemma *pmf-return-spmf*: *pmf* (*return-spmf x*) *y* = *indicator* {*y*} (*Some x*)
by(*fact pmf-return*)

lemma *measure-spmf-return-spmf*: *measure-spmf* (*return-spmf x*) = *Giry-Monad.return* (*count-space UNIV*) *x*
by(*rule measure-eqI*)(*simp-all add*: *measure-spmf-def emeasure-distr space-restrict-space emeasure-restrict-space indicator-def*)

lemma *measure-spmf-return-pmf-None* [*simp*]: *measure-spmf* (*return-pmf None*) = *null-measure* (*count-space UNIV*)
by(*rule measure-eqI*)(*auto simp add*: *measure-spmf-def emeasure-distr space-restrict-space emeasure-restrict-space indicator-eq-0-iff*)

lemma *set-return-spmf* [*simp*]: *set-spmf* (*return-spmf x*) = {*x*}
by(*auto simp add*: *set-spmf-def*)

25.5.2 Bind

definition *bind-spmf* :: *'a spmf* ⇒ (*'a* ⇒ *'b spmf*) ⇒ *'b spmf*
where *bind-spmf x f* = *bind-pmf x* (λ*a*. *case a of None* ⇒ *return-pmf None* | *Some a'* ⇒ *f a'*)

adhoc-overloading *Monad-Syntax.bind bind-spmf*

lemma *return-None-bind-spmf* [*simp*]: *return-pmf None* ⤜ (*f* :: *'a* ⇒ -) = *return-pmf None*
by(*simp add*: *bind-spmf-def bind-return-pmf*)

lemma *return-bind-spmf* [*simp*]: *return-spmf x* ⤜ *f* = *f x*
by(*simp add*: *bind-spmf-def bind-return-pmf*)

lemma *bind-return-spmf* [*simp*]: *x* ⤜ *return-spmf* = *x*
proof −
  have ⋀*a* :: *'a option*. (*case a of None* ⇒ *return-pmf None* | *Some a'* ⇒ *return-spmf a'*) = *return-pmf a*
    by(*simp split*: *option.split*)
  then show *?thesis*
    by(*simp add*: *bind-spmf-def bind-return-pmf'*)
qed

lemma *bind-spmf-assoc* [*simp*]:
  fixes *x* :: *'a spmf* **and** *f* :: *'a* ⇒ *'b spmf* **and** *g* :: *'b* ⇒ *'c spmf*
  shows (*x* ⤜ *f*) ⤜ *g* = *x* ⤜ (λ*y*. *f y* ⤜ *g*)

by(*auto simp add*: *bind-spmf-def bind-assoc-pmf fun-eq-iff bind-return-pmf split*: *option.split intro*: *arg-cong*[**where** *f*=*bind-pmf x*])

lemma *pmf-bind-spmf-None*: *pmf* (*p* ⤜ *f*) *None* = *pmf p None* + ∫ *x*. *pmf* (*f x*) *None* ∂*measure-spmf p*
(**is** *?lhs* = *?rhs*)
proof −
let *?f* = λ*x*. *pmf* (*case x of None* ⇒ *return-pmf None* | *Some x* ⇒ *f x*) *None*
have *?lhs* = ∫ *x*. *?f x* ∂*measure-pmf p*
by(*simp add*: *bind-spmf-def pmf-bind*)
also have ... = ∫ *x*. *?f None* ∗ *indicator* {*None*} *x* + *?f x* ∗ *indicator* (*range Some*) *x* ∂*measure-pmf p*
by(*rule Bochner-Integration.integral-cong*)(*auto simp add*: *indicator-def*)
also have ... = (∫ *x*. *?f None* ∗ *indicator* {*None*} *x* ∂*measure-pmf p*) + (∫ *x*. *?f x* ∗ *indicator* (*range Some*) *x* ∂*measure-pmf p*)
by(*rule Bochner-Integration.integral-add*)(*auto 4 3 intro*: *integrable-real-mult-indicator measure-pmf.integrable-const-bound*[**where** *B*=*1*] *simp add*: *AE-measure-pmf-iff pmf-le-1*)
also have ... = *pmf p None* + ∫ *x*. *indicator* (*range Some*) *x* ∗ *pmf* (*f* (*the x*)) *None* ∂*measure-pmf p*
by(*auto simp add*: *measure-measure-pmf-finite indicator-eq-0-iff intro*!: *Bochner-Integration.integral-cong*)
also have ... = *?rhs* **unfolding** *measure-spmf-def*
by(*subst integral-distr*)(*auto simp add*: *integral-restrict-space*)
finally show *?thesis* **.**
qed

lemma *spmf-bind*: *spmf* (*p* ⤜ *f*) *y* = ∫ *x*. *spmf* (*f x*) *y* ∂*measure-spmf p*
unfolding *measure-spmf-def*
by(*subst integral-distr*)(*auto simp add*: *bind-spmf-def pmf-bind integral-restrict-space indicator-eq-0-iff intro*!: *Bochner-Integration.integral-cong split*: *option.split*)

lemma *ennreal-spmf-bind*: *ennreal* (*spmf* (*p* ⤜ *f*) *x*) = ∫⁺ *y*. *spmf* (*f y*) *x* ∂*measure-spmf p*
by(*auto simp add*: *bind-spmf-def ennreal-pmf-bind nn-integral-measure-spmf-conv-measure-pmf nn-integral-restrict-space intro*: *nn-integral-cong split*: *split-indicator option.split*)

lemma *measure-spmf-bind-pmf*: *measure-spmf* (*p* ⤜ *f*) = *measure-pmf p* ⤜ *measure-spmf* ∘ *f*
(**is** *?lhs* = *?rhs*)
proof(*rule measure-eqI*)
show *sets ?lhs* = *sets ?rhs*
by(*simp add*: *sets-bind*[**where** *N*=*count-space UNIV*] *space-measure-spmf*)
next
fix *A* :: *'a set*
have *emeasure ?lhs A* = ∫⁺ *x*. *emeasure* (*measure-spmf* (*f x*)) *A* ∂*measure-pmf p*
by(*simp add*: *measure-spmf-def emeasure-distr space-restrict-space emeasure-restrict-space bind-spmf-def*)
also have ... = *emeasure ?rhs A*

by(*simp add*: *emeasure-bind*[**where** *N*=*count-space UNIV*] *space-measure-spmf space-subprob-algebra*)
finally show *emeasure ?lhs A* = *emeasure ?rhs A* **.**
qed

lemma *measure-spmf-bind*: *measure-spmf* (*p* ⤜ *f*) = *measure-spmf p* ⤜ *measure-spmf* ∘ *f*
(**is** *?lhs* = *?rhs*)
proof(*rule measure-eqI*)
show *sets ?lhs* = *sets ?rhs*
by(*simp add*: *sets-bind*[**where** *N*=*count-space UNIV*] *space-measure-spmf*)
next
fix *A* :: *'a set*
let *?A* = *the* −‘ *A* ∩ *range Some*
have *emeasure ?lhs A* = ∫⁺ *x*. *emeasure* (*measure-pmf* (*case x of None* ⇒ *return-pmf None* | *Some x* ⇒ *f x*)) *?A* ∂*measure-pmf p*
by(*simp add*: *measure-spmf-def emeasure-distr space-restrict-space emeasure-restrict-space bind-spmf-def*)
also have ... = ∫⁺ *x*. *emeasure* (*measure-pmf* (*f* (*the x*))) *?A* ∗ *indicator* (*range Some*) *x* ∂*measure-pmf p*
by(*rule nn-integral-cong*)(*auto split*: *option.split simp add*: *indicator-def*)
also have ... = ∫⁺ *x*. *emeasure* (*measure-spmf* (*f x*)) *A* ∂*measure-spmf p*
by(*simp add*: *measure-spmf-def nn-integral-distr nn-integral-restrict-space emeasure-distr space-restrict-space emeasure-restrict-space*)
also have ... = *emeasure ?rhs A*
by(*simp add*: *emeasure-bind*[**where** *N*=*count-space UNIV*] *space-measure-spmf space-subprob-algebra*)
finally show *emeasure ?lhs A* = *emeasure ?rhs A* **.**
qed

lemma *map-spmf-bind-spmf*: *map-spmf f* (*bind-spmf p g*) = *bind-spmf p* (*map-spmf f* ∘ *g*)
by(*auto simp add*: *bind-spmf-def map-bind-pmf fun-eq-iff split*: *option.split intro*: *arg-cong2*[**where** *f*=*bind-pmf*])

lemma *bind-map-spmf*: *map-spmf f p* ⤜ *g* = *p* ⤜ *g* ∘ *f*
by(*simp add*: *bind-spmf-def bind-map-pmf o-def cong del*: *option.case-cong-weak*)

lemma *spmf-bind-leI*:
assumes ⋀*y*. *y* ∈ *set-spmf p* ⟹ *spmf* (*f y*) *x* ≤ *r*
and *0* ≤ *r*
shows *spmf* (*bind-spmf p f*) *x* ≤ *r*
proof −
have *ennreal* (*spmf* (*bind-spmf p f*) *x*) = ∫⁺ *y*. *spmf* (*f y*) *x* ∂*measure-spmf p*
by(*rule ennreal-spmf-bind*)
also have ... ≤ ∫⁺ *y*. *r* ∂*measure-spmf p* **by**(*rule nn-integral-mono-AE*)(*simp add*: *assms*)
also have ... ≤ *r* **using** *assms measure-spmf.emeasure-space-le-1*
by(*auto simp add*: *measure-spmf.emeasure-eq-measure intro*!: *mult-left-le*)

finally show *?thesis* **using** *assms*(*2*) **by**(*simp*)
qed

lemma *map-spmf-conv-bind-spmf*: *map-spmf f p* = (*p* ⤜ (λ*x*. *return-spmf* (*f x*)))
by(*simp add*: *map-pmf-def bind-spmf-def*)(*rule bind-pmf-cong*, *simp-all split*: *option.split*)

lemma *bind-spmf-cong*:
⟦ *p* = *q*; ⋀*x*. *x* ∈ *set-spmf q* ⟹ *f x* = *g x* ⟧
⟹ *bind-spmf p f* = *bind-spmf q g*
by(*auto simp add*: *bind-spmf-def in-set-spmf intro*: *bind-pmf-cong option.case-cong*)

lemma *bind-spmf-cong-simp*:
⟦ *p* = *q*; ⋀*x*. *x* ∈ *set-spmf q* =*simp*=> *f x* = *g x* ⟧
⟹ *bind-spmf p f* = *bind-spmf q g*
by(*simp add*: *simp-implies-def cong*: *bind-spmf-cong*)

lemma *set-bind-spmf*: *set-spmf* (*M* ⤜ *f*) = *set-spmf M* ⤜ (*set-spmf* ∘ *f*)
by(*auto simp add*: *set-spmf-def bind-spmf-def bind-UNION split*: *option.splits*)

lemma *bind-spmf-const-return-None* [*simp*]: *bind-spmf p* (λ-. *return-pmf None*) = *return-pmf None*
by(*simp add*: *bind-spmf-def case-option-collapse*)

lemma *bind-commute-spmf*:
bind-spmf p (λ*x*. *bind-spmf q* (*f x*)) = *bind-spmf q* (λ*y*. *bind-spmf p* (λ*x*. *f x y*))
(**is** *?lhs* = *?rhs*)
proof −
let *?f* = λ*x y*. *case x of None* ⇒ *return-pmf None* | *Some a* ⇒ (*case y of None* ⇒ *return-pmf None* | *Some b* ⇒ *f a b*)
have *?lhs* = *p* ⤜ (λ*x*. *q* ⤜ *?f x*)
unfolding *bind-spmf-def* **by**(*rule bind-pmf-cong*[*OF refl*])(*simp split*: *option.split*)
also have ... = *q* ⤜ (λ*y*. *p* ⤜ (λ*x*. *?f x y*)) **by**(*rule bind-commute-pmf*)
also have ... = *?rhs* **unfolding** *bind-spmf-def*
by(*rule bind-pmf-cong*[*OF refl*])(*auto split*: *option.split*, *metis bind-spmf-const-return-None bind-spmf-def*)
finally show *?thesis* **.**
qed

25.6 Relator

abbreviation *rel-spmf* :: ('*a* ⇒ '*b* ⇒ *bool*) ⇒ '*a spmf* ⇒ '*b spmf* ⇒ *bool*
where *rel-spmf R* ≡ *rel-pmf* (*rel-option R*)

lemma *rel-pmf-mono*:
⟦*rel-pmf A f g*; ⋀*x y*. *A x y* ⟹ *B x y* ⟧ ⟹ *rel-pmf B f g*
using *pmf.rel-mono*[*of A B*] **by**(*simp add*: *le-fun-def*)

lemma *rel-spmf-mono*:

⟦*rel-spmf A f g*; ⋀*x y*. *A x y* ⟹ *B x y* ⟧ ⟹ *rel-spmf B f g*
apply(*erule rel-pmf-mono*)
using *option.rel-mono*[*of A B*] **by**(*simp add*: *le-fun-def*)

lemma *rel-spmf-mono-strong*:
⟦ *rel-spmf A f g*; ⋀*x y*. ⟦ *A x y*; *x* ∈ *set-spmf f*; *y* ∈ *set-spmf g* ⟧ ⟹ *B x y* ⟧ ⟹ *rel-spmf B f g*
apply(*erule pmf.rel-mono-strong*)
apply(*erule option.rel-mono-strong*)
apply(*auto simp add*: *in-set-spmf*)
done

lemma *rel-spmf-reflI*: (⋀*x*. *x* ∈ *set-spmf p* ⟹ *P x x*) ⟹ *rel-spmf P p p*
by(*rule rel-pmf-reflI*)(*auto simp add*: *set-spmf-def intro*: *rel-option-reflI*)

lemma *rel-spmfI* [*intro?*]:
⟦ ⋀*x y*. (*x*, *y*) ∈ *set-spmf pq* ⟹ *P x y*; *map-spmf fst pq* = *p*; *map-spmf snd pq* = *q* ⟧
⟹ *rel-spmf P p q*
by(*rule rel-pmf.intros*[**where** *pq*=*map-pmf* (λ*x*. *case x of None* ⇒ (*None*, *None*) | *Some* (*a*, *b*) ⇒ (*Some a*, *Some b*)) *pq*])
(*auto simp add*: *pmf.map-comp o-def in-set-spmf split*: *option.splits intro*: *pmf.map-cong*)

lemma *rel-spmfE* [*elim?*, *consumes 1*, *case-names rel-spmf*]:
assumes *rel-spmf P p q*
obtains *pq* **where**
⋀*x y*. (*x*, *y*) ∈ *set-spmf pq* ⟹ *P x y*
p = *map-spmf fst pq*
q = *map-spmf snd pq*
using *assms*
proof(*cases rule*: *rel-pmf.cases*[*consumes 1*, *case-names rel-pmf*])
case (*rel-pmf pq*)
let *?pq* = *map-pmf* (λ(*a*, *b*). *case* (*a*, *b*) *of* (*Some x*, *Some y*) ⇒ *Some* (*x*, *y*) | - ⇒ *None*) *pq*
have ⋀*x y*. (*x*, *y*) ∈ *set-spmf ?pq* ⟹ *P x y*
by(*auto simp add*: *in-set-spmf split*: *option.split-asm dest*: *rel-pmf*(*1*))
moreover
have ⋀*x*. (*x*, *None*) ∈ *set-pmf pq* ⟹ *x* = *None* **by**(*auto dest*!: *rel-pmf*(*1*))
then have *p* = *map-spmf fst ?pq* **using** *rel-pmf*(*2*)
by(*auto simp add*: *pmf.map-comp split-beta intro*!: *pmf.map-cong split*: *option.split*)
moreover
have ⋀*y*. (*None*, *y*) ∈ *set-pmf pq* ⟹ *y* = *None* **by**(*auto dest*!: *rel-pmf*(*1*))
then have *q* = *map-spmf snd ?pq* **using** *rel-pmf*(*3*)
by(*auto simp add*: *pmf.map-comp split-beta intro*!: *pmf.map-cong split*: *option.split*)
ultimately show *thesis* **..**
qed

lemma *rel-spmf-simps*:
rel-spmf R p q $\longleftrightarrow$ ($\exists$ *pq*. ($\forall$ (*x*, *y*)$\in$*set-spmf pq*. *R x y*) $\wedge$ *map-spmf fst pq* = *p* $\wedge$ *map-spmf snd pq* = *q*)
by(*auto intro*: *rel-spmfI elim*!: *rel-spmfE*)

lemma *spmf-rel-map*:
shows *spmf-rel-map1*: $\bigwedge$*R f x*. *rel-spmf R* (*map-spmf f x*) = *rel-spmf* (λ*x*. *R* (*f x*)) *x*
and *spmf-rel-map2*: $\bigwedge$*R x g y*. *rel-spmf R x* (*map-spmf g y*) = *rel-spmf* (λ*x y*. *R x* (*g y*)) *x y*
by(*simp-all add*: *fun-eq-iff pmf.rel-map option.rel-map*[*abs-def*])

lemma *spmf-rel-conversep*: *rel-spmf* R^{-1-1} = (*rel-spmf R*)$^{-1-1}$
by(*simp add*: *option.rel-conversep pmf.rel-conversep*)

lemma *spmf-rel-eq*: *rel-spmf* (=) = (=)
by(*simp add*: *pmf.rel-eq option.rel-eq*)

context includes *lifting-syntax*
begin

lemma *bind-spmf-parametric* [*transfer-rule*]:
(*rel-spmf A* ===> (*A* ===> *rel-spmf B*) ===> *rel-spmf B*) *bind-spmf bind-spmf*
unfolding *bind-spmf-def*[*abs-def*] **by** *transfer-prover*

lemma *return-spmf-parametric*: (*A* ===> *rel-spmf A*) *return-spmf return-spmf*
by *transfer-prover*

lemma *map-spmf-parametric*: ((*A* ===> *B*) ===> *rel-spmf A* ===> *rel-spmf B*) *map-spmf map-spmf*
by *transfer-prover*

lemma *rel-spmf-parametric*:
((*A* ===> *B* ===> (=)) ===> *rel-spmf A* ===> *rel-spmf B* ===> (=)) *rel-spmf rel-spmf*
by *transfer-prover*

lemma *set-spmf-parametric* [*transfer-rule*]:
(*rel-spmf A* ===> *rel-set A*) *set-spmf set-spmf*
unfolding *set-spmf-def*[*abs-def*] **by** *transfer-prover*

lemma *return-spmf-None-parametric*:
(*rel-spmf A*) (*return-pmf None*) (*return-pmf None*)
by *simp*

end

lemma *rel-spmf-bindI*:
⟦ *rel-spmf R p q*; $\bigwedge$*x y*. *R x y* $\Longrightarrow$ *rel-spmf P* (*f x*) (*g y*) ⟧

$\Longrightarrow$ *rel-spmf P* (*p* $\gg\!=$ *f*) (*q* $\gg\!=$ *g*)
by(*fact bind-spmf-parametric*[*THEN rel-funD*, *THEN rel-funD*, *OF - rel-funI*])

lemma *rel-spmf-bind-reflI*:
($\bigwedge$*x*. *x* $\in$ *set-spmf p* $\Longrightarrow$ *rel-spmf P* (*f x*) (*g x*)) $\Longrightarrow$ *rel-spmf P* (*p* $\gg\!=$ *f*) (*p* $\gg\!=$ *g*)
by(*rule rel-spmf-bindI*[**where** *R*=λ*x y*. *x* = *y* $\wedge$ *x* $\in$ *set-spmf p*])(*auto intro*: *rel-spmf-reflI*)

lemma *rel-pmf-return-pmfI*: *P x y* $\Longrightarrow$ *rel-pmf P* (*return-pmf x*) (*return-pmf y*)
by(*rule rel-pmf.intros*[**where** *pq*=*return-pmf* (*x*, *y*)])(*simp-all*)

context includes *lifting-syntax*
begin

We do not yet have a relator for *'a measure*, so we combine *Sigma-Algebra.measure* and *measure-pmf*

lemma *measure-pmf-parametric*:
(*rel-pmf A* ===> *rel-pred A* ===> (=)) (λ*p*. *measure* (*measure-pmf p*)) (λ*q*. *measure* (*measure-pmf q*))
proof(*rule rel-funI*)+
fix *p q X Y*
assume *rel-pmf A p q* **and** *rel-pred A X Y*
from *this*(*1*) **obtain** *pq* **where** *A*: $\bigwedge$*x y*. (*x*, *y*) $\in$ *set-pmf pq* $\Longrightarrow$ *A x y*
and *p*: *p* = *map-pmf fst pq* **and** *q*: *q* = *map-pmf snd pq* **by** *cases auto*
show *measure p X* = *measure q Y* **unfolding** *p q measure-map-pmf*
by(*rule measure-pmf.finite-measure-eq-AE*)(*auto simp add*: *AE-measure-pmf-iff dest*!: *A rel-predD*[*OF* ‹*rel-pred* - - -›])
qed

lemma *measure-spmf-parametric*:
(*rel-spmf A* ===> *rel-pred A* ===> (=)) (λ*p*. *measure* (*measure-spmf p*)) (λ*q*. *measure* (*measure-spmf q*))
unfolding *measure-measure-spmf-conv-measure-pmf*[*abs-def*]
apply(*rule rel-funI*)+
apply(*erule measure-pmf-parametric*[*THEN rel-funD*, *THEN rel-funD*])
apply(*auto simp add*: *rel-pred-def rel-fun-def elim*: *option.rel-cases*)
done

end

25.7 From *'a pmf* to *'a spmf*

definition *spmf-of-pmf* :: *'a pmf* $\Rightarrow$ *'a spmf*
where *spmf-of-pmf* = *map-pmf Some*

lemma *set-spmf-spmf-of-pmf* [*simp*]: *set-spmf* (*spmf-of-pmf p*) = *set-pmf p*
by(*auto simp add*: *spmf-of-pmf-def set-spmf-def bind-image o-def*)

lemma *spmf-spmf-of-pmf* [*simp*]: *spmf* (*spmf-of-pmf p*) *x* = *pmf p x*
by(*simp add*: *spmf-of-pmf-def*)

lemma *pmf-spmf-of-pmf-None* [*simp*]: *pmf* (*spmf-of-pmf p*) *None* = *0*
using *ennreal-pmf-map*[*of Some p None*] **by**(*simp add*: *spmf-of-pmf-def*)

lemma *emeasure-spmf-of-pmf* [*simp*]: *emeasure* (*measure-spmf* (*spmf-of-pmf p*))
A = *emeasure* (*measure-pmf p*) *A*
by(*simp add*: *emeasure-measure-spmf-conv-measure-pmf spmf-of-pmf-def inj-vimage-image-eq*)

lemma *measure-spmf-spmf-of-pmf* [*simp*]: *measure-spmf* (*spmf-of-pmf p*) = *measure-pmf p*
by(*rule measure-eqI*) *simp-all*

lemma *map-spmf-of-pmf* [*simp*]: *map-spmf f* (*spmf-of-pmf p*) = *spmf-of-pmf* (*map-pmf f p*)
by(*simp add*: *spmf-of-pmf-def pmf.map-comp o-def*)

lemma *rel-spmf-spmf-of-pmf* [*simp*]: *rel-spmf R* (*spmf-of-pmf p*) (*spmf-of-pmf q*) = *rel-pmf R p q*
by(*simp add*: *spmf-of-pmf-def pmf.rel-map*)

lemma *spmf-of-pmf-return-pmf* [*simp*]: *spmf-of-pmf* (*return-pmf x*) = *return-spmf x*
by(*simp add*: *spmf-of-pmf-def*)

lemma *bind-spmf-of-pmf* [*simp*]: *bind-spmf* (*spmf-of-pmf p*) *f* = *bind-pmf p f*
by(*simp add*: *spmf-of-pmf-def bind-spmf-def bind-map-pmf*)

lemma *set-spmf-bind-pmf*: *set-spmf* (*bind-pmf p f*) = *Set.bind* (*set-pmf p*) (*set-spmf* ∘ *f*)
unfolding *bind-spmf-of-pmf*[*symmetric*] **by**(*subst set-bind-spmf*) *simp*

lemma *spmf-of-pmf-bind*: *spmf-of-pmf* (*bind-pmf p f*) = *bind-pmf p* (λ*x*. *spmf-of-pmf* (*f x*))
by(*simp add*: *spmf-of-pmf-def map-bind-pmf*)

lemma *bind-pmf-return-spmf*: *p* ⤜ (λ*x*. *return-spmf* (*f x*)) = *spmf-of-pmf* (*map-pmf f p*)
by(*simp add*: *map-pmf-def spmf-of-pmf-bind*)

25.8 Weight of a subprobability

abbreviation *weight-spmf* :: *'a spmf* ⇒ *real*
where *weight-spmf p* ≡ *measure* (*measure-spmf p*) (*space* (*measure-spmf p*))

lemma *weight-spmf-def*: *weight-spmf p* = *measure* (*measure-spmf p*) *UNIV*
by(*simp add*: *space-measure-spmf*)

lemma *weight-spmf-le-1*: *weight-spmf p* $\leq$ *1*
by(*simp add*: *measure-spmf.subprob-measure-le-1*)

lemma *weight-return-spmf* [*simp*]: *weight-spmf* (*return-spmf x*) = *1*
by(*simp add*: *measure-spmf-return-spmf measure-return*)

lemma *weight-return-pmf-None* [*simp*]: *weight-spmf* (*return-pmf None*) = *0*
by(*simp*)

lemma *weight-map-spmf* [*simp*]: *weight-spmf* (*map-spmf f p*) = *weight-spmf p*
by(*simp add*: *weight-spmf-def measure-map-spmf*)

lemma *weight-spmf-of-pmf* [*simp*]: *weight-spmf* (*spmf-of-pmf p*) = *1*
using *measure-pmf.prob-space*[*of p*] **by**(*simp add*: *spmf-of-pmf-def weight-spmf-def*)

lemma *weight-spmf-nonneg*: *weight-spmf p* $\geq$ *0*
by(*fact measure-nonneg*)

lemma (**in** *finite-measure*) *integrable-weight-spmf* [*simp*]:
(*λx. weight-spmf* (*f x*)) $\in$ *borel-measurable M* $\Longrightarrow$ *integrable M* (*λx. weight-spmf* (*f x*))
by(*rule integrable-const-bound*[**where** *B*=*1*])(*simp-all add*: *weight-spmf-nonneg weight-spmf-le-1*)

lemma *weight-spmf-eq-nn-integral-spmf*: *weight-spmf p* = $\int^+$ *x. spmf p x* ∂*count-space UNIV*
by(*simp add*: *measure-measure-spmf-conv-measure-pmf space-measure-spmf measure-pmf.emeasure-eq-measure*[*symmetric*] *nn-integral-pmf*[*symmetric*] *embed-measure-count-space*[*symmetric*] *inj-on-def nn-integral-embed-measure measurable-embed-measure1*)

lemma *weight-spmf-eq-nn-integral-support*:
weight-spmf p = $\int^+$ *x. spmf p x* ∂*count-space* (*set-spmf p*)
unfolding *weight-spmf-eq-nn-integral-spmf*
by(*auto simp add*: *nn-integral-count-space-indicator in-set-spmf-iff-spmf intro*!: *nn-integral-cong split*: *split-indicator*)

lemma *pmf-None-eq-weight-spmf*: *pmf p None* = *1* − *weight-spmf p*
proof −
have *weight-spmf p* = $\int^+$ *x. spmf p x* ∂*count-space UNIV* **by**(*rule weight-spmf-eq-nn-integral-spmf*)
also have ... = $\int^+$ *x. ennreal* (*pmf p x*) ∗ *indicator* (*range Some*) *x* ∂*count-space UNIV*
by(*simp add*: *nn-integral-count-space-indicator*[*symmetric*] *embed-measure-count-space*[*symmetric*] *nn-integral-embed-measure measurable-embed-measure1*)
also have ... + *pmf p None* = $\int^+$ *x. ennreal* (*pmf p x*) ∗ *indicator* (*range Some*) *x* + *ennreal* (*pmf p None*) ∗ *indicator* {*None*} *x* ∂*count-space UNIV*
by(*subst nn-integral-add*)(*simp-all add*: *max-def*)
also have ... = $\int^+$ *x. pmf p x* ∂*count-space UNIV*
by(*rule nn-integral-cong*)(*auto split*: *split-indicator*)
also have ... = *1* **by** (*simp add*: *nn-integral-pmf*)
finally show *?thesis* **by**(*simp add*: *ennreal-plus*[*symmetric*] *del*: *ennreal-plus*)

qed

lemma *weight-spmf-conv-pmf-None*: *weight-spmf p = 1 − pmf p None*
by(*simp add*: *pmf-None-eq-weight-spmf*)

lemma *weight-spmf-le-0*: *weight-spmf p ≤ 0 ⟷ weight-spmf p = 0*
by(*rule measure-le-0-iff*)

lemma *weight-spmf-lt-0*: *¬ weight-spmf p < 0*
by(*simp add*: *not-less weight-spmf-nonneg*)

lemma *spmf-le-weight*: *spmf p x ≤ weight-spmf p*
proof −
have *ennreal* (*spmf p x*) ≤ *weight-spmf p*
unfolding *weight-spmf-eq-nn-integral-spmf* **by**(*rule nn-integral-ge-point*) *simp*
then show *?thesis* **by** *simp*
qed

lemma *weight-spmf-eq-0*: *weight-spmf p = 0 ⟷ p = return-pmf None*
by(*auto intro*!: *pmf-eqI simp add*: *pmf-None-eq-weight-spmf split*: *split-indicator*)(*metis not-Some-eq pmf-le-0-iff spmf-le-weight*)

lemma *weight-bind-spmf*: *weight-spmf* (*x* ⤜ *f*) = *lebesgue-integral* (*measure-spmf x*) (*weight-spmf* ∘ *f*)
unfolding *weight-spmf-def*
by(*simp add*: *measure-spmf-bind o-def measure-spmf.measure-bind*[**where** *N*=*count-space UNIV*])

lemma *rel-spmf-weightD*: *rel-spmf A p q ⟹ weight-spmf p = weight-spmf q*
by(*erule rel-spmfE*) *simp*

lemma *rel-spmf-bij-betw*:
assumes *f*: *bij-betw f* (*set-spmf p*) (*set-spmf q*)
and *eq*: ⋀*x*. *x* ∈ *set-spmf p ⟹ spmf p x = spmf q* (*f x*)
shows *rel-spmf* (λ*x y*. *f x = y*) *p q*
proof −
let *?f* = *map-option f*

have *weq*: *ennreal* (*weight-spmf p*) = *ennreal* (*weight-spmf q*)
unfolding *weight-spmf-eq-nn-integral-support*
by(*subst nn-integral-bij-count-space*[*OF f*, *symmetric*])(*rule nn-integral-cong-AE*, *simp add*: *eq AE-count-space*)
then have *None* ∈ *set-pmf p ⟷ None* ∈ *set-pmf q*
by(*simp add*: *pmf-None-eq-weight-spmf set-pmf-iff*)
with *f* **have** *bij-betw* (*map-option f*) (*set-pmf p*) (*set-pmf q*)
apply(*auto simp add*: *bij-betw-def in-set-spmf inj-on-def intro*: *option.expand*)
apply(*rename-tac* [!] *x*)
apply(*case-tac* [!] *x*)
apply(*auto iff*: *in-set-spmf*)

done
then have *rel-pmf* $(\lambda x\ y.\ ?f\ x = y)\ p\ q$
by(*rule rel-pmf-bij-betw*)(*case-tac x*, *simp-all add*: *weq*[*simplified*] *eq in-set-spmf pmf-None-eq-weight-spmf*)
thus *?thesis* **by**(*rule pmf.rel-mono-strong*)(*auto intro*!: *rel-optionI simp add*: *Option.is-none-def*)
qed

25.9 From density to spmfs

context fixes $f :: 'a \Rightarrow real$ **begin**

definition *embed-spmf* :: *'a spmf*
where *embed-spmf* = *embed-pmf* $(\lambda x.$ *case x of None* $\Rightarrow$ *1* $-$ *enn2real* $(\int^+ x.$ *ennreal* $(f\ x)$ *∂count-space UNIV*$)$ | *Some* $x' \Rightarrow$ *max 0* $(f\ x'))$

context
assumes *prob*: $(\int^+ x.$ *ennreal* $(f\ x)$ *∂count-space UNIV*$) \leq$ *1*
begin

lemma *nn-integral-embed-spmf-eq-1*:
$(\int^+ x.$ *ennreal* (*case x of None* $\Rightarrow$ *1* $-$ *enn2real* $(\int^+ x.$ *ennreal* $(f\ x)$ *∂count-space UNIV*$)$ | *Some* $x' \Rightarrow$ *max 0* $(f\ x'))$ *∂count-space UNIV*$) =$ *1*
(**is** *?lhs* = - **is** $(\int^+ x.\ ?f\ x\ \partial ?M) =$ -)
proof −
have *?lhs* $= \int^+ x.\ ?f\ x *$ *indicator* $\{None\}\ x +$ *?f x ∗ indicator* (*range Some*) *x ∂?M*
by(*rule nn-integral-cong*)(*auto split*: *split-indicator*)
also have ... $=$ (*1* $-$ *enn2real* $(\int^+ x.$ *ennreal* $(f\ x)$ *∂count-space UNIV*$)) +$ $\int^+ x.$ *?f x ∗ indicator* (*range Some*) *x ∂?M*
(**is** - = *?None* + *?Some*)
by(*subst nn-integral-add*)(*simp-all add*: *AE-count-space max-def le-diff-eq real-le-ereal-iff one-ereal-def*[*symmetric*] *prob split*: *option.split*)
also have *?Some* $= \int^+ x.$ *?f x ∂count-space* (*range Some*)
by(*simp add*: *nn-integral-count-space-indicator*)
also have *count-space* (*range Some*) = *embed-measure* (*count-space UNIV*) *Some*
by(*simp add*: *embed-measure-count-space*)
also have $(\int^+ x.\ ?f\ x\ \partial \ldots) = \int^+ x.$ *ennreal* $(f\ x)$ *∂count-space UNIV*
by(*subst nn-integral-embed-measure*)(*simp-all add*: *measurable-embed-measure1*)
also have *?None* + ... = *1* **using** *prob*
by(*auto simp add*: *ennreal-minus*[*symmetric*] *ennreal-1*[*symmetric*] *ennreal-enn2real-if top-unique simp del*: *ennreal-1*)(*simp add*: *diff-add-self-ennreal*)
finally show *?thesis* **.**
qed

lemma *pmf-embed-spmf-None*: *pmf embed-spmf None* = *1* $-$ *enn2real* $(\int^+ x.$ *ennreal* $(f\ x)$ *∂count-space UNIV*$)$
unfolding *embed-spmf-def*
apply(*subst pmf-embed-pmf*)

subgoal using *prob* **by**(*simp add*: *field-simps enn2real-leI split*: *option.split*)
apply(*rule nn-integral-embed-spmf-eq-1*)
apply *simp*
done

lemma *spmf-embed-spmf* [*simp*]: *spmf embed-spmf x* = *max 0* (*f x*)
unfolding *embed-spmf-def*
apply(*subst pmf-embed-pmf*)
subgoal using *prob* **by**(*simp add*: *field-simps enn2real-leI split*: *option.split*)
apply(*rule nn-integral-embed-spmf-eq-1*)
apply *simp*
done

end

end

lemma *embed-spmf-K-0*[*simp*]: *embed-spmf* (*λ*-. *0*) = *return-pmf None* (**is** *?lhs* = *?rhs*)
by(*rule spmf-eqI*)(*simp add*: *zero-ereal-def*[*symmetric*])

25.10 Ordering on spmfs

rel-pmf does not preserve a ccpo structure. Counterexample by Saheb-Djahromi: Take prefix order over *bool llist* and the set *range* (*λn* :: *nat*. *uniform* (*llist-n n*)) where *llist-n* is the set of all *llist*s of length *n* and *uniform* returns a uniform distribution over the given set. The set forms a chain in *ord-pmf lprefix*, but it has not an upper bound. Any upper bound may contain only infinite lists in its support because otherwise it is not greater than the *n+1*-st element in the chain where *n* is the length of the finite list. Moreover its support must contain all infinite lists, because otherwise there is a finite list all of whose finite extensions are not in the support - a contradiction to the upper bound property. Hence, the support is uncountable, but pmf's only have countable support.

However, if all chains in the ccpo are finite, then it should preserve the ccpo structure.

abbreviation *ord-spmf* :: ('*a* ⇒ '*a* ⇒ *bool*) ⇒ '*a spmf* ⇒ '*a spmf* ⇒ *bool*
where *ord-spmf ord* ≡ *rel-pmf* (*ord-option ord*)

locale *ord-spmf-syntax* **begin**
notation *ord-spmf* (**infix** ⊑₁ *60*)
end

lemma *ord-spmf-map-spmf1*: *ord-spmf R* (*map-spmf f p*) = *ord-spmf* (*λx*. *R* (*f x*)) *p*
by(*simp add*: *pmf.rel-map*[*abs-def*] *ord-option-map1*[*abs-def*])

lemma *ord-spmf-map-spmf2*: *ord-spmf R p* (*map-spmf f q*) = *ord-spmf* (λ*x y*. *R x* (*f y*)) *p q*
by(*simp add*: *pmf.rel-map ord-option-map2*)

lemma *ord-spmf-map-spmf12*: *ord-spmf R* (*map-spmf f p*) (*map-spmf f q*) = *ord-spmf* (λ*x y*. *R* (*f x*) (*f y*)) *p q*
by(*simp add*: *pmf.rel-map ord-option-map1*[*abs-def*] *ord-option-map2*)

lemmas *ord-spmf-map-spmf* = *ord-spmf-map-spmf1 ord-spmf-map-spmf2 ord-spmf-map-spmf12*

context fixes *ord* :: *'a* ⇒ *'a* ⇒ *bool* (**structure**) **begin**
interpretation *ord-spmf-syntax* **.**

lemma *ord-spmfI*:
⟦ ⋀*x y*. (*x*, *y*) ∈ *set-spmf pq* ⟹ *ord x y*; *map-spmf fst pq* = *p*; *map-spmf snd pq* = *q* ⟧
⟹ *p* ⊑ *q*
by(*rule rel-pmf.intros*[**where** *pq*=*map-pmf* (λ*x*. *case x of None* ⇒ (*None*, *None*) | *Some* (*a*, *b*) ⇒ (*Some a*, *Some b*)) *pq*])
(*auto simp add*: *pmf.map-comp o-def in-set-spmf split*: *option.splits intro*: *pmf.map-cong*)

lemma *ord-spmf-None* [*simp*]: *return-pmf None* ⊑ *x*
by(*rule rel-pmf.intros*[**where** *pq*=*map-pmf* (*Pair None*) *x*])(*auto simp add*: *pmf.map-comp o-def*)

lemma *ord-spmf-reflI*: (⋀*x*. *x* ∈ *set-spmf p* ⟹ *ord x x*) ⟹ *p* ⊑ *p*
by(*rule rel-pmf-reflI ord-option-reflI*)+(*auto simp add*: *in-set-spmf*)

lemma *rel-spmf-inf*:
assumes *p* ⊑ *q*
and *q* ⊑ *p*
and *refl*: *reflp ord*
and *trans*: *transp ord*
shows *rel-spmf* (*inf ord ord*$^{-1-1}$) *p q*
proof −
from ‹*p* ⊑ *q*› ‹*q* ⊑ *p*›
have *rel-pmf* (*inf* (*ord-option ord*) (*ord-option ord*)$^{-1-1}$) *p q*
by(*rule rel-pmf-inf*)(*blast intro*: *reflp-ord-option transp-ord-option refl trans*)+
also have *inf* (*ord-option ord*) (*ord-option ord*)$^{-1-1}$ = *rel-option* (*inf ord ord*$^{-1-1}$)
by(*auto simp add*: *fun-eq-iff elim*: *ord-option.cases option.rel-cases*)
finally show *?thesis* **.**
qed

end

lemma *ord-spmf-return-spmf2*: *ord-spmf R p* (*return-spmf y*) ⟷ (∀ *x*∈*set-spmf p*. *R x y*)
by(*auto simp add*: *rel-pmf-return-pmf2 in-set-spmf ord-option.simps intro*: *ccontr*)

lemma *ord-spmf-mono*: ⟦ *ord-spmf A p q*; ⋀*x y*. *A x y* ⟹ *B x y* ⟧ ⟹ *ord-spmf B p q*
by(*erule rel-pmf-mono*)(*erule ord-option-mono*)

lemma *ord-spmf-compp*: *ord-spmf* (*A OO B*) = *ord-spmf A OO ord-spmf B*
by(*simp add*: *ord-option-compp pmf.rel-compp*)

lemma *ord-spmf-bindI*:
  assumes *pq*: *ord-spmf R p q*
  and *fg*: ⋀*x y*. *R x y* ⟹ *ord-spmf P* (*f x*) (*g y*)
  shows *ord-spmf P* (*p* ⤜ *f*) (*q* ⤜ *g*)
unfolding *bind-spmf-def* **using** *pq*
by(*rule rel-pmf-bindI*)(*auto split*: *option.split intro*: *fg*)

lemma *ord-spmf-bind-reflI*:
  (⋀*x*. *x* ∈ *set-spmf p* ⟹ *ord-spmf R* (*f x*) (*g x*))
  ⟹ *ord-spmf R* (*p* ⤜ *f*) (*p* ⤜ *g*)
by(*rule ord-spmf-bindI*[**where** *R*=λ*x y*. *x* = *y* ∧ *x* ∈ *set-spmf p*])(*auto intro*: *ord-spmf-reflI*)

lemma *ord-pmf-increaseI*:
  assumes *le*: ⋀*x*. *spmf p x* ≤ *spmf q x*
  and *refl*: ⋀*x*. *x* ∈ *set-spmf p* ⟹ *R x x*
  shows *ord-spmf R p q*
proof(*rule rel-pmf.intros*)
  define *pq* **where** *pq* = *embed-pmf*
    (λ(*x*, *y*). *case x of Some x'* ⇒ (*case y of Some y'* ⇒ *if x' = y' then spmf p x' else 0* | *None* ⇒ *0*)
      | *None* ⇒ (*case y of None* ⇒ *pmf q None* | *Some y'* ⇒ *spmf q y' − spmf p y'*))
     (**is** - = *embed-pmf ?f*)
  have *nonneg*: ⋀*xy*. *?f xy* ≥ *0*
    by(*clarsimp simp add*: *le field-simps split*: *option.split*)
  have *integral*: (∫⁺ *xy*. *?f xy* ∂*count-space UNIV*) = *1* (**is** *nn-integral ?M* - = -)
  proof −
    have (∫⁺ *xy*. *?f xy* ∂*count-space UNIV*) =
      ∫⁺ *xy*. *ennreal* (*?f xy*) ∗ *indicator* {(*None*, *None*)} *xy* +
          ennreal (*?f xy*) ∗ *indicator* (*range* (λ*x*. (*None*, *Some x*))) *xy* +
          ennreal (*?f xy*) ∗ *indicator* (*range* (λ*x*. (*Some x*, *Some x*))) *xy* ∂*?M*
      by(*rule nn-integral-cong*)(*auto split*: *split-indicator option.splits if-split-asm*)
    also have ... = (∫⁺ *xy*. *?f xy* ∗ *indicator* {(*None*, *None*)} *xy* ∂*?M*) +
      (∫⁺ *xy*. *ennreal* (*?f xy*) ∗ *indicator* (*range* (λ*x*. (*None*, *Some x*))) *xy* ∂*?M*)
+
      (∫⁺ *xy*. *ennreal* (*?f xy*) ∗ *indicator* (*range* (λ*x*. (*Some x*, *Some x*))) *xy* ∂*?M*)
     (**is** - = *?None* + *?Some2* + *?Some*)
    by(*subst nn-integral-add*)(*simp-all add*: *nn-integral-add AE-count-space le-diff-eq le split*: *option.split*)
    also have *?None* = *pmf q None* **by** *simp*
    also have *?Some2* = ∫⁺ *x*. *ennreal* (*spmf q x*) − *spmf p x* ∂*count-space UNIV*

by(*simp add*: *nn-integral-count-space-indicator*[*symmetric*] *embed-measure-count-space*[*symmetric*] *inj-on-def nn-integral-embed-measure measurable-embed-measure1 ennreal-minus*)
also have ... = $(\int^+ x.\ spmf\ q\ x\ \partial count\text{-}space\ UNIV) - (\int^+ x.\ spmf\ p\ x\ \partial count\text{-}space\ UNIV)$
(**is** - = *?Some2'* − *?Some2''*)
by(*subst nn-integral-diff*)(*simp-all add*: *le nn-integral-spmf-neq-top*)
also have *?Some* = $\int^+ x.\ spmf\ p\ x\ \partial count\text{-}space\ UNIV$
by(*simp add*: *nn-integral-count-space-indicator*[*symmetric*] *embed-measure-count-space*[*symmetric*] *inj-on-def nn-integral-embed-measure measurable-embed-measure1*)
also have *pmf q None* + (*?Some2'* − *?Some2''*) + ... = *pmf q None* + *?Some2'*
by(*auto simp add*: *diff-add-self-ennreal le intro*!: *nn-integral-mono*)
also have ... = $\int^+ x.\ ennreal\ (pmf\ q\ x) * indicator\ \{None\}\ x + ennreal\ (pmf\ q\ x) * indicator\ (range\ Some)\ x\ \partial count\text{-}space\ UNIV$
by(*subst nn-integral-add*)(*simp-all add*: *nn-integral-count-space-indicator*[*symmetric*] *embed-measure-count-space*[*symmetric*] *nn-integral-embed-measure measurable-embed-measure1*)
also have ... = $\int^+ x.\ pmf\ q\ x\ \partial count\text{-}space\ UNIV$
by(*rule nn-integral-cong*)(*auto split*: *split-indicator*)
also have ... = *1* **by**(*simp add*: *nn-integral-pmf*)
finally show *?thesis* **.**
qed
note *f* = *nonneg integral*

{ fix *x y*
assume (*x*, *y*) ∈ *set-pmf pq*
hence *?f* (*x*, *y*) ≠ *0* **unfolding** *pq-def* **by**(*simp add*: *set-embed-pmf*[*OF f*])
then show *ord-option R x y*
by(*simp add*: *spmf-eq-0-set-spmf refl split*: *option.split-asm if-split-asm*) **}**

have *weight-le*: *weight-spmf p* ≤ *weight-spmf q*
by(*subst ennreal-le-iff*[*symmetric*])(*auto simp add*: *weight-spmf-eq-nn-integral-spmf intro*!: *nn-integral-mono le*)

show *map-pmf fst pq* = *p*
proof(*rule pmf-eqI*)
fix *i*
have *ennreal* (*pmf* (*map-pmf fst pq*) *i*) = $(\int^+ y.\ pmf\ pq\ (i, y)\ \partial count\text{-}space\ UNIV)$
unfolding *pq-def ennreal-pmf-map*
apply(*simp add*: *embed-pmf.rep-eq*[*OF f*] *o-def emeasure-density nn-integral-count-space-indicator*[*symmet*
apply(*subst pmf-embed-pmf*[*OF f*])
apply(*rule nn-integral-bij-count-space*[*symmetric*])
apply(*auto simp add*: *bij-betw-def inj-on-def*)
done
also have ... = *pmf p i*
proof(*cases i*)
case (*Some x*)
have $(\int^+ y.\ pmf\ pq\ (Some\ x, y)\ \partial count\text{-}space\ UNIV) = \int^+ y.\ pmf\ p\ (Some\ x) * indicator\ \{Some\ x\}\ y\ \partial count\text{-}space\ UNIV$

by(*rule nn-integral-cong*)(*simp add*: *pq-def pmf-embed-pmf*[*OF f*] *split*: *option.split*)
then show *?thesis* **using** *Some* **by** *simp*
next
case *None*
have (∫⁺ *y*. *pmf pq* (*None*, *y*) ∂*count-space UNIV*) =
(∫⁺ *y*. *ennreal* (*pmf pq* (*None*, *Some* (*the y*))) ∗ *indicator* (*range Some*) *y* +
ennreal (*pmf pq* (*None*, *None*)) ∗ *indicator* {*None*} *y* ∂*count-space UNIV*)
by(*rule nn-integral-cong*)(*auto split*: *split-indicator*)
also have ... = (∫⁺ *y*. *ennreal* (*pmf pq* (*None*, *Some* (*the y*))) ∂*count-space* (*range Some*)) + *pmf pq* (*None*, *None*)
by(*subst nn-integral-add*)(*simp-all add*: *nn-integral-count-space-indicator*)
also have ... = (∫⁺ *y*. *ennreal* (*spmf q y*) − *ennreal* (*spmf p y*) ∂*count-space UNIV*) + *pmf q None*
by(*simp add*: *pq-def pmf-embed-pmf*[*OF f*] *embed-measure-count-space*[*symmetric*] *nn-integral-embed-measure measurable-embed-measure1 ennreal-minus*)
also have (∫⁺ *y*. *ennreal* (*spmf q y*) − *ennreal* (*spmf p y*) ∂*count-space UNIV*) =
(∫⁺ *y*. *spmf q y* ∂*count-space UNIV*) − (∫⁺ *y*. *spmf p y* ∂*count-space UNIV*)
by(*subst nn-integral-diff*)(*simp-all add*: *AE-count-space le nn-integral-spmf-neq-top split*: *split-indicator*)
also have ... = *pmf p None* − *pmf q None*
by(*simp add*: *pmf-None-eq-weight-spmf weight-spmf-eq-nn-integral-spmf*[*symmetric*] *ennreal-minus*)
also have ... = *ennreal* (*pmf p None*) − *ennreal* (*pmf q None*) **by**(*simp add*: *ennreal-minus*)
finally show *?thesis* **using** *None weight-le*
by(*auto simp add*: *diff-add-self-ennreal pmf-None-eq-weight-spmf intro*: *ennreal-leI*)
qed
finally show *pmf* (*map-pmf fst pq*) *i* = *pmf p i* **by** *simp*
qed

show *map-pmf snd pq* = *q*
proof(*rule pmf-eqI*)
fix *i*
have *ennreal* (*pmf* (*map-pmf snd pq*) *i*) = (∫⁺ *x*. *pmf pq* (*x*, *i*) ∂*count-space UNIV*)
unfolding *pq-def ennreal-pmf-map*
apply(*simp add*: *embed-pmf.rep-eq*[*OF f*] *o-def emeasure-density nn-integral-count-space-indicator*[*symmet*
apply(*subst pmf-embed-pmf*[*OF f*])
apply(*rule nn-integral-bij-count-space*[*symmetric*])
apply(*auto simp add*: *bij-betw-def inj-on-def*)
done
also have ... = *ennreal* (*pmf q i*)
proof(*cases i*)

case *None*
have $(\int^+ x.\ pmf\ pq\ (x, None)\ \partial count\text{-}space\ UNIV) = \int^+ x.\ pmf\ q\ None * indicator\ \{None :: 'a\ option\}\ x\ \partial count\text{-}space\ UNIV$
by(*rule nn-integral-cong*)(*simp add*: *pq-def pmf-embed-pmf*[*OF f*] *split*: *option.split*)
then show *?thesis* **using** *None* **by** *simp*
next
case (*Some y*)
have $(\int^+ x.\ pmf\ pq\ (x, Some\ y)\ \partial count\text{-}space\ UNIV) =$
$(\int^+ x.\ ennreal\ (pmf\ pq\ (x, Some\ y)) * indicator\ (range\ Some)\ x +$
$ennreal\ (pmf\ pq\ (None, Some\ y)) * indicator\ \{None\}\ x\ \partial count\text{-}space\ UNIV)$
by(*rule nn-integral-cong*)(*auto split*: *split-indicator*)
also have $\ldots = (\int^+ x.\ ennreal\ (pmf\ pq\ (x, Some\ y)) * indicator\ (range\ Some)\ x\ \partial count\text{-}space\ UNIV) + pmf\ pq\ (None, Some\ y)$
by(*subst nn-integral-add*)(*simp-all*)
also have $\ldots = (\int^+ x.\ ennreal\ (spmf\ p\ y) * indicator\ \{Some\ y\}\ x\ \partial count\text{-}space\ UNIV) + (spmf\ q\ y - spmf\ p\ y)$
by(*auto simp add*: *pq-def pmf-embed-pmf*[*OF f*] *one-ereal-def*[*symmetric*] *simp del*: *nn-integral-indicator-singleton intro*!: *arg-cong2*[**where** *f*=(+)] *nn-integral-cong split*: *option.split*)
also have $\ldots = spmf\ q\ y$ **by**(*simp add*: *ennreal-minus*[*symmetric*] *le*)
finally show *?thesis* **using** *Some* **by** *simp*
qed
finally show $pmf\ (map\text{-}pmf\ snd\ pq)\ i = pmf\ q\ i$ **by** *simp*
qed
qed

lemma *ord-spmf-eq-leD*:
assumes *ord-spmf* (=) *p q*
shows $spmf\ p\ x \leq spmf\ q\ x$
proof(*cases* $x \in set\text{-}spmf\ p$)
case *False*
thus *?thesis* **by**(*simp add*: *in-set-spmf-iff-spmf*)
next
case *True*
from *assms* **obtain** *pq*
where *pq*: $\bigwedge x\ y.\ (x, y) \in set\text{-}pmf\ pq \Longrightarrow ord\text{-}option\ (=)\ x\ y$
and *p*: $p = map\text{-}pmf\ fst\ pq$
and *q*: $q = map\text{-}pmf\ snd\ pq$ **by** *cases auto*
have $ennreal\ (spmf\ p\ x) = integral^N\ pq\ (indicator\ (fst -` \{Some\ x\}))$
using *p* **by**(*simp add*: *ennreal-pmf-map*)
also have $\ldots = integral^N\ pq\ (indicator\ \{(Some\ x, Some\ x)\})$
by(*rule nn-integral-cong-AE*)(*auto simp add*: *AE-measure-pmf-iff split*: *split-indicator dest*: *pq*)
also have $\ldots \leq integral^N\ pq\ (indicator\ (snd -` \{Some\ x\}))$
by(*rule nn-integral-mono*) *simp*
also have $\ldots = ennreal\ (spmf\ q\ x)$ **using** *q* **by**(*simp add*: *ennreal-pmf-map*)
finally show *?thesis* **by** *simp*

qed

lemma *ord-spmf-eqD-set-spmf*: *ord-spmf* (=) *p q* ⟹ *set-spmf p* ⊆ *set-spmf q*
by(*rule subsetI*)(*drule-tac x*=*x* **in** *ord-spmf-eq-leD*, *auto simp add*: *in-set-spmf-iff-spmf*)

lemma *ord-spmf-eqD-emeasure*:
ord-spmf (=) *p q* ⟹ *emeasure* (*measure-spmf p*) *A* ≤ *emeasure* (*measure-spmf q*) *A*
by(*auto intro*!: *nn-integral-mono split*: *split-indicator dest*: *ord-spmf-eq-leD simp add*: *nn-integral-measure-spmf nn-integral-indicator*[*symmetric*])

lemma *ord-spmf-eqD-measure-spmf*: *ord-spmf* (=) *p q* ⟹ *measure-spmf p* ≤ *measure-spmf q*
by (*subst le-measure*) (*auto simp*: *ord-spmf-eqD-emeasure*)

25.11 CCPO structure for the flat ccpo *ord-option* (=)

context fixes *Y* :: *'a spmf set* **begin**

definition *lub-spmf* :: *'a spmf*
where *lub-spmf* = *embed-spmf* (λ*x*. *enn2real* (*SUP p* ∈ *Y*. *ennreal* (*spmf p x*)))
— We go through *ennreal* to have a sensible definition even if *Y* is empty.

lemma *lub-spmf-empty* [*simp*]: *SPMF.lub-spmf* {} = *return-pmf None*
by(*simp add*: *SPMF.lub-spmf-def bot-ereal-def*)

context assumes *chain*: *Complete-Partial-Order.chain* (*ord-spmf* (=)) *Y* **begin**

lemma *chain-ord-spmf-eqD*: *Complete-Partial-Order.chain* (≤) ((λ*p x*. *ennreal* (*spmf p x*)) ‘ *Y*)
(**is** *Complete-Partial-Order.chain* - (*?f* ‘ -))
proof(*rule chainI*)
fix *f g*
assume *f* ∈ *?f* ‘ *Y g* ∈ *?f* ‘ *Y*
then obtain *p q* **where** *f*: *f* = *?f p p* ∈ *Y* **and** *g*: *g* = *?f q q* ∈ *Y* **by** *blast*
from *chain* ‹*p* ∈ *Y*› ‹*q* ∈ *Y*› **have** *ord-spmf* (=) *p q* ∨ *ord-spmf* (=) *q p* **by**(*rule chainD*)
thus *f* ≤ *g* ∨ *g* ≤ *f*
proof
assume *ord-spmf* (=) *p q*
hence ⋀*x*. *spmf p x* ≤ *spmf q x* **by**(*rule ord-spmf-eq-leD*)
hence *f* ≤ *g* **unfolding** *f g* **by**(*auto intro*: *le-funI*)
thus *?thesis* ..
next
assume *ord-spmf* (=) *q p*
hence ⋀*x*. *spmf q x* ≤ *spmf p x* **by**(*rule ord-spmf-eq-leD*)
hence *g* ≤ *f* **unfolding** *f g* **by**(*auto intro*: *le-funI*)
thus *?thesis* ..
qed

qed

lemma *ord-spmf-eq-pmf-None-eq*:
assumes *le*: *ord-spmf* (=) *p q*
and *None*: *pmf p None* = *pmf q None*
shows *p* = *q*
proof(*rule spmf-eqI*)
fix *i*
from *le* **have** *le'*: ⋀*x*. *spmf p x* ≤ *spmf q x* **by**(*rule ord-spmf-eq-leD*)
have (∫⁺ *x*. *ennreal* (*spmf q x*) − *spmf p x* ∂*count-space UNIV*) =
(∫⁺ *x*. *spmf q x* ∂*count-space UNIV*) − (∫⁺ *x*. *spmf p x* ∂*count-space UNIV*)
by(*subst nn-integral-diff*)(*simp-all add*: *AE-count-space le' nn-integral-spmf-neq-top*)
also have ... = (*1* − *pmf q None*) − (*1* − *pmf p None*) **unfolding** *pmf-None-eq-weight-spmf*
by(*simp add*: *weight-spmf-eq-nn-integral-spmf*[*symmetric*] *ennreal-minus*)
also have ... = *0* **using** *None* **by** *simp*
finally have ⋀*x*. *spmf q x* ≤ *spmf p x*
by(*simp add*: *nn-integral-0-iff-AE AE-count-space ennreal-minus ennreal-eq-0-iff*)
with *le'* **show** *spmf p i* = *spmf q i* **by**(*rule antisym*)
qed

lemma *ord-spmf-eqD-pmf-None*:
assumes *ord-spmf* (=) *x y*
shows *pmf x None* ≥ *pmf y None*
using *assms*
apply *cases*
apply(*clarsimp simp only*: *ennreal-le-iff*[*symmetric*, *OF pmf-nonneg*] *ennreal-pmf-map*)
apply(*fastforce simp add*: *AE-measure-pmf-iff intro*!: *nn-integral-mono-AE*)
done

Chains on *'a spmf* maintain countable support. Thanks to Johannes Hölzl for the proof idea.

lemma *spmf-chain-countable*: *countable* (⋃*p*∈*Y*. *set-spmf p*)
proof(*cases Y* = {})
case *Y*: *False*
show *?thesis*
proof(*cases* ∃ *x*∈*Y*. ∀ *y*∈*Y*. *ord-spmf* (=) *y x*)
case *True*
then obtain *x* **where** *x*: *x* ∈ *Y* **and** *upper*: ⋀*y*. *y* ∈ *Y* ⟹ *ord-spmf* (=) *y x*
by *blast*
hence (⋃*x*∈*Y*. *set-spmf x*) ⊆ *set-spmf x* **by**(*auto dest*: *ord-spmf-eqD-set-spmf*)
thus *?thesis* **by**(*rule countable-subset*) *simp*
next
case *False*
define *N* :: *'a option pmf* ⇒ *real* **where** *N p* = *pmf p None* **for** *p*

have *N-less-imp-le-spmf*: ⟦ *x* ∈ *Y*; *y* ∈ *Y*; *N y* < *N x* ⟧ ⟹ *ord-spmf* (=) *x y*
for *x y*
using *chainD*[*OF chain*, *of x y*] *ord-spmf-eqD-pmf-None*[*of x y*] *ord-spmf-eqD-pmf-None*[*of y x*]

```
    by (auto simp: N-def)
  have N-eq-imp-eq: ⟦ x ∈ Y; y ∈ Y; N y = N x ⟧ ⟹ x = y for x y
  using chainD[OF chain, of x y] by(auto simp add: N-def dest: ord-spmf-eq-pmf-None-eq)

  have NC: N ‘ Y ≠ {} bdd-below (N ‘ Y)
    using ‹Y ≠ {}› by(auto intro!: bdd-belowI[of - 0] simp: N-def)
  have NC-less: Inf (N ‘ Y) < N x if x ∈ Y for x unfolding cInf-less-iff[OF
NC]
  proof(rule ccontr)
    assume **: ¬ (∃ y∈N ‘ Y. y < N x)
    { fix y
      assume y ∈ Y
     with ** consider N x < N y | N x = N y by(auto simp add: not-less le-less)
      hence ord-spmf (=) y x using ‹y ∈ Y› ‹x ∈ Y›
       by cases(auto dest: N-less-imp-le-spmf N-eq-imp-eq intro: ord-spmf-reflI) }
    with False ‹x ∈ Y› show False by blast
  qed

  from NC have Inf (N ‘ Y) ∈ closure (N ‘ Y) by (intro closure-contains-Inf)
  then obtain X' where ⋀n. X' n ∈ N ‘ Y and X': X' ⟶ Inf (N ‘ Y)
    unfolding closure-sequential by auto
  then obtain X where X: ⋀n. X n ∈ Y and X' = (λn. N (X n)) unfolding
image-iff Bex-def by metis

  with X' have seq: (λn. N (X n)) ⟶ Inf (N ‘ Y) by simp
  have (⋃ x ∈ Y. set-spmf x) ⊆ (⋃ n. set-spmf (X n))
  proof(rule UN-least)
    fix x
    assume x ∈ Y
    from order-tendstoD(2)[OF seq NC-less[OF ‹x ∈ Y›]]
    obtain i where N (X i) < N x by (auto simp: eventually-sequentially)
    thus set-spmf x ⊆ (⋃ n. set-spmf (X n)) using X ‹x ∈ Y›
      by(blast dest: N-less-imp-le-spmf ord-spmf-eqD-set-spmf)
  qed
  thus ?thesis by(rule countable-subset) simp
 qed
qed simp

lemma lub-spmf-subprob: (∫⁺ x. (SUP p ∈ Y. ennreal (spmf p x)) ∂count-space
UNIV) ≤ 1
proof(cases Y = {})
 case True
 thus ?thesis by(simp add: bot-ennreal)
next
 case False
 let ?B = ⋃ p∈Y. set-spmf p
 have countable: countable ?B by(rule spmf-chain-countable)

 have (∫⁺ x. (SUP p∈Y. ennreal (spmf p x)) ∂count-space UNIV) =
```

```
      (∫⁺ x. (SUP p∈Y. ennreal (spmf p x) * indicator ?B x) ∂count-space UNIV)
    by (intro nn-integral-cong arg-cong [of - - Sup]) (auto split: split-indicator simp
add: spmf-eq-0-set-spmf)
  also have ... = (∫⁺ x. (SUP p∈Y. ennreal (spmf p x)) ∂count-space ?B)
    unfolding ennreal-indicator[symmetric] using False
   by(subst SUP-mult-right-ennreal[symmetric])(simp add: ennreal-indicator nn-integral-count-space-indicator)
  also have ... = (SUP p∈Y. ∫⁺ x. spmf p x ∂count-space ?B) using False -
countable
   by(rule nn-integral-monotone-convergence-SUP-countable)(rule chain-ord-spmf-eqD)
  also have ... ≤ 1
  proof(rule SUP-least)
    fix p
    assume p ∈ Y
   have (∫⁺ x. spmf p x ∂count-space ?B) = ∫⁺ x. ennreal (spmf p x) * indicator
?B x ∂count-space UNIV
      by(simp add: nn-integral-count-space-indicator)
    also have ... = ∫⁺ x. spmf p x ∂count-space UNIV
     by(rule nn-integral-cong)(auto split: split-indicator simp add: spmf-eq-0-set-spmf
‹p ∈ Y›)
    also have ... ≤ 1
      by(simp add: weight-spmf-eq-nn-integral-spmf[symmetric] weight-spmf-le-1)
    finally show (∫⁺ x. spmf p x ∂count-space ?B) ≤ 1 .
  qed
  finally show ?thesis .
qed

lemma spmf-lub-spmf:
  assumes Y ≠ {}
  shows spmf lub-spmf x = (SUP p ∈ Y. spmf p x)
proof −
  from assms obtain p where p ∈ Y by auto
   have spmf lub-spmf x = max 0 (enn2real (SUP p∈Y. ennreal (spmf p x)))
unfolding lub-spmf-def
      by(rule spmf-embed-spmf)(simp del: SUP-eq-top-iff Sup-eq-top-iff add: en-
nreal-enn2real-if SUP-spmf-neq-top' lub-spmf-subprob)
  also have ... = enn2real (SUP p∈Y. ennreal (spmf p x))
    by(rule max-absorb2)(simp)
  also have ... = enn2real (ennreal (SUP p ∈ Y. spmf p x)) using assms
   by(subst ennreal-SUP[symmetric])(simp-all add: SUP-spmf-neq-top' del: SUP-eq-top-iff
Sup-eq-top-iff)
  also have 0 ≤ (⨆p∈Y. spmf p x) using assms
    by(auto intro!: cSUP-upper2 bdd-aboveI[where M=1] simp add: pmf-le-1)
  then have enn2real (ennreal (SUP p ∈ Y. spmf p x)) = (SUP p ∈ Y. spmf p x)
    by(rule enn2real-ennreal)
  finally show ?thesis .
qed

lemma ennreal-spmf-lub-spmf: Y ≠ {} ⟹ ennreal (spmf lub-spmf x) = (SUP
p∈Y. ennreal (spmf p x))
```

unfolding *spmf-lub-spmf* **by**(*subst ennreal-SUP*)(*simp-all add*: *SUP-spmf-neq-top'* *del*: *SUP-eq-top-iff Sup-eq-top-iff*)

lemma *lub-spmf-upper*:
assumes *p*: $p \in Y$
shows *ord-spmf* (=) *p lub-spmf*
proof(*rule ord-pmf-increaseI*)
fix *x*
from *p* **have** [*simp*]: $Y \neq \{\}$ **by** *auto*
from *p* **have** *ennreal* (*spmf p x*) ≤ (*SUP p*∈*Y*. *ennreal* (*spmf p x*)) **by**(*rule SUP-upper*)
also have ... = *ennreal* (*spmf lub-spmf x*) **using** *p*
by(*subst spmf-lub-spmf*)(*auto simp add*: *ennreal-SUP SUP-spmf-neq-top' simp del*: *SUP-eq-top-iff Sup-eq-top-iff*)
finally show *spmf p x* ≤ *spmf lub-spmf x* **by** *simp*
qed *simp*

lemma *lub-spmf-least*:
assumes *z*: $\bigwedge x.\ x \in Y \Longrightarrow$ *ord-spmf* (=) *x z*
shows *ord-spmf* (=) *lub-spmf z*
proof(*cases* $Y = \{\}$)
case *nonempty*: *False*
show *?thesis*
proof(*rule ord-pmf-increaseI*)
fix *x*
from *nonempty* **obtain** *p* **where** *p*: $p \in Y$ **by** *auto*
have *ennreal* (*spmf lub-spmf x*) = (*SUP p*∈*Y*. *ennreal* (*spmf p x*))
by(*subst spmf-lub-spmf*)(*auto simp add*: *ennreal-SUP SUP-spmf-neq-top' nonempty simp del*: *SUP-eq-top-iff Sup-eq-top-iff*)
also have ... ≤ *ennreal* (*spmf z x*) **by**(*rule SUP-least*)(*simp add*: *ord-spmf-eq-leD z*)
finally show *spmf lub-spmf x* ≤ *spmf z x* **by** *simp*
qed *simp*
qed *simp*

lemma *set-lub-spmf*: *set-spmf lub-spmf* = ($\bigcup p \in Y$. *set-spmf p*) (**is** *?lhs* = *?rhs*)
proof(*cases* $Y = \{\}$)
case [*simp*]: *False*
show *?thesis*
proof(*rule set-eqI*)
fix *x*
have *x* ∈ *?lhs* ⟷ *ennreal* (*spmf lub-spmf x*) > *0*
by(*simp-all add*: *in-set-spmf-iff-spmf less-le*)
also have ... ⟷ (∃ *p*∈*Y*. *ennreal* (*spmf p x*) > *0*)
by(*simp add*: *ennreal-spmf-lub-spmf less-SUP-iff*)
also have ... ⟷ *x* ∈ *?rhs*
by(*auto simp add*: *in-set-spmf-iff-spmf less-le*)
finally show *x* ∈ *?lhs* ⟷ *x* ∈ *?rhs* **.**
qed

qed *simp*

lemma *emeasure-lub-spmf*:
assumes *Y*: $Y \neq \{\}$
shows *emeasure* (*measure-spmf lub-spmf*) *A* = (*SUP y*∈*Y*. *emeasure* (*measure-spmf y*) *A*)
(**is** *?lhs* = *?rhs*)
proof −
let *?M* = *count-space* (*set-spmf lub-spmf*)
have *?lhs* = $\int^+ x.$ *ennreal* (*spmf lub-spmf x*) ∗ *indicator A x* ∂*?M*
by(*auto simp add*: *nn-integral-indicator*[*symmetric*] *nn-integral-measure-spmf'*)
also have ... = $\int^+ x.$ (*SUP y*∈*Y*. *ennreal* (*spmf y x*) ∗ *indicator A x*) ∂*?M*
unfolding *ennreal-indicator*[*symmetric*]
by(*simp add*: *spmf-lub-spmf assms ennreal-SUP*[*OF SUP-spmf-neq-top'*] *SUP-mult-right-ennreal*)
also from *assms* **have** ... = (*SUP y*∈*Y*. $\int^+ x.$ *ennreal* (*spmf y x*) ∗ *indicator A x* ∂*?M*)
proof(*rule nn-integral-monotone-convergence-SUP-countable*)
have (λ*i x*. *ennreal* (*spmf i x*) ∗ *indicator A x*) ' *Y* = (λ*f x*. *f x* ∗ *indicator A x*) ' (λ*p x*. *ennreal* (*spmf p x*)) ' *Y*
by(*simp add*: *image-image*)
also have *Complete-Partial-Order.chain* (≤) ... **using** *chain-ord-spmf-eqD*
by(*rule chain-imageI*)(*auto simp add*: *le-fun-def split*: *split-indicator*)
finally show *Complete-Partial-Order.chain* (≤) ((λ*i x*. *ennreal* (*spmf i x*) ∗ *indicator A x*) ' *Y*) **.**
qed *simp*
also have ... = (*SUP y*∈*Y*. $\int^+ x.$ *ennreal* (*spmf y x*) ∗ *indicator A x* ∂*count-space UNIV*)
by(*auto simp add*: *nn-integral-count-space-indicator set-lub-spmf spmf-eq-0-set-spmf split*: *split-indicator intro*!: *arg-cong* [*of - - Sup*] *image-cong nn-integral-cong*)
also have ... = *?rhs*
by(*auto simp add*: *nn-integral-indicator*[*symmetric*] *nn-integral-measure-spmf*)
finally show *?thesis* **.**
qed

lemma *measure-lub-spmf*:
assumes *Y*: $Y \neq \{\}$
shows *measure* (*measure-spmf lub-spmf*) *A* = (*SUP y*∈*Y*. *measure* (*measure-spmf y*) *A*) (**is** *?lhs* = *?rhs*)
proof −
have *ennreal ?lhs* = *ennreal ?rhs*
using *emeasure-lub-spmf*[*OF assms*] *SUP-emeasure-spmf-neq-top*[*of A Y*] *Y*
unfolding *measure-spmf.emeasure-eq-measure* **by**(*subst ennreal-SUP*)
moreover have *0* ≤ *?rhs* **using** *Y*
by(*auto intro*!: *cSUP-upper2 bdd-aboveI*[**where** *M=1*] *measure-spmf.subprob-measure-le-1*)
ultimately show *?thesis* **by**(*simp*)
qed

lemma *weight-lub-spmf*:
assumes *Y*: $Y \neq \{\}$

shows *weight-spmf lub-spmf* = (*SUP y*∈*Y*. *weight-spmf y*)
unfolding *weight-spmf-def* **by**(*rule measure-lub-spmf*) *fact*

lemma *measure-spmf-lub-spmf*:
assumes *Y*: *Y* ≠ {}
shows *measure-spmf lub-spmf* = (*SUP p*∈*Y*. *measure-spmf p*) (**is** *?lhs* = *?rhs*)
proof(*rule measure-eqI*)
from *assms* **obtain** *p* **where** *p*: *p* ∈ *Y* **by** *auto*
from *chain* **have** *chain'*: *Complete-Partial-Order.chain* (≤) (*measure-spmf* ' *Y*)
by(*rule chain-imageI*)(*rule ord-spmf-eqD-measure-spmf*)
show *sets ?lhs* = *sets ?rhs*
using *Y* **by** (*subst sets-SUP*) *auto*
show *emeasure ?lhs A* = *emeasure ?rhs A* **for** *A*
using *chain' Y p* **by** (*subst emeasure-SUP-chain*) (*auto simp*: *emeasure-lub-spmf*)
qed

end

end

lemma *partial-function-definitions-spmf*: *partial-function-definitions* (*ord-spmf* (=)) *lub-spmf*
(**is** *partial-function-definitions ?R* -)
proof
fix *x* **show** *?R x x* **by**(*simp add*: *ord-spmf-reflI*)
next
fix *x y z*
assume *?R x y ?R y z*
with *transp-ord-option*[*OF transp-equality*] **show** *?R x z* **by**(*rule transp-rel-pmf*[*THEN transpD*])
next
fix *x y*
assume *?R x y ?R y x*
thus *x* = *y*
by(*rule rel-pmf-antisym*)(*simp-all add*: *reflp-ord-option transp-ord-option antisymp-ord-option*)
next
fix *Y x*
assume *Complete-Partial-Order.chain ?R Y x* ∈ *Y*
then show *?R x* (*lub-spmf Y*)
by(*rule lub-spmf-upper*)
next
fix *Y z*
assume *Complete-Partial-Order.chain ?R Y* ⋀*x*. *x* ∈ *Y* ⟹ *?R x z*
then show *?R* (*lub-spmf Y*) *z*
by(*cases Y* = {})(*simp-all add*: *lub-spmf-least*)
qed

lemma *ccpo-spmf*: *class.ccpo lub-spmf* (*ord-spmf* (=)) (*mk-less* (*ord-spmf* (=)))

by(*rule ccpo partial-function-definitions-spmf*)+

interpretation *spmf*: *partial-function-definitions ord-spmf* (=) *lub-spmf*
rewrites *lub-spmf* {} ≡ *return-pmf None*
by(*rule partial-function-definitions-spmf*) *simp*

declaration ‹*Partial-Function.init spmf* ***term*** ‹*spmf.fixp-fun*›
term ‹*spmf.mono-body*› @{*thm spmf.fixp-rule-uc*} @{*thm spmf.fixp-induct-uc*}
NONE›

declare *spmf.leq-refl*[*simp*]
declare *admissible-leI*[*OF ccpo-spmf*, *cont-intro*]

abbreviation *mono-spmf* ≡ *monotone* (*fun-ord* (*ord-spmf* (=))) (*ord-spmf* (=))

lemma *lub-spmf-const* [*simp*]: *lub-spmf* {*p*} = *p*
by(*rule spmf-eqI*)(*simp add*: *spmf-lub-spmf*[*OF ccpo.chain-singleton*[*OF ccpo-spmf*]])

lemma *bind-spmf-mono'*:
assumes *fg*: *ord-spmf* (=) *f g*
and *hk*: ⋀*x* :: *'a*. *ord-spmf* (=) (*h x*) (*k x*)
shows *ord-spmf* (=) (*f* ⤜ *h*) (*g* ⤜ *k*)
unfolding *bind-spmf-def* **using** *assms*(*1*)
by(*rule rel-pmf-bindI*)(*auto split*: *option.split simp add*: *hk*)

lemma *bind-spmf-mono* [*partial-function-mono*]:
assumes *mf*: *mono-spmf B* **and** *mg*: ⋀*y*. *mono-spmf* (λ*f*. *C y f*)
shows *mono-spmf* (λ*f*. *bind-spmf* (*B f*) (λ*y*. *C y f*))
proof (*rule monotoneI*)
fix *f g* :: *'a* ⇒ *'b spmf*
assume *fg*: *fun-ord* (*ord-spmf* (=)) *f g*
with *mf* **have** *ord-spmf* (=) (*B f*) (*B g*) **by** (*rule monotoneD*[*of - - - f g*])
moreover from *mg* **have** ⋀*y'*. *ord-spmf* (=) (*C y' f*) (*C y' g*)
by (*rule monotoneD*) (*rule fg*)
ultimately show *ord-spmf* (=) (*bind-spmf* (*B f*) (λ*y*. *C y f*)) (*bind-spmf* (*B g*)
(λ*y'*. *C y' g*))
by(*rule bind-spmf-mono'*)
qed

lemma *monotone-bind-spmf1*: *monotone* (*ord-spmf* (=)) (*ord-spmf* (=)) (λ*y*. *bind-spmf y g*)
by(*rule monotoneI*)(*simp add*: *bind-spmf-mono' ord-spmf-reflI*)

lemma *monotone-bind-spmf2*:
assumes *g*: ⋀*x*. *monotone ord* (*ord-spmf* (=)) (λ*y*. *g y x*)
shows *monotone ord* (*ord-spmf* (=)) (λ*y*. *bind-spmf p* (*g y*))
by(*rule monotoneI*)(*auto intro*: *bind-spmf-mono' monotoneD*[*OF g*] *ord-spmf-reflI*)

lemma *bind-lub-spmf*:

assumes *chain*: *Complete-Partial-Order.chain* (*ord-spmf* (=)) *Y*
shows *bind-spmf* (*lub-spmf Y*) *f* = *lub-spmf* ((λ*p*. *bind-spmf p f*) ‘ *Y*) (**is** *?lhs* = *?rhs*)
proof(*cases Y* = {})
case *Y*: *False*
show *?thesis*
proof(*rule spmf-eqI*)
fix *i*
have *chain′*: *Complete-Partial-Order.chain* (≤) ((λ*p x*. *ennreal* (*spmf p x* ∗ *spmf* (*f x*) *i*)) ‘ *Y*)
using *chain* **by**(*rule chain-imageI*)(*auto simp add*: *le-fun-def dest*: *ord-spmf-eq-leD intro*: *mult-right-mono*)
have *chain″*: *Complete-Partial-Order.chain* (*ord-spmf* (=)) ((λ*p*. *p* ⤜ *f*) ‘ *Y*)
using *chain* **by**(*rule chain-imageI*)(*auto intro*!: *monotoneI bind-spmf-mono′ ord-spmf-reflI*)
let *?M* = *count-space* (*set-spmf* (*lub-spmf Y*))
have *ennreal* (*spmf ?lhs i*) = ∫⁺ *x*. *ennreal* (*spmf* (*lub-spmf Y*) *x*) ∗ *ennreal* (*spmf* (*f x*) *i*) ∂*?M*
by(*auto simp add*: *ennreal-spmf-lub-spmf ennreal-spmf-bind nn-integral-measure-spmf′*)
also have ... = ∫⁺ *x*. (*SUP p*∈*Y*. *ennreal* (*spmf p x* ∗ *spmf* (*f x*) *i*)) ∂*?M*
by(*subst ennreal-spmf-lub-spmf*[*OF chain Y*])(*subst SUP-mult-right-ennreal*, *simp-all add*: *ennreal-mult Y*)
also have ... = (*SUP p*∈*Y*. ∫⁺ *x*. *ennreal* (*spmf p x* ∗ *spmf* (*f x*) *i*) ∂*?M*)
using *Y chain′* **by**(*rule nn-integral-monotone-convergence-SUP-countable*) *simp*
also have ... = (*SUP p*∈*Y*. *ennreal* (*spmf* (*bind-spmf p f*) *i*))
by(*auto simp add*: *ennreal-spmf-bind nn-integral-measure-spmf nn-integral-count-space-indicator set-lub-spmf*[*OF chain*] *in-set-spmf-iff-spmf ennreal-mult intro*!: *arg-cong* [*of* - - *Sup*] *image-cong nn-integral-cong split*: *split-indicator*)
also have ... = *ennreal* (*spmf ?rhs i*) **using** *chain″* **by**(*simp add*: *ennreal-spmf-lub-spmf Y image-comp*)
finally show *spmf ?lhs i* = *spmf ?rhs i* **by** *simp*
qed
qed *simp*

lemma *map-lub-spmf*:
Complete-Partial-Order.chain (*ord-spmf* (=)) *Y*
⟹ *map-spmf f* (*lub-spmf Y*) = *lub-spmf* (*map-spmf f* ‘ *Y*)
unfolding *map-spmf-conv-bind-spmf*[*abs-def*] **by**(*simp add*: *bind-lub-spmf o-def*)

lemma *mcont-bind-spmf1*: *mcont lub-spmf* (*ord-spmf* (=)) *lub-spmf* (*ord-spmf* (=)) (λ*y*. *bind-spmf y f*)
using *monotone-bind-spmf1* **by**(*rule mcontI*)(*rule contI*, *simp add*: *bind-lub-spmf*)

lemma *bind-lub-spmf2*:
assumes *chain*: *Complete-Partial-Order.chain ord Y*
and *g*: ⋀*y*. *monotone ord* (*ord-spmf* (=)) (*g y*)
shows *bind-spmf x* (λ*y*. *lub-spmf* (*g y* ‘ *Y*)) = *lub-spmf* ((λ*p*. *bind-spmf x* (λ*y*. *g y p*)) ‘ *Y*)

(**is** *?lhs = ?rhs*)
proof(*cases Y = {}*)
case *Y*: *False*
show *?thesis*
proof(*rule spmf-eqI*)
fix *i*
have *chain'*: ⋀*y*. *Complete-Partial-Order.chain* (*ord-spmf* (=)) (*g y ' Y*)
using *chain g*[*THEN monotoneD*] **by**(*rule chain-imageI*)
have *chain''*: *Complete-Partial-Order.chain* (≤) ((λ*p y*. *ennreal* (*spmf x y* ∗ *spmf* (*g y p*) *i*)) *' Y*)
using *chain* **by**(*rule chain-imageI*)(*auto simp add*: *le-fun-def dest*: *ord-spmf-eq-leD monotoneD*[*OF g*] *intro*!: *mult-left-mono*)
have *chain'''*: *Complete-Partial-Order.chain* (*ord-spmf* (=)) ((λ*p*. *bind-spmf x* (λ*y*. *g y p*)) *' Y*)
using *chain* **by**(*rule chain-imageI*)(*rule monotone-bind-spmf2*[*OF g*, *THEN monotoneD*])

have *ennreal* (*spmf ?lhs i*) = ∫⁺ *y*. (*SUP p*∈*Y*. *ennreal* (*spmf x y* ∗ *spmf* (*g y p*) *i*)) ∂*count-space* (*set-spmf x*)
by(*simp add*: *ennreal-spmf-bind ennreal-spmf-lub-spmf*[*OF chain'*] *Y nn-integral-measure-spmf' SUP-mult-left-ennreal ennreal-mult image-comp*)
also have ... = (*SUP p*∈*Y*. ∫⁺ *y*. *ennreal* (*spmf x y* ∗ *spmf* (*g y p*) *i*) ∂*count-space* (*set-spmf x*))
unfolding *nn-integral-measure-spmf'* **using** *Y chain''*
by(*rule nn-integral-monotone-convergence-SUP-countable*) *simp*
also have ... = (*SUP p*∈*Y*. *ennreal* (*spmf* (*bind-spmf x* (λ*y*. *g y p*)) *i*))
by(*simp add*: *ennreal-spmf-bind nn-integral-measure-spmf' ennreal-mult*)
also have ... = *ennreal* (*spmf ?rhs i*) **using** *chain'''*
by(*auto simp add*: *ennreal-spmf-lub-spmf Y image-comp*)
finally show *spmf ?lhs i = spmf ?rhs i* **by** *simp*
qed
qed *simp*

lemma *mcont-bind-spmf* [*cont-intro*]:
assumes *f*: *mcont luba orda lub-spmf* (*ord-spmf* (=)) *f*
and *g*: ⋀*y*. *mcont luba orda lub-spmf* (*ord-spmf* (=)) (*g y*)
shows *mcont luba orda lub-spmf* (*ord-spmf* (=)) (λ*x*. *bind-spmf* (*f x*) (λ*y*. *g y x*))
proof(*rule spmf.mcont2mcont'*[*OF - - f*])
fix *z*
show *mcont lub-spmf* (*ord-spmf* (=)) *lub-spmf* (*ord-spmf* (=)) (λ*x*. *bind-spmf x* (λ*y*. *g y z*))
by(*rule mcont-bind-spmf1*)
next
fix *x*
let *?f* = λ*z*. *bind-spmf x* (λ*y*. *g y z*)
have *monotone orda* (*ord-spmf* (=)) *?f* **using** *mcont-mono*[*OF g*] **by**(*rule monotone-bind-spmf2*)
moreover have *cont luba orda lub-spmf* (*ord-spmf* (=)) *?f*

proof(*rule contI*)
fix *Y*
assume *chain*: *Complete-Partial-Order.chain orda Y* **and** *Y*: $Y \neq \{\}$
have *bind-spmf x* (λ*y*. *g y* (*luba Y*)) = *bind-spmf x* (λ*y*. *lub-spmf* (*g y* ‘ *Y*))
by(*rule bind-spmf-cong*)(*simp-all add*: *mcont-contD*[*OF g chain Y*])
also have ... = *lub-spmf* ((λ*p*. *x* ⤜ (λ*y*. *g y p*)) ‘ *Y*) **using** *chain*
by(*rule bind-lub-spmf2*)(*rule mcont-mono*[*OF g*])
finally show *bind-spmf x* (λ*y*. *g y* (*luba Y*)) = ... **.**
qed
ultimately show *mcont luba orda lub-spmf* (*ord-spmf* (=)) *?f* **by**(*rule mcontI*)
qed

lemma *bind-pmf-mono* [*partial-function-mono*]:
(⋀*y*. *mono-spmf* (λ*f*. *C y f*)) ⟹ *mono-spmf* (λ*f*. *bind-pmf p* (λ*x*. *C x f*))
using *bind-spmf-mono*[*of* λ-. *spmf-of-pmf p C*] **by** *simp*

lemma *map-spmf-mono* [*partial-function-mono*]: *mono-spmf B* ⟹ *mono-spmf* (λ*g*. *map-spmf f* (*B g*))
unfolding *map-spmf-conv-bind-spmf* **by**(*rule bind-spmf-mono*) *simp-all*

lemma *mcont-map-spmf* [*cont-intro*]:
mcont luba orda lub-spmf (*ord-spmf* (=)) *g*
⟹ *mcont luba orda lub-spmf* (*ord-spmf* (=)) (λ*x*. *map-spmf f* (*g x*))
unfolding *map-spmf-conv-bind-spmf* **by**(*rule mcont-bind-spmf*) *simp-all*

lemma *monotone-set-spmf*: *monotone* (*ord-spmf* (=)) (⊆) *set-spmf*
by(*rule monotoneI*)(*rule ord-spmf-eqD-set-spmf*)

lemma *cont-set-spmf*: *cont lub-spmf* (*ord-spmf* (=)) *Union* (⊆) *set-spmf*
by(*rule contI*)(*subst set-lub-spmf*; *simp*)

lemma *mcont2mcont-set-spmf*[*THEN mcont2mcont*, *cont-intro*]:
shows *mcont-set-spmf*: *mcont lub-spmf* (*ord-spmf* (=)) *Union* (⊆) *set-spmf*
by(*rule mcontI monotone-set-spmf cont-set-spmf*)+

lemma *monotone-spmf*: *monotone* (*ord-spmf* (=)) (≤) (λ*p*. *spmf p x*)
by(*rule monotoneI*)(*simp add*: *ord-spmf-eq-leD*)

lemma *cont-spmf*: *cont lub-spmf* (*ord-spmf* (=)) *Sup* (≤) (λ*p*. *spmf p x*)
by(*rule contI*)(*simp add*: *spmf-lub-spmf*)

lemma *mcont-spmf*: *mcont lub-spmf* (*ord-spmf* (=)) *Sup* (≤) (λ*p*. *spmf p x*)
by(*rule mcontI monotone-spmf cont-spmf*)+

lemma *cont-ennreal-spmf*: *cont lub-spmf* (*ord-spmf* (=)) *Sup* (≤) (λ*p*. *ennreal* (*spmf p x*))
by(*rule contI*)(*simp add*: *ennreal-spmf-lub-spmf*)

lemma *mcont2mcont-ennreal-spmf* [*THEN mcont2mcont*, *cont-intro*]:

shows *mcont-ennreal-spmf*: *mcont lub-spmf* (*ord-spmf* (=)) *Sup* (≤) (λ*p*. *ennreal* (*spmf p x*))
by(*rule mcontI mono2mono-ennreal monotone-spmf cont-ennreal-spmf*)+

lemma *nn-integral-map-spmf* [*simp*]: *nn-integral* (*measure-spmf* (*map-spmf f p*)) *g* = *nn-integral* (*measure-spmf p*) (*g* ∘ *f*)
by(*auto 4 3 simp add*: *measure-spmf-def nn-integral-distr nn-integral-restrict-space intro*: *nn-integral-cong split*: *split-indicator*)

25.11.1 Admissibility of *rel-spmf*

lemma *rel-spmf-measureD*:
assumes *rel-spmf R p q*
shows *measure* (*measure-spmf p*) *A* ≤ *measure* (*measure-spmf q*) {*y*. ∃ *x*∈*A*. *R x y*} (**is** *?lhs* ≤ *?rhs*)
proof −
have *?lhs* = *measure* (*measure-pmf p*) (*Some* ‘ *A*) **by**(*simp add*: *measure-measure-spmf-conv-measure-pmf*)
also have ... ≤ *measure* (*measure-pmf q*) {*y*. ∃ *x*∈*Some* ‘ *A*. *rel-option R x y*}
using *assms* **by**(*rule rel-pmf-measureD*)
also have ... = *?rhs* **unfolding** *measure-measure-spmf-conv-measure-pmf*
by(*rule arg-cong2*[**where** *f*=*measure*])(*auto simp add*: *option-rel-Some1*)
finally show *?thesis* **.**
qed

locale *rel-spmf-characterisation* =
assumes *rel-pmf-measureI*:
⋀(*R* :: *'a option* ⇒ *'b option* ⇒ *bool*) *p q*.
(⋀*A*. *measure* (*measure-pmf p*) *A* ≤ *measure* (*measure-pmf q*) {*y*. ∃ *x*∈*A*. *R x y*})
⟹ *rel-pmf R p q*
— This assumption is shown to hold in general in the AFP entry *MFMC-Countable*.
begin

context fixes *R* :: *'a* ⇒ *'b* ⇒ *bool* **begin**

lemma *rel-spmf-measureI*:
assumes *eq1*: ⋀*A*. *measure* (*measure-spmf p*) *A* ≤ *measure* (*measure-spmf q*) {*y*. ∃ *x*∈*A*. *R x y*}
assumes *eq2*: *weight-spmf q* ≤ *weight-spmf p*
shows *rel-spmf R p q*
proof(*rule rel-pmf-measureI*)
fix *A* :: *'a option set*
define *A'* **where** *A'* = *the* ‘ (*A* ∩ *range Some*)
define *A''* **where** *A''* = *A* ∩ {*None*}
have *A*: *A* = *Some* ‘ *A'* ∪ *A''* *Some* ‘ *A'* ∩ *A''* = {}
unfolding *A'-def A''-def* **by**(*auto 4 3 intro*: *rev-image-eqI*)
have *measure* (*measure-pmf p*) *A* = *measure* (*measure-pmf p*) (*Some* ‘ *A'*) + *measure* (*measure-pmf p*) *A''*
by(*simp add*: *A measure-pmf.finite-measure-Union*)

```
  also have measure (measure-pmf p) (Some ` A') = measure (measure-spmf p) A'
    by(simp add: measure-measure-spmf-conv-measure-pmf)
  also have ... ≤ measure (measure-spmf q) {y. ∃x∈A'. R x y} by(rule eq1)
  also (ord-eq-le-trans[OF - add-right-mono])
  have ... = measure (measure-pmf q) {y. ∃x∈A'. rel-option R (Some x) y}
    unfolding measure-measure-spmf-conv-measure-pmf
   by(rule arg-cong2[where f=measure])(auto simp add: A'-def option-rel-Some1)
  also
  { have weight-spmf p ≤ measure (measure-spmf q) {y. ∃x. R x y}
      using eq1[of UNIV] unfolding weight-spmf-def by simp
    also have ... ≤ weight-spmf q unfolding weight-spmf-def
      by(rule measure-spmf.finite-measure-mono) simp-all
    finally have weight-spmf p = weight-spmf q using eq2 by simp }
  then have measure (measure-pmf p) A'' = measure (measure-pmf q) (if None ∈
A then {None} else {})
    unfolding A''-def by(simp add: pmf-None-eq-weight-spmf measure-pmf-single)
  also have measure (measure-pmf q) {y. ∃x∈A'. rel-option R (Some x) y} + ...
= measure (measure-pmf q) {y. ∃x∈A. rel-option R x y}
    by(subst measure-pmf.finite-measure-Union[symmetric])
      (auto 4 3 intro!: arg-cong2[where f=measure] simp add: option-rel-Some1
option-rel-Some2 A'-def intro: rev-bexI elim: option.rel-cases)
  finally show measure (measure-pmf p) A ≤ ... .
qed

lemma admissible-rel-spmf:
  ccpo.admissible (prod-lub lub-spmf lub-spmf) (rel-prod (ord-spmf (=)) (ord-spmf
(=))) (case-prod (rel-spmf R))
  (is ccpo.admissible ?lub ?ord ?P)
proof(rule ccpo.admissibleI)
  fix Y
  assume chain: Complete-Partial-Order.chain ?ord Y
    and Y: Y ≠ {}
    and R: ∀(p, q) ∈ Y. rel-spmf R p q
  from R have R: ⋀p q. (p, q) ∈ Y ⟹ rel-spmf R p q by auto
  have chain1: Complete-Partial-Order.chain (ord-spmf (=)) (fst ` Y)
    and chain2: Complete-Partial-Order.chain (ord-spmf (=)) (snd ` Y)
    using chain by(rule chain-imageI; clarsimp)+
  from Y have Y1: fst ` Y ≠ {} and Y2: snd ` Y ≠ {} by auto

  have rel-spmf R (lub-spmf (fst ` Y)) (lub-spmf (snd ` Y))
  proof(rule rel-spmf-measureI)
    show weight-spmf (lub-spmf (snd ` Y)) ≤ weight-spmf (lub-spmf (fst ` Y))
      by(auto simp add: weight-lub-spmf chain1 chain2 Y rel-spmf-weightD[OF R,
symmetric] intro!: cSUP-least intro: cSUP-upper2[OF bdd-aboveI2[OF weight-spmf-le-1]])

    fix A
   have measure (measure-spmf (lub-spmf (fst ` Y))) A = (SUP y∈fst ` Y. measure
(measure-spmf y) A)
      using chain1 Y1 by(rule measure-lub-spmf)
```

also have ... ≤ (*SUP y∈snd ‘ Y*. *measure* (*measure-spmf y*) {*y*. ∃ *x*∈*A*. *R x y*}) **using** *Y1*
by(*rule cSUP-least*)(*auto intro*!: *cSUP-upper2*[*OF bdd-aboveI2*[*OF measure-spmf.subprob-measure-le-1*]] *rel-spmf-measureD R*)
also have ... = *measure* (*measure-spmf* (*lub-spmf* (*snd ‘ Y*))) {*y*. ∃ *x*∈*A*. *R x y*}
using *chain2 Y2* **by**(*rule measure-lub-spmf*[*symmetric*])
finally show *measure* (*measure-spmf* (*lub-spmf* (*fst ‘ Y*))) *A* ≤ ... .
qed
then show *?P* (*?lub Y*) **by**(*simp add*: *prod-lub-def*)
qed

lemma *admissible-rel-spmf-mcont* [*cont-intro*]:
⟦ *mcont lub ord lub-spmf* (*ord-spmf* (=)) *f*; *mcont lub ord lub-spmf* (*ord-spmf* (=)) *g* ⟧
⟹ *ccpo.admissible lub ord* (λ*x*. *rel-spmf R* (*f x*) (*g x*))
by(*rule admissible-subst*[*OF admissible-rel-spmf*, **where** *f*=λ*x*. (*f x*, *g x*), *simplified*])(*rule mcont-Pair*)

context includes *lifting-syntax*
begin

lemma *fixp-spmf-parametric'*:
assumes *f*: ⋀*x*. *monotone* (*ord-spmf* (=)) (*ord-spmf* (=)) *F*
and *g*: ⋀*x*. *monotone* (*ord-spmf* (=)) (*ord-spmf* (=)) *G*
and *param*: (*rel-spmf R* ===> *rel-spmf R*) *F G*
shows (*rel-spmf R*) (*ccpo.fixp lub-spmf* (*ord-spmf* (=)) *F*) (*ccpo.fixp lub-spmf* (*ord-spmf* (=)) *G*)
by(*rule parallel-fixp-induct*[*OF ccpo-spmf ccpo-spmf - f g*])(*auto intro*: *param*[*THEN rel-funD*])

lemma *fixp-spmf-parametric*:
assumes *f*: ⋀*x*. *mono-spmf* (λ*f*. *F f x*)
and *g*: ⋀*x*. *mono-spmf* (λ*f*. *G f x*)
and *param*: ((*A* ===> *rel-spmf R*) ===> *A* ===> *rel-spmf R*) *F G*
shows (*A* ===> *rel-spmf R*) (*spmf.fixp-fun F*) (*spmf.fixp-fun G*)
using *f g*
proof(*rule parallel-fixp-induct-1-1*[*OF partial-function-definitions-spmf partial-function-definitions-spmf - - reflexive reflexive*, **where** *P*=(*A* ===> *rel-spmf R*)])
show *ccpo.admissible* (*prod-lub* (*fun-lub lub-spmf*) (*fun-lub lub-spmf*)) (*rel-prod* (*fun-ord* (*ord-spmf* (=))) (*fun-ord* (*ord-spmf* (=)))) (λ*x*. (*A* ===> *rel-spmf R*) (*fst x*) (*snd x*))
unfolding *rel-fun-def*
apply(*rule admissible-all admissible-imp admissible-rel-spmf-mcont*)+
apply(*rule spmf.mcont2mcont*[*OF mcont-call*])
apply(*rule mcont-fst*)
apply(*rule spmf.mcont2mcont*[*OF mcont-call*])
apply(*rule mcont-snd*)
done

show $(A ===> rel\text{-}spmf\ R)\ (\lambda\text{-}.\ lub\text{-}spmf\ \{\})\ (\lambda\text{-}.\ lub\text{-}spmf\ \{\})$ **by** *auto*
show $(A ===> rel\text{-}spmf\ R)\ (F\ f)\ (G\ g)$ **if** $(A ===> rel\text{-}spmf\ R)\ f\ g$ **for** $f\ g$
using *that* **by**(*rule rel-funD*[*OF param*])
qed

end

end

end

25.12 Restrictions on spmfs

definition *restrict-spmf* :: $'a\ spmf \Rightarrow 'a\ set \Rightarrow 'a\ spmf$ (**infixl** $\upharpoonright$ *110*)
where $p \upharpoonright A = map\text{-}pmf\ (\lambda x.\ x \gg= (\lambda y.\ if\ y \in A\ then\ Some\ y\ else\ None))\ p$

lemma *set-restrict-spmf* [*simp*]: $set\text{-}spmf\ (p \upharpoonright A) = set\text{-}spmf\ p \cap A$
by(*fastforce simp add*: *restrict-spmf-def set-spmf-def split*: *bind-splits if-split-asm*)

lemma *restrict-map-spmf*: $map\text{-}spmf\ f\ p \upharpoonright A = map\text{-}spmf\ f\ (p \upharpoonright (f\ -`\ A))$
by(*simp add*: *restrict-spmf-def pmf.map-comp o-def map-option-bind bind-map-option if-distrib cong del*: *if-weak-cong*)

lemma *restrict-restrict-spmf* [*simp*]: $p \upharpoonright A \upharpoonright B = p \upharpoonright (A \cap B)$
by(*auto simp add*: *restrict-spmf-def pmf.map-comp o-def intro*!: *pmf.map-cong bind-option-cong*)

lemma *restrict-spmf-empty* [*simp*]: $p \upharpoonright \{\} = return\text{-}pmf\ None$
by(*simp add*: *restrict-spmf-def*)

lemma *restrict-spmf-UNIV* [*simp*]: $p \upharpoonright UNIV = p$
by(*simp add*: *restrict-spmf-def*)

lemma *spmf-restrict-spmf-outside* [*simp*]: $x \notin A \Longrightarrow spmf\ (p \upharpoonright A)\ x = 0$
by(*simp add*: *spmf-eq-0-set-spmf*)

lemma *emeasure-restrict-spmf* [*simp*]:
$emeasure\ (measure\text{-}spmf\ (p \upharpoonright A))\ X = emeasure\ (measure\text{-}spmf\ p)\ (X \cap A)$
by(*auto simp add*: *restrict-spmf-def measure-spmf-def emeasure-distr measurable-restrict-space1 emeasure-restrict-space space-restrict-space intro*: *arg-cong2*[**where** *f*=*emeasure*] *split*: *bind-splits if-split-asm*)

lemma *measure-restrict-spmf* [*simp*]:
$measure\ (measure\text{-}spmf\ (p \upharpoonright A))\ X = measure\ (measure\text{-}spmf\ p)\ (X \cap A)$
using *emeasure-restrict-spmf*[*of p A X*]
by(*simp only*: *measure-spmf.emeasure-eq-measure ennreal-inj measure-nonneg*)

lemma *spmf-restrict-spmf*: $spmf\ (p \upharpoonright A)\ x = (if\ x \in A\ then\ spmf\ p\ x\ else\ 0)$
by(*simp add*: *spmf-conv-measure-spmf*)

lemma *spmf-restrict-spmf-inside* [*simp*]: $x \in A \Longrightarrow spmf\ (p \upharpoonright A)\ x = spmf\ p\ x$
by(*simp add*: *spmf-restrict-spmf*)

lemma *pmf-restrict-spmf-None*: $pmf\ (p \upharpoonright A)\ None = pmf\ p\ None + measure\ (measure\text{-}spmf\ p)\ (-A)$
proof −
have [*simp*]: $None \notin Some\ `\ (-A)$ **by** *auto*
have $(\lambda x.\ x \gg= (\lambda y.\ if\ y \in A\ then\ Some\ y\ else\ None)) -` \{None\} = \{None\} \cup (Some\ `\ (-A))$
by(*auto split*: *bind-splits if-split-asm*)
then show *?thesis* **unfolding** *ereal.inject*[*symmetric*]
by(*simp add*: *restrict-spmf-def ennreal-pmf-map emeasure-pmf-single del*: *ereal.inject*)
(*simp add*: *pmf.rep-eq measure-pmf.finite-measure-Union*[*symmetric*] *measure-measure-spmf-conv-measure-pmf measure-pmf.emeasure-eq-measure*)
qed

lemma *restrict-spmf-trivial*: $(\bigwedge x.\ x \in set\text{-}spmf\ p \Longrightarrow x \in A) \Longrightarrow p \upharpoonright A = p$
by(*rule spmf-eqI*)(*auto simp add*: *spmf-restrict-spmf spmf-eq-0-set-spmf*)

lemma *restrict-spmf-trivial'*: $set\text{-}spmf\ p \subseteq A \Longrightarrow p \upharpoonright A = p$
by(*rule restrict-spmf-trivial*) *blast*

lemma *restrict-return-spmf*: $return\text{-}spmf\ x \upharpoonright A = (if\ x \in A\ then\ return\text{-}spmf\ x\ else\ return\text{-}pmf\ None)$
by(*simp add*: *restrict-spmf-def*)

lemma *restrict-return-spmf-inside* [*simp*]: $x \in A \Longrightarrow return\text{-}spmf\ x \upharpoonright A = return\text{-}spmf\ x$
by(*simp add*: *restrict-return-spmf*)

lemma *restrict-return-spmf-outside* [*simp*]: $x \notin A \Longrightarrow return\text{-}spmf\ x \upharpoonright A = return\text{-}pmf\ None$
by(*simp add*: *restrict-return-spmf*)

lemma *restrict-spmf-return-pmf-None* [*simp*]: $return\text{-}pmf\ None \upharpoonright A = return\text{-}pmf\ None$
by(*simp add*: *restrict-spmf-def*)

lemma *restrict-bind-pmf*: $bind\text{-}pmf\ p\ g \upharpoonright A = p \gg= (\lambda x.\ g\ x \upharpoonright A)$
by(*simp add*: *restrict-spmf-def map-bind-pmf o-def*)

lemma *restrict-bind-spmf*: $bind\text{-}spmf\ p\ g \upharpoonright A = p \gg= (\lambda x.\ g\ x \upharpoonright A)$
by(*auto simp add*: *bind-spmf-def restrict-bind-pmf cong del*: *option.case-cong-weak cong*: *option.case-cong intro*!: *bind-pmf-cong split*: *option.split*)

lemma *bind-restrict-pmf*: $bind\text{-}pmf\ (p \upharpoonright A)\ g = p \gg= (\lambda x.\ if\ x \in Some\ `\ A\ then\ g\ x\ else\ g\ None)$
by(*auto simp add*: *restrict-spmf-def bind-map-pmf fun-eq-iff split*: *bind-split intro*: *arg-cong2*[**where** *f*=*bind-pmf*])

lemma *bind-restrict-spmf*: *bind-spmf* (*p* ↾ *A*) *g* = *p* ⤜ (λ*x*. *if x* ∈ *A then g x else return-pmf None*)
by(*auto simp add*: *bind-spmf-def bind-restrict-pmf fun-eq-iff intro*: *arg-cong2*[**where** *f*=*bind-pmf*] *split*: *option.split*)

lemma *spmf-map-restrict*: *spmf* (*map-spmf fst* (*p* ↾ (*snd* −‘ {*y*}))) *x* = *spmf p* (*x*, *y*)
by(*subst spmf-map*)(*auto intro*: *arg-cong2*[**where** *f*=*measure*] *simp add*: *spmf-conv-measure-spmf*)

lemma *measure-eqI-restrict-spmf*:
assumes *rel-spmf R* (*restrict-spmf p A*) (*restrict-spmf q B*)
shows *measure* (*measure-spmf p*) *A* = *measure* (*measure-spmf q*) *B*
proof −
from *assms* **have** *weight-spmf* (*restrict-spmf p A*) = *weight-spmf* (*restrict-spmf q B*) **by**(*rule rel-spmf-weightD*)
thus *?thesis* **by**(*simp add*: *weight-spmf-def*)
qed

25.13 Subprobability distributions of sets

definition *spmf-of-set* :: *'a set* ⇒ *'a spmf*
where
spmf-of-set A = (*if finite A* ∧ *A* ≠ {} *then spmf-of-pmf* (*pmf-of-set A*) *else return-pmf None*)

lemma *spmf-of-set*: *spmf* (*spmf-of-set A*) *x* = *indicator A x* / *card A*
by(*auto simp add*: *spmf-of-set-def*)

lemma *pmf-spmf-of-set-None* [*simp*]: *pmf* (*spmf-of-set A*) *None* = *indicator* {*A*. *infinite A* ∨ *A* = {}} *A*
by(*simp add*: *spmf-of-set-def*)

lemma *set-spmf-of-set*: *set-spmf* (*spmf-of-set A*) = (*if finite A then A else* {})
by(*simp add*: *spmf-of-set-def*)

lemma *set-spmf-of-set-finite* [*simp*]: *finite A* ⟹ *set-spmf* (*spmf-of-set A*) = *A*
by(*simp add*: *set-spmf-of-set*)

lemma *spmf-of-set-singleton*: *spmf-of-set* {*x*} = *return-spmf x*
by(*simp add*: *spmf-of-set-def pmf-of-set-singleton*)

lemma *map-spmf-of-set-inj-on* [*simp*]:
inj-on f A ⟹ *map-spmf f* (*spmf-of-set A*) = *spmf-of-set* (*f* ‘ *A*)
by(*auto simp add*: *spmf-of-set-def map-pmf-of-set-inj dest*: *finite-imageD*)

lemma *spmf-of-pmf-pmf-of-set* [*simp*]:
⟦ *finite A*; *A* ≠ {} ⟧ ⟹ *spmf-of-pmf* (*pmf-of-set A*) = *spmf-of-set A*
by(*simp add*: *spmf-of-set-def*)

lemma *weight-spmf-of-set*:
weight-spmf (*spmf-of-set A*) = (*if finite A* ∧ *A* ≠ {} *then 1 else 0*)
by(*auto simp only*: *spmf-of-set-def weight-spmf-of-pmf weight-return-pmf-None split*: *if-split*)

lemma *weight-spmf-of-set-finite* [*simp*]: ⟦ *finite A*; *A* ≠ {} ⟧ ⟹ *weight-spmf* (*spmf-of-set A*) = *1*
by(*simp add*: *weight-spmf-of-set*)

lemma *weight-spmf-of-set-infinite* [*simp*]: *infinite A* ⟹ *weight-spmf* (*spmf-of-set A*) = *0*
by(*simp add*: *weight-spmf-of-set*)

lemma *measure-spmf-spmf-of-set*:
measure-spmf (*spmf-of-set A*) = (*if finite A* ∧ *A* ≠ {} *then measure-pmf* (*pmf-of-set A*) *else null-measure* (*count-space UNIV*))
by(*simp add*: *spmf-of-set-def del*: *spmf-of-pmf-pmf-of-set*)

lemma *emeasure-spmf-of-set*:
emeasure (*measure-spmf* (*spmf-of-set S*)) *A* = *card* (*S* ∩ *A*) / *card S*
by(*auto simp add*: *measure-spmf-spmf-of-set emeasure-pmf-of-set*)

lemma *measure-spmf-of-set*:
measure (*measure-spmf* (*spmf-of-set S*)) *A* = *card* (*S* ∩ *A*) / *card S*
by(*auto simp add*: *measure-spmf-spmf-of-set measure-pmf-of-set*)

lemma *nn-integral-spmf-of-set*: *nn-integral* (*measure-spmf* (*spmf-of-set A*)) *f* = *sum f A* / *card A*
by(*cases finite A*)(*auto simp add*: *spmf-of-set-def nn-integral-pmf-of-set card-gt-0-iff simp del*: *spmf-of-pmf-pmf-of-set*)

lemma *integral-spmf-of-set*: *integral*^{L} (*measure-spmf* (*spmf-of-set A*)) *f* = *sum f A* / *card A*
by(*clarsimp simp add*: *spmf-of-set-def integral-pmf-of-set card-gt-0-iff simp del*: *spmf-of-pmf-pmf-of-set*)

notepad begin — *pmf-of-set* is not fully parametric.
define *R* :: *nat* ⇒ *nat* ⇒ *bool* **where** *R x y* ⟷ (*x* ≠ *0* ⟶ *y* = *0*) **for** *x y*
define *A* :: *nat set* **where** *A* = {*0*, *1*}
define *B* :: *nat set* **where** *B* = {*0*, *1*, *2*}
have *rel-set R A B* **unfolding** *R-def*[*abs-def*] *A-def B-def rel-set-def* **by** *auto*
have ¬ *rel-pmf R* (*pmf-of-set A*) (*pmf-of-set B*)
proof
assume *rel-pmf R* (*pmf-of-set A*) (*pmf-of-set B*)
then obtain *pq* **where** *pq*: ⋀*x y*. (*x*, *y*) ∈ *set-pmf pq* ⟹ *R x y*
and *1*: *map-pmf fst pq* = *pmf-of-set A*
and *2*: *map-pmf snd pq* = *pmf-of-set B*
by *cases auto*

```
    have pmf (pmf-of-set B) 1 = 1 / 3 by(simp add: B-def)
    have pmf (pmf-of-set B) 2 = 1 / 3 by(simp add: B-def)

     have 2 / 3 = pmf (pmf-of-set B) 1 + pmf (pmf-of-set B) 2 by(simp add:
B-def)
    also have ... = measure (measure-pmf (pmf-of-set B)) ({1} ∪ {2})
     by(subst measure-pmf.finite-measure-Union)(simp-all add: measure-pmf-single)
    also have ... = emeasure (measure-pmf pq) (snd -` {2, 1})
     unfolding 2[symmetric] measure-pmf.emeasure-eq-measure[symmetric] by(simp)
    also have ... = emeasure (measure-pmf pq) {(0, 2), (0, 1)}
      by(rule emeasure-eq-AE)(auto simp add: AE-measure-pmf-iff R-def dest!: pq)
    also have ... ≤ emeasure (measure-pmf pq) (fst -` {0})
      by(rule emeasure-mono) auto
    also have ... = emeasure (measure-pmf (pmf-of-set A)) {0}
      unfolding 1[symmetric] by simp
    also have ... = pmf (pmf-of-set A) 0
      by(simp add: measure-pmf-single measure-pmf.emeasure-eq-measure)
    also have pmf (pmf-of-set A) 0 = 1 / 2 by(simp add: A-def)
    finally show False by(subst (asm) ennreal-le-iff; simp)
  qed
end

lemma rel-pmf-of-set-bij:
  assumes f: bij-betw f A B
  and A: A ≠ {} finite A
  and B: B ≠ {} finite B
  and R: ⋀x. x ∈ A ⟹ R x (f x)
  shows rel-pmf R (pmf-of-set A) (pmf-of-set B)
proof(rule pmf.rel-mono-strong)
  define AB where AB = (λx. (x, f x)) ` A
  define R' where R' x y ⟷ (x, y) ∈ AB for x y
  have (x, y) ∈ AB if (x, y) ∈ set-pmf (pmf-of-set AB) for x y
    using that by(auto simp add: AB-def A)
  moreover have map-pmf fst (pmf-of-set AB) = pmf-of-set A
   by(simp add: AB-def map-pmf-of-set-inj[symmetric] inj-on-def A pmf.map-comp
o-def)
  moreover
  from f have [simp]: inj-on f A by(rule bij-betw-imp-inj-on)
  from f have [simp]: f ` A = B by(rule bij-betw-imp-surj-on)
  have map-pmf snd (pmf-of-set AB) = pmf-of-set B
   by(simp add: AB-def map-pmf-of-set-inj[symmetric] inj-on-def A pmf.map-comp
o-def)
      (simp add: map-pmf-of-set-inj A)
  ultimately show rel-pmf (λx y. (x, y) ∈ AB) (pmf-of-set A) (pmf-of-set B) ..
qed(auto intro: R)

lemma rel-spmf-of-set-bij:
  assumes f: bij-betw f A B
  and R: ⋀x. x ∈ A ⟹ R x (f x)
```

shows *rel-spmf R* (*spmf-of-set A*) (*spmf-of-set B*)
proof −
have *finite A* ⟷ *finite B* **using** *f* **by**(*rule bij-betw-finite*)
moreover have *A* = {} ⟷ *B* = {} **using** *f* **by**(*auto dest*: *bij-betw-empty2 bij-betw-empty1*)
ultimately show *?thesis* **using** *assms*
by(*auto simp add*: *spmf-of-set-def simp del*: *spmf-of-pmf-pmf-of-set intro*: *rel-pmf-of-set-bij*)
qed

context includes *lifting-syntax*
begin

lemma *rel-spmf-of-set*:
assumes *bi-unique R*
shows (*rel-set R* ===> *rel-spmf R*) *spmf-of-set spmf-of-set*
proof
fix *A B*
assume *R*: *rel-set R A B*
with *assms* **obtain** *f* **where** *bij-betw f A B* **and** *f*: ⋀*x*. *x* ∈ *A* ⟹ *R x* (*f x*)
by(*auto dest*: *bi-unique-rel-set-bij-betw*)
then show *rel-spmf R* (*spmf-of-set A*) (*spmf-of-set B*) **by**(*rule rel-spmf-of-set-bij*)
qed

end

lemma *map-mem-spmf-of-set*:
assumes *finite B B* ≠ {}
shows *map-spmf* (λ*x*. *x* ∈ *A*) (*spmf-of-set B*) = *spmf-of-pmf* (*bernoulli-pmf* (*card* (*A* ∩ *B*) / *card B*))
(**is** *?lhs* = *?rhs*)
proof(*rule spmf-eqI*)
fix *i*
have *ennreal* (*spmf ?lhs i*) = *card* (*B* ∩ (λ*x*. *x* ∈ *A*) −‘ {*i*}) / (*card B*)
by(*subst ennreal-spmf-map*)(*simp add*: *measure-spmf-spmf-of-set assms emeasure-pmf-of-set*)
also have ... = (*if i then card* (*B* ∩ *A*) / *card B else card* (*B* − *A*) / *card B*)
by(*auto intro*: *arg-cong*[**where** *f*=*card*])
also have ... = (*if i then card* (*B* ∩ *A*) / *card B else* (*card B* − *card* (*B* ∩ *A*)) / *card B*)
by(*auto simp add*: *card-Diff-subset-Int assms*)
also have ... = *ennreal* (*spmf ?rhs i*)
by(*simp add*: *assms card-gt-0-iff field-simps card-mono Int-commute of-nat-diff*)
finally show *spmf ?lhs i* = *spmf ?rhs i* **by** *simp*
qed

abbreviation *coin-spmf* :: *bool spmf*
where *coin-spmf* ≡ *spmf-of-set UNIV*

lemma *map-eq-const-coin-spmf*: *map-spmf* ((=) *c*) *coin-spmf* = *coin-spmf*

proof −
have *inj* ((⟷) *c*) *range* ((⟷) *c*) = *UNIV* **by**(*auto intro*: *inj-onI*)
then show *?thesis* **by** *simp*
qed

lemma *bind-coin-spmf-eq-const*: *coin-spmf* ⤜ (λ*x* :: *bool*. *return-spmf* (*b* = *x*))
= *coin-spmf*
using *map-eq-const-coin-spmf* **unfolding** *map-spmf-conv-bind-spmf* **by** *simp*

lemma *bind-coin-spmf-eq-const′*: *coin-spmf* ⤜ (λ*x* :: *bool*. *return-spmf* (*x* = *b*))
= *coin-spmf*
by(*rewrite* **in** - = ⌑ *bind-coin-spmf-eq-const*[*symmetric*, *of b*])(*auto intro*: *bind-spmf-cong*)

25.14 Losslessness

definition *lossless-spmf* :: *′a spmf* ⇒ *bool*
where *lossless-spmf p* ⟷ *weight-spmf p* = *1*

lemma *lossless-iff-pmf-None*: *lossless-spmf p* ⟷ *pmf p None* = *0*
by(*simp add*: *lossless-spmf-def pmf-None-eq-weight-spmf*)

lemma *lossless-return-spmf* [*iff*]: *lossless-spmf* (*return-spmf x*)
by(*simp add*: *lossless-iff-pmf-None*)

lemma *lossless-return-pmf-None* [*iff*]: ¬ *lossless-spmf* (*return-pmf None*)
by(*simp add*: *lossless-iff-pmf-None*)

lemma *lossless-map-spmf* [*simp*]: *lossless-spmf* (*map-spmf f p*) ⟷ *lossless-spmf*
p
by(*auto simp add*: *lossless-iff-pmf-None pmf-eq-0-set-pmf*)

lemma *lossless-bind-spmf* [*simp*]:
lossless-spmf (*p* ⤜ *f*) ⟷ *lossless-spmf p* ∧ (∀ *x*∈*set-spmf p*. *lossless-spmf* (*f*
x))
by(*simp add*: *lossless-iff-pmf-None pmf-bind-spmf-None add-nonneg-eq-0-iff integral-nonneg-AE integral-nonneg-eq-0-iff-AE measure-spmf.integrable-const-bound*[**where**
B=*1*] *pmf-le-1*)

lemma *lossless-weight-spmfD*: *lossless-spmf p* ⟹ *weight-spmf p* = *1*
by(*simp add*: *lossless-spmf-def*)

lemma *lossless-iff-set-pmf-None*:
lossless-spmf p ⟷ *None* ∉ *set-pmf p*
by (*simp add*: *lossless-iff-pmf-None pmf-eq-0-set-pmf*)

lemma *lossless-spmf-of-set* [*simp*]: *lossless-spmf* (*spmf-of-set A*) ⟷ *finite A* ∧ *A*
≠ {}
by(*auto simp add*: *lossless-spmf-def weight-spmf-of-set*)

lemma *lossless-spmf-spmf-of-spmf* [*simp*]: *lossless-spmf* (*spmf-of-pmf p*)
by(*simp add*: *lossless-spmf-def*)

lemma *lossless-spmf-bind-pmf* [*simp*]:
lossless-spmf (*bind-pmf p f*) ⟷ (∀ *x*∈*set-pmf p*. *lossless-spmf* (*f x*))
by(*simp add*: *lossless-iff-pmf-None pmf-bind integral-nonneg-AE integral-nonneg-eq-0-iff-AE measure-pmf.integrable-const-bound*[**where** *B=1*] *AE-measure-pmf-iff pmf-le-1*)

lemma *lossless-spmf-conv-spmf-of-pmf*: *lossless-spmf p* ⟷ (∃ *p'*. *p* = *spmf-of-pmf p'*)
proof
assume *lossless-spmf p*
hence ∗: ⋀*y*. *y* ∈ *set-pmf p* ⟹ ∃ *x*. *y* = *Some x*
by(*case-tac y*)(*simp-all add*: *lossless-iff-set-pmf-None*)

let *?p* = *map-pmf the p*
have *p* = *spmf-of-pmf ?p*
proof(*rule spmf-eqI*)
fix *i*
have *ennreal* (*pmf* (*map-pmf the p*) *i*) = ∫⁺ *x*. *indicator* (*the* −‘ {*i*}) *x* ∂*p*
by(*simp add*: *ennreal-pmf-map*)
also have ... = ∫⁺ *x*. *indicator* {*i*} *x* ∂*measure-spmf p* **unfolding** *measure-spmf-def*
by(*subst nn-integral-distr*)(*auto simp add*: *nn-integral-restrict-space AE-measure-pmf-iff simp del*: *nn-integral-indicator intro*!: *nn-integral-cong-AE split*: *split-indicator dest*!: ∗)
also have ... = *spmf p i* **by**(*simp add*: *emeasure-spmf-single*)
finally show *spmf p i* = *spmf* (*spmf-of-pmf ?p*) *i* **by** *simp*
qed
thus ∃ *p'*. *p* = *spmf-of-pmf p'* **..**
qed *auto*

lemma *spmf-False-conv-True*: *lossless-spmf p* ⟹ *spmf p False* = *1* − *spmf p True*
by(*clarsimp simp add*: *lossless-spmf-conv-spmf-of-pmf pmf-False-conv-True*)

lemma *spmf-True-conv-False*: *lossless-spmf p* ⟹ *spmf p True* = *1* − *spmf p False*
by(*simp add*: *spmf-False-conv-True*)

lemma *bind-eq-return-spmf*:
bind-spmf p f = *return-spmf x* ⟷ (∀ *y*∈*set-spmf p*. *f y* = *return-spmf x*) ∧ *lossless-spmf p*
by(*auto simp add*: *bind-spmf-def bind-eq-return-pmf in-set-spmf lossless-iff-pmf-None pmf-eq-0-set-pmf iff del*: *not-None-eq split*: *option.split*)

lemma *rel-spmf-return-spmf2*:
rel-spmf R p (*return-spmf x*) ⟷ *lossless-spmf p* ∧ (∀ *a*∈*set-spmf p*. *R a x*)
by(*auto simp add*: *lossless-iff-set-pmf-None rel-pmf-return-pmf2 option-rel-Some2 in-set-spmf*, *metis in-set-spmf not-None-eq*)

lemma *rel-spmf-return-spmf1*:
rel-spmf R (*return-spmf x*) *p* ⟷ *lossless-spmf p* ∧ (∀ *a*∈*set-spmf p*. *R x a*)
using *rel-spmf-return-spmf2*[*of* R^{-1-1}] **by**(*simp add*: *spmf-rel-conversep*)

lemma *rel-spmf-bindI1*:
assumes *f*: ⋀*x*. *x* ∈ *set-spmf p* ⟹ *rel-spmf R* (*f x*) *q*
and *p*: *lossless-spmf p*
shows *rel-spmf R* (*bind-spmf p f*) *q*
proof −
fix *x* :: *'a*
have *rel-spmf R* (*bind-spmf p f*) (*bind-spmf* (*return-spmf x*) (λ-. *q*))
by(*rule rel-spmf-bindI*[**where** *R*=λ*x* -. *x* ∈ *set-spmf p*])(*simp-all add*: *rel-spmf-return-spmf2 p f*)
then show *?thesis* **by** *simp*
qed

lemma *rel-spmf-bindI2*:
⟦ ⋀*x*. *x* ∈ *set-spmf q* ⟹ *rel-spmf R p* (*f x*); *lossless-spmf q* ⟧
⟹ *rel-spmf R p* (*bind-spmf q f*)
using *rel-spmf-bindI1*[*of q conversep R f p*] **by**(*simp add*: *spmf-rel-conversep*)

25.15 Scaling

definition *scale-spmf* :: *real* ⇒ *'a spmf* ⇒ *'a spmf*
where
scale-spmf r p = *embed-spmf* (λ*x*. *min* (*inverse* (*weight-spmf p*)) (*max 0 r*) ∗ *spmf p x*)

lemma *scale-spmf-le-1*:
(∫⁺ *x*. *min* (*inverse* (*weight-spmf p*)) (*max 0 r*) ∗ *spmf p x* ∂*count-space UNIV*) ≤ *1* (**is** *?lhs* ≤ -)
proof −
have *?lhs* = *min* (*inverse* (*weight-spmf p*)) (*max 0 r*) ∗ ∫⁺ *x*. *spmf p x* ∂*count-space UNIV*
by(*subst nn-integral-cmult*[*symmetric*])(*simp-all add*: *weight-spmf-nonneg max-def min-def ennreal-mult*)
also have ... ≤ *1* **unfolding** *weight-spmf-eq-nn-integral-spmf*[*symmetric*]
by(*simp add*: *min-def max-def weight-spmf-nonneg order.strict-iff-order field-simps ennreal-mult*[*symmetric*])
finally show *?thesis* **.**
qed

lemma *spmf-scale-spmf*: *spmf* (*scale-spmf r p*) *x* = *max 0* (*min* (*inverse* (*weight-spmf p*)) *r*) ∗ *spmf p x* (**is** *?lhs* = *?rhs*)
unfolding *scale-spmf-def*
apply(*subst spmf-embed-spmf*[*OF scale-spmf-le-1*])
apply(*simp add*: *max-def min-def weight-spmf-le-0 field-simps weight-spmf-nonneg not-le order.strict-iff-order*)
apply(*metis antisym-conv order-trans weight-spmf-nonneg zero-le-mult-iff zero-le-one*)

done

lemma *real-inverse-le-1-iff*: **fixes** $x :: real$
shows $[\![0 \leq x; x \leq 1]\!] \Longrightarrow 1 / x \leq 1 \longleftrightarrow x = 1 \vee x = 0$
by *auto*

lemma *spmf-scale-spmf'*: $r \leq 1 \Longrightarrow spmf\ (scale\text{-}spmf\ r\ p)\ x = max\ 0\ r * spmf\ p\ x$
using *real-inverse-le-1-iff*[*OF weight-spmf-nonneg weight-spmf-le-1*, *of p*]
by(*auto simp add*: *spmf-scale-spmf max-def min-def field-simps*)(*metis pmf-le-0-iff spmf-le-weight*)

lemma *scale-spmf-neg*: $r \leq 0 \Longrightarrow scale\text{-}spmf\ r\ p = return\text{-}pmf\ None$
by(*rule spmf-eqI*)(*simp add*: *spmf-scale-spmf' max-def*)

lemma *scale-spmf-return-None* [*simp*]: $scale\text{-}spmf\ r\ (return\text{-}pmf\ None) = return\text{-}pmf\ None$
by(*rule spmf-eqI*)(*simp add*: *spmf-scale-spmf*)

lemma *scale-spmf-conv-bind-bernoulli*:
assumes $r \leq 1$
shows $scale\text{-}spmf\ r\ p = bind\text{-}pmf\ (bernoulli\text{-}pmf\ r)\ (\lambda b.\ if\ b\ then\ p\ else\ return\text{-}pmf\ None)$ (**is** *?lhs = ?rhs*)
proof(*rule spmf-eqI*)
fix x
have $ennreal\ (spmf\ ?lhs\ x) = ennreal\ (spmf\ ?rhs\ x)$ **using** *assms*
unfolding *spmf-scale-spmf ennreal-pmf-bind nn-integral-measure-pmf UNIV-bool bernoulli-pmf.rep-eq*
apply(*auto simp add*: *nn-integral-count-space-finite max-def min-def field-simps real-inverse-le-1-iff*[*OF weight-spmf-nonneg weight-spmf-le-1*] *weight-spmf-lt-0 not-le ennreal-mult*[*symmetric*])
apply (*metis pmf-le-0-iff spmf-le-weight*)
apply (*metis pmf-le-0-iff spmf-le-weight*)
apply (*meson le-divide-eq-1-pos measure-spmf.subprob-measure-le-1 not-less order-trans weight-spmf-le-0*)
by (*meson divide-le-0-1-iff less-imp-le order-trans weight-spmf-le-0*)
thus *spmf ?lhs x = spmf ?rhs x* **by** *simp*
qed

lemma *nn-integral-spmf*: $(\int^+ x.\ spmf\ p\ x\ \partial count\text{-}space\ A) = emeasure\ (measure\text{-}spmf\ p)\ A$
apply(*simp add*: *measure-spmf-def emeasure-distr emeasure-restrict-space space-restrict-space nn-integral-pmf*[*symmetric*])
apply(*rule nn-integral-bij-count-space*[**where** *g*=*Some*])
apply(*auto simp add*: *bij-betw-def*)
done

lemma *measure-spmf-scale-spmf*: $measure\text{-}spmf\ (scale\text{-}spmf\ r\ p) = scale\text{-}measure\ (min\ (inverse\ (weight\text{-}spmf\ p))\ r)\ (measure\text{-}spmf\ p)$

apply(*rule measure-eqI*)
apply *simp*
apply(*simp add*: *nn-integral-spmf*[*symmetric*] *spmf-scale-spmf*)
apply(*subst nn-integral-cmult*[*symmetric*])
apply(*auto simp add*: *max-def min-def ennreal-mult*[*symmetric*] *not-le ennreal-lt-0*)
done

lemma *measure-spmf-scale-spmf'*:
$r \leq 1 \Longrightarrow$ *measure-spmf* (*scale-spmf r p*) = *scale-measure r* (*measure-spmf p*)
unfolding *measure-spmf-scale-spmf*
apply(*cases weight-spmf p* > *0*)
apply(*simp add*: *min.absorb2 field-simps weight-spmf-le-1 mult-le-one*)
apply(*clarsimp simp add*: *weight-spmf-le-0 min-def scale-spmf-neg weight-spmf-eq-0 not-less*)
done

lemma *scale-spmf-1* [*simp*]: *scale-spmf 1 p* = *p*
apply(*rule spmf-eqI*)
apply(*simp add*: *spmf-scale-spmf max-def min-def order.strict-iff-order field-simps weight-spmf-nonneg*)
apply(*metis antisym-conv divide-le-eq-1 less-imp-le pmf-nonneg spmf-le-weight weight-spmf-nonneg weight-spmf-le-1*)
done

lemma *scale-spmf-0* [*simp*]: *scale-spmf 0 p* = *return-pmf None*
by(*rule spmf-eqI*)(*simp add*: *spmf-scale-spmf min-def max-def weight-spmf-le-0*)

lemma *bind-scale-spmf*:
assumes *r*: $r \leq 1$
shows *bind-spmf* (*scale-spmf r p*) *f* = *bind-spmf p* (λ*x*. *scale-spmf r* (*f x*))
(**is** *?lhs* = *?rhs*)
proof(*rule spmf-eqI*)
fix *x*
have *ennreal* (*spmf ?lhs x*) = *ennreal* (*spmf ?rhs x*) **using** *r*
by(*simp add*: *ennreal-spmf-bind measure-spmf-scale-spmf' nn-integral-scale-measure spmf-scale-spmf'*)
(*simp add*: *ennreal-mult ennreal-lt-0 nn-integral-cmult max-def min-def*)
thus *spmf ?lhs x* = *spmf ?rhs x* **by** *simp*
qed

lemma *scale-bind-spmf*:
assumes $r \leq 1$
shows *scale-spmf r* (*bind-spmf p f*) = *bind-spmf p* (λ*x*. *scale-spmf r* (*f x*))
(**is** *?lhs* = *?rhs*)
proof(*rule spmf-eqI*)
fix *x*
have *ennreal* (*spmf ?lhs x*) = *ennreal* (*spmf ?rhs x*) **using** *assms*
unfolding *spmf-scale-spmf'*[*OF assms*]
by(*simp add*: *ennreal-mult ennreal-spmf-bind spmf-scale-spmf' nn-integral-cmult*

max-def min-def)
thus *spmf ?lhs x = spmf ?rhs x* **by** *simp*
qed

lemma *bind-spmf-const*: *bind-spmf p* (*λx. q*) = *scale-spmf* (*weight-spmf p*) *q* (**is** *?lhs = ?rhs*)
proof(*rule spmf-eqI*)
fix *x*
have *ennreal* (*spmf ?lhs x*) = *ennreal* (*spmf ?rhs x*)
using *measure-spmf.subprob-measure-le-1*[*of p space* (*measure-spmf p*)]
by(*subst ennreal-spmf-bind*)(*simp add*: *spmf-scale-spmf' weight-spmf-le-1 ennreal-mult mult.commute max-def min-def measure-spmf.emeasure-eq-measure*)
thus *spmf ?lhs x = spmf ?rhs x* **by** *simp*
qed

lemma *map-scale-spmf*: *map-spmf f* (*scale-spmf r p*) = *scale-spmf r* (*map-spmf f p*) (**is** *?lhs = ?rhs*)
proof(*rule spmf-eqI*)
fix *i*
show *spmf ?lhs i = spmf ?rhs i* **unfolding** *spmf-scale-spmf*
by(*subst* (*1 2*) *spmf-map*)(*auto simp add*: *measure-spmf-scale-spmf max-def min-def ennreal-lt-0*)
qed

lemma *set-scale-spmf*: *set-spmf* (*scale-spmf r p*) = (*if r > 0 then set-spmf p else* {})
apply(*auto simp add*: *in-set-spmf-iff-spmf spmf-scale-spmf*)
apply(*simp add*: *max-def min-def not-le weight-spmf-lt-0 weight-spmf-eq-0 split*: *if-split-asm*)
done

lemma *set-scale-spmf'* [*simp*]: *0 < r ⟹ set-spmf* (*scale-spmf r p*) = *set-spmf p*
by(*simp add*: *set-scale-spmf*)

lemma *rel-spmf-scaleI*:
assumes *r > 0 ⟹ rel-spmf A p q*
shows *rel-spmf A* (*scale-spmf r p*) (*scale-spmf r q*)
proof(*cases r > 0*)
case *True*
from *assms*[*OF this*] **show** *?thesis*
by(*rule rel-spmfE*)(*auto simp add*: *map-scale-spmf*[*symmetric*] *spmf-rel-map True intro*: *rel-spmf-reflI*)
qed(*simp add*: *not-less scale-spmf-neg*)

lemma *weight-scale-spmf*: *weight-spmf* (*scale-spmf r p*) = *min 1* (*max 0 r ∗ weight-spmf p*)
proof −
have *ennreal* (*weight-spmf* (*scale-spmf r p*)) = *min 1* (*max 0 r ∗ ennreal* (*weight-spmf p*))

unfolding *weight-spmf-eq-nn-integral-spmf*
apply(*simp add*: *spmf-scale-spmf ennreal-mult zero-ereal-def*[*symmetric*] *nn-integral-cmult*)
apply(*auto simp add*: *weight-spmf-eq-nn-integral-spmf*[*symmetric*] *field-simps min-def max-def not-le weight-spmf-lt-0 ennreal-mult*[*symmetric*])
subgoal by(*subst* (*asm*) *ennreal-mult*[*symmetric*], *meson divide-less-0-1-iff le-less-trans not-le weight-spmf-lt-0*, *simp+*, *meson not-le pos-divide-le-eq weight-spmf-le-0*)
subgoal by(*cases* $r \geq 0$)(*simp-all add*: *ennreal-mult*[*symmetric*] *weight-spmf-nonneg ennreal-lt-0*, *meson le-less-trans not-le pos-divide-le-eq zero-less-divide-1-iff*)
done
thus *?thesis* **by**(*auto simp add*: *min-def max-def ennreal-mult*[*symmetric*] *split*: *if-split-asm*)
qed

lemma *weight-scale-spmf'* [*simp*]:
$[\![0 \leq r; r \leq 1]\!] \Longrightarrow$ *weight-spmf* (*scale-spmf r p*) = $r *$ *weight-spmf p*
by(*simp add*: *weight-scale-spmf max-def min-def*)(*metis antisym-conv mult-left-le order-trans weight-spmf-le-1*)

lemma *pmf-scale-spmf-None*:
pmf (*scale-spmf k p*) *None* = 1 − *min* 1 (*max* 0 *k* ∗ (1 − *pmf p None*))
unfolding *pmf-None-eq-weight-spmf* **by**(*simp add*: *weight-scale-spmf*)

lemma *scale-scale-spmf*:
scale-spmf r (*scale-spmf r' p*) = *scale-spmf* (*r* ∗ *max* 0 (*min* (*inverse* (*weight-spmf p*)) *r'*)) *p*
(**is** *?lhs* = *?rhs*)
proof(*rule spmf-eqI*)
fix *i*
have *max* 0 (*min* (1 / *weight-spmf p*) *r'*) ∗
max 0 (*min* (1 / *min* 1 (*weight-spmf p* ∗ *max* 0 *r'*)) *r*) =
max 0 (*min* (1 / *weight-spmf p*) (*r* ∗ *max* 0 (*min* (1 / *weight-spmf p*) *r'*)))
proof(*cases weight-spmf p* > 0)
case *False*
thus *?thesis* **by**(*simp add*: *not-less weight-spmf-le-0*)
next
case *True*
thus *?thesis* **by**(*simp add*: *field-simps max-def min.absorb-iff2*[*symmetric*])(*auto simp add*: *min-def field-simps zero-le-mult-iff*)
qed
then show *spmf ?lhs i* = *spmf ?rhs i*
by(*simp add*: *spmf-scale-spmf field-simps weight-scale-spmf*)
qed

lemma *scale-scale-spmf'* [*simp*]:
$[\![0 \leq r; r \leq 1; 0 \leq r'; r' \leq 1]\!]$
$\Longrightarrow$ *scale-spmf r* (*scale-spmf r' p*) = *scale-spmf* (*r* ∗ *r'*) *p*
apply(*cases weight-spmf p* > 0)
apply(*auto simp add*: *scale-scale-spmf min-def max-def field-simps not-le weight-spmf-lt-0 weight-spmf-eq-0 not-less weight-spmf-le-0*)

apply(*subgoal-tac 1 = r'*)
apply (*metis* (*no-types*) *div-by-1 eq-iff measure-spmf.subprob-measure-le-1 mult.commute mult-cancel-right1*)
apply(*meson eq-iff le-divide-eq-1-pos measure-spmf.subprob-measure-le-1 mult-imp-div-pos-le order.trans*)
done

lemma *scale-spmf-eq-same*: *scale-spmf r p = p* ⟷ *weight-spmf p = 0* ∨ *r = 1* ∨ *r* ≥ *1* ∧ *weight-spmf p = 1*
(**is** *?lhs* ⟷ *?rhs*)
proof
assume *?lhs*
hence *weight-spmf* (*scale-spmf r p*) = *weight-spmf p* **by** *simp*
hence ∗: *min 1* (*max 0 r* ∗ *weight-spmf p*) = *weight-spmf p* **by**(*simp add*: *weight-scale-spmf*)
hence ∗∗: *weight-spmf p = 0* ∨ *r* ≥ *1* **by**(*auto simp add*: *min-def max-def split*: *if-split-asm*)
show *?rhs*
proof(*cases weight-spmf p = 0*)
case *False*
with ∗∗ **have** *r* ≥ *1* **by** *simp*
with ∗ *False* **have** *r = 1* ∨ *weight-spmf p = 1* **by**(*simp add*: *max-def min-def not-le split*: *if-split-asm*)
with ‹*r* ≥ *1*› **show** *?thesis* **by** *simp*
qed *simp*
qed(*auto intro*!: *spmf-eqI simp add*: *spmf-scale-spmf*, *metis pmf-le-0-iff spmf-le-weight*)

lemma *map-const-spmf-of-set*:
⟦ *finite A*; *A* ≠ {} ⟧ ⟹ *map-spmf* (λ-. *c*) (*spmf-of-set A*) = *return-spmf c*
by(*simp add*: *map-spmf-conv-bind-spmf bind-spmf-const*)

25.16 Conditional spmfs

lemma *set-pmf-Int-Some*: *set-pmf p* ∩ *Some* ‘ *A* = {} ⟷ *set-spmf p* ∩ *A* = {}
by(*auto simp add*: *in-set-spmf*)

lemma *measure-spmf-zero-iff*: *measure* (*measure-spmf p*) *A = 0* ⟷ *set-spmf p* ∩ *A* = {}
unfolding *measure-measure-spmf-conv-measure-pmf* **by**(*simp add*: *measure-pmf-zero-iff set-pmf-Int-Some*)

definition *cond-spmf* :: *'a spmf* ⇒ *'a set* ⇒ *'a spmf*
where *cond-spmf p A* = (*if set-spmf p* ∩ *A* = {} *then return-pmf None else cond-pmf p* (*Some* ‘ *A*))

lemma *set-cond-spmf* [*simp*]: *set-spmf* (*cond-spmf p A*) = *set-spmf p* ∩ *A*
by(*auto 4 4 simp add*: *cond-spmf-def in-set-spmf iff*: *set-cond-pmf*[*THEN set-eq-iff*[*THEN iffD1*], *THEN spec*, *rotated*])

lemma *cond-map-spmf* [*simp*]: *cond-spmf* (*map-spmf f p*) *A* = *map-spmf f* (*cond-spmf p* (*f* −‘ *A*))
proof −
have *map-option f* −‘ *Some* ‘ *A* = *Some* ‘ *f* −‘ *A* **by** *auto*
moreover have *set-pmf p* ∩ *map-option f* −‘ *Some* ‘ *A* ≠ {} **if** *Some x* ∈ *set-pmf p f x* ∈ *A* **for** *x*
using *that* **by** *auto*
ultimately show *?thesis* **by**(*auto simp add*: *cond-spmf-def in-set-spmf cond-map-pmf*)
qed

lemma *spmf-cond-spmf* [*simp*]:
spmf (*cond-spmf p A*) *x* = (*if x* ∈ *A then spmf p x* / *measure* (*measure-spmf p*) *A else 0*)
by(*auto simp add*: *cond-spmf-def pmf-cond set-pmf-Int-Some*[*symmetric*] *measure-measure-spmf-conv-measure-measure-pmf-zero-iff*)

lemma *bind-eq-return-pmf-None*:
bind-spmf p f = *return-pmf None* ⟷ (∀ *x*∈*set-spmf p*. *f x* = *return-pmf None*)
by(*auto simp add*: *bind-spmf-def bind-eq-return-pmf in-set-spmf split*: *option.splits*)

lemma *return-pmf-None-eq-bind*:
return-pmf None = *bind-spmf p f* ⟷ (∀ *x*∈*set-spmf p*. *f x* = *return-pmf None*)
using *bind-eq-return-pmf-None*[*of p f*] **by** *auto*

25.17 Product spmf

definition *pair-spmf* :: *'a spmf* ⇒ *'b spmf* ⇒ (*'a* × *'b*) *spmf*
where *pair-spmf p q* = *bind-pmf* (*pair-pmf p q*) (λ*xy*. *case xy of* (*Some x*, *Some y*) ⇒ *return-spmf* (*x*, *y*) | - ⇒ *return-pmf None*)

lemma *map-fst-pair-spmf* [*simp*]: *map-spmf fst* (*pair-spmf p q*) = *scale-spmf* (*weight-spmf q*) *p*
unfolding *bind-spmf-const*[*symmetric*]
apply(*simp add*: *pair-spmf-def map-bind-pmf pair-pmf-def bind-assoc-pmf option.case-distrib*)
apply(*subst bind-commute-pmf*)
apply(*auto intro*!: *bind-pmf-cong*[*OF refl*] *simp add*: *bind-return-pmf bind-spmf-def bind-return-pmf' case-option-collapse option.case-distrib*[**where** *h*=*map-spmf* -] *option.case-distrib*[*symmetric*] *case-option-id split*: *option.split cong del*: *option.case-cong-weak*)
done

lemma *map-snd-pair-spmf* [*simp*]: *map-spmf snd* (*pair-spmf p q*) = *scale-spmf* (*weight-spmf p*) *q*
unfolding *bind-spmf-const*[*symmetric*]
apply(*simp add*: *pair-spmf-def map-bind-pmf pair-pmf-def bind-assoc-pmf option.case-distrib cong del*: *option.case-cong-weak*)
apply(*auto intro*!: *bind-pmf-cong*[*OF refl*] *simp add*: *bind-return-pmf bind-spmf-def bind-return-pmf' case-option-collapse option.case-distrib*[**where** *h*=*map-spmf* -] *option.case-distrib*[*symmetric*] *case-option-id split*: *option.split cong del*: *option.case-cong-weak*)

done

lemma *set-pair-spmf* [*simp*]: *set-spmf* (*pair-spmf p q*) = *set-spmf p* × *set-spmf q*
by(*auto 4 3 simp add*: *pair-spmf-def set-spmf-bind-pmf bind-UNION in-set-spmf intro*: *rev-bexI split*: *option.splits*)

lemma *spmf-pair* [*simp*]: *spmf* (*pair-spmf p q*) (*x*, *y*) = *spmf p x* ∗ *spmf q y* (**is** *?lhs* = *?rhs*)
proof −
have *ennreal ?lhs* = ∫⁺ *a*. ∫⁺ *b*. *indicator* {(*x*, *y*)} (*a*, *b*) ∂*measure-spmf q* ∂*measure-spmf p*
unfolding *measure-spmf-def pair-spmf-def ennreal-pmf-bind nn-integral-pair-pmf'*
by(*auto simp add*: *zero-ereal-def*[*symmetric*] *nn-integral-distr nn-integral-restrict-space nn-integral-multc*[*symmetric*] *intro*!: *nn-integral-cong split*: *option.split split-indicator*)
also have ... = ∫⁺ *a*. (∫⁺ *b*. *indicator* {*y*} *b* ∂*measure-spmf q*) ∗ *indicator* {*x*} *a* ∂*measure-spmf p*
by(*subst nn-integral-multc*[*symmetric*])(*auto intro*!: *nn-integral-cong split*: *split-indicator*)
also have ... = *ennreal ?rhs* **by**(*simp add*: *emeasure-spmf-single max-def ennreal-mult mult.commute*)
finally show *?thesis* **by** *simp*
qed

lemma *pair-map-spmf2*: *pair-spmf p* (*map-spmf f q*) = *map-spmf* (*apsnd f*) (*pair-spmf p q*)
by(*auto simp add*: *pair-spmf-def pair-map-pmf2 bind-map-pmf map-bind-pmf intro*: *bind-pmf-cong split*: *option.split*)

lemma *pair-map-spmf1*: *pair-spmf* (*map-spmf f p*) *q* = *map-spmf* (*apfst f*) (*pair-spmf p q*)
by(*auto simp add*: *pair-spmf-def pair-map-pmf1 bind-map-pmf map-bind-pmf intro*: *bind-pmf-cong split*: *option.split*)

lemma *pair-map-spmf*: *pair-spmf* (*map-spmf f p*) (*map-spmf g q*) = *map-spmf* (*map-prod f g*) (*pair-spmf p q*)
unfolding *pair-map-spmf2 pair-map-spmf1 spmf.map-comp* **by**(*simp add*: *apfst-def apsnd-def o-def prod.map-comp*)

lemma *pair-spmf-alt-def*: *pair-spmf p q* = *bind-spmf p* (λ*x*. *bind-spmf q* (λ*y*. *return-spmf* (*x*, *y*)))
by(*auto simp add*: *pair-spmf-def pair-pmf-def bind-spmf-def bind-assoc-pmf bind-return-pmf split*: *option.split intro*: *bind-pmf-cong*)

lemma *weight-pair-spmf* [*simp*]: *weight-spmf* (*pair-spmf p q*) = *weight-spmf p* ∗ *weight-spmf q*
unfolding *pair-spmf-alt-def* **by**(*simp add*: *weight-bind-spmf o-def*)

lemma *pair-scale-spmf1*:
r ≤ *1* ⟹ *pair-spmf* (*scale-spmf r p*) *q* = *scale-spmf r* (*pair-spmf p q*)
by(*simp add*: *pair-spmf-alt-def scale-bind-spmf bind-scale-spmf*)

lemma *pair-scale-spmf2*:
  $r \leq 1 \Longrightarrow$ *pair-spmf p* (*scale-spmf r q*) = *scale-spmf r* (*pair-spmf p q*)
by(*simp add*: *pair-spmf-alt-def scale-bind-spmf bind-scale-spmf*)

lemma *pair-spmf-return-None1* [*simp*]: *pair-spmf* (*return-pmf None*) *p* = *return-pmf None*
by(*rule spmf-eqI*)(*clarsimp*)

lemma *pair-spmf-return-None2* [*simp*]: *pair-spmf p* (*return-pmf None*) = *return-pmf None*
by(*rule spmf-eqI*)(*clarsimp*)

lemma *pair-spmf-return-spmf1*: *pair-spmf* (*return-spmf x*) *q* = *map-spmf* (*Pair x*) *q*
by(*rule spmf-eqI*)(*auto split*: *split-indicator simp add*: *spmf-map-inj′ inj-on-def intro*: *spmf-map-outside*)

lemma *pair-spmf-return-spmf2*: *pair-spmf p* (*return-spmf y*) = *map-spmf* (λ*x*. (*x*, *y*)) *p*
by(*rule spmf-eqI*)(*auto split*: *split-indicator simp add*: *inj-on-def intro*!: *spmf-map-outside spmf-map-inj′*[*symmetric*])

lemma *pair-spmf-return-spmf* [*simp*]: *pair-spmf* (*return-spmf x*) (*return-spmf y*) = *return-spmf* (*x*, *y*)
by(*simp add*: *pair-spmf-return-spmf1*)

lemma *rel-pair-spmf-prod*:
  rel-spmf (*rel-prod A B*) (*pair-spmf p q*) (*pair-spmf p′ q′*) ⟷
   rel-spmf A (*scale-spmf* (*weight-spmf q*) *p*) (*scale-spmf* (*weight-spmf q′*) *p′*) ∧
   rel-spmf B (*scale-spmf* (*weight-spmf p*) *q*) (*scale-spmf* (*weight-spmf p′*) *q′*)
  (**is** *?lhs* ⟷ *?rhs* **is** - ⟷ *?A* ∧ *?B* **is** - ⟷ *rel-spmf* - *?p ?p′* ∧ *rel-spmf* - *?q ?q′*)
proof(*intro iffI conjI*)
  assume *?rhs*
  then obtain *pq pq′* **where** *p*: *map-spmf fst pq* = *?p* **and** *p′*: *map-spmf snd pq* = *?p′*
    and *q*: *map-spmf fst pq′* = *?q* **and** *q′*: *map-spmf snd pq′* = *?q′*
    and ∗: ⋀*x x′*. (*x*, *x′*) ∈ *set-spmf pq* ⟹ *A x x′*
    and ∗∗: ⋀*y y′*. (*y*, *y′*) ∈ *set-spmf pq′* ⟹ *B y y′* **by**(*auto elim*!: *rel-spmfE*)
  let *?f* = λ((*x*, *x′*), (*y*, *y′*)). ((*x*, *y*), (*x′*, *y′*))
  let *?r* = *1* / (*weight-spmf p* ∗ *weight-spmf q*)
  let *?pq* = *scale-spmf ?r* (*map-spmf ?f* (*pair-spmf pq pq′*))

  { fix *p* :: *′x spmf* **and** *q* :: *′y spmf*
    assume *weight-spmf q* ≠ *0*
      and *weight-spmf p* ≠ *0*
      and *1* / (*weight-spmf p* ∗ *weight-spmf q*) ≤ *weight-spmf p* ∗ *weight-spmf q*
    hence *1* ≤ (*weight-spmf p* ∗ *weight-spmf q*) ∗ (*weight-spmf p* ∗ *weight-spmf q*)

by(*simp add*: *pos-divide-le-eq order.strict-iff-order weight-spmf-nonneg*)
moreover have (*weight-spmf p* ∗ *weight-spmf q*) ∗ (*weight-spmf p* ∗ *weight-spmf q*) ≤ (*1* ∗ *1*) ∗ (*1* ∗ *1*)
by(*intro mult-mono*)(*simp-all add*: *weight-spmf-nonneg weight-spmf-le-1*)
ultimately have (*weight-spmf p* ∗ *weight-spmf q*) ∗ (*weight-spmf p* ∗ *weight-spmf q*) = *1* **by** *simp*
hence ∗: *weight-spmf p* ∗ *weight-spmf q* = *1*
by(*metis antisym-conv less-le mult-less-cancel-left1 weight-pair-spmf weight-spmf-le-1 weight-spmf-nonneg*)
hence ∗∗: *weight-spmf p* = *1* **by**(*metis antisym-conv mult-left-le weight-spmf-le-1 weight-spmf-nonneg*)
moreover from ∗ ∗∗ **have** *weight-spmf q* = *1* **by** *simp*
moreover note *calculation* **}**
note *full* = *this*

show *?lhs*
proof
have [*simp*]: *fst* ∘ *?f* = *map-prod fst fst* **by**(*simp add*: *fun-eq-iff*)
have *map-spmf fst ?pq* = *scale-spmf ?r* (*pair-spmf ?p ?q*)
by(*simp add*: *pair-map-spmf*[*symmetric*] *p q map-scale-spmf spmf.map-comp*)
also have ... = *pair-spmf p q* **using** *full*[*of p q*]
by(*simp add*: *pair-scale-spmf1 pair-scale-spmf2 weight-spmf-le-1 weight-spmf-nonneg*)
(*auto simp add*: *scale-scale-spmf max-def min-def field-simps weight-spmf-nonneg weight-spmf-eq-0*)
finally show *map-spmf fst ?pq* = ... **.**

have [*simp*]: *snd* ∘ *?f* = *map-prod snd snd* **by**(*simp add*: *fun-eq-iff*)
from ‹*?rhs*› **have** *eq*: *weight-spmf p* ∗ *weight-spmf q* = *weight-spmf p′* ∗ *weight-spmf q′*
by(*auto dest*!: *rel-spmf-weightD simp add*: *weight-spmf-le-1 weight-spmf-nonneg*)

have *map-spmf snd ?pq* = *scale-spmf ?r* (*pair-spmf ?p′ ?q′*)
by(*simp add*: *pair-map-spmf*[*symmetric*] *p′ q′ map-scale-spmf spmf.map-comp*)
also have ... = *pair-spmf p′ q′* **using** *full*[*of p′ q′*] *eq*
by(*simp add*: *pair-scale-spmf1 pair-scale-spmf2 weight-spmf-le-1 weight-spmf-nonneg*)
(*auto simp add*: *scale-scale-spmf max-def min-def field-simps weight-spmf-nonneg weight-spmf-eq-0*)
finally show *map-spmf snd ?pq* = ... **.**
qed(*auto simp add*: *set-scale-spmf split*: *if-split-asm dest*: ∗ ∗∗)
next
assume *?lhs*
then obtain *pq* **where** *pq*: *map-spmf fst pq* = *pair-spmf p q*
and *pq′*: *map-spmf snd pq* = *pair-spmf p′ q′*
and ∗: ⋀*x y x′ y′*. ((*x*, *y*), (*x′*, *y′*)) ∈ *set-spmf pq* ⟹ *A x x′* ∧ *B y y′*
by(*auto elim*: *rel-spmfE*)

show *?A*
proof
let *?f* = (λ((*x*, *y*), (*x′*, *y′*)). (*x*, *x′*))

```
    let ?pq = map-spmf ?f pq
    have [simp]: fst ∘ ?f = fst ∘ fst by(simp add: split-def o-def)
    show map-spmf fst ?pq = scale-spmf (weight-spmf q) p using pq
      by(simp add: spmf.map-comp)(simp add: spmf.map-comp[symmetric])

    have [simp]: snd ∘ ?f = fst ∘ snd by(simp add: split-def o-def)
    show map-spmf snd ?pq = scale-spmf (weight-spmf q') p' using pq'
      by(simp add: spmf.map-comp)(simp add: spmf.map-comp[symmetric])
  qed(auto dest: * )

  show ?B
  proof
    let ?f = (λ((x, y), (x', y')). (y, y'))
    let ?pq = map-spmf ?f pq
    have [simp]: fst ∘ ?f = snd ∘ fst by(simp add: split-def o-def)
    show map-spmf fst ?pq = scale-spmf (weight-spmf p) q using pq
      by(simp add: spmf.map-comp)(simp add: spmf.map-comp[symmetric])

    have [simp]: snd ∘ ?f = snd ∘ snd by(simp add: split-def o-def)
    show map-spmf snd ?pq = scale-spmf (weight-spmf p') q' using pq'
      by(simp add: spmf.map-comp)(simp add: spmf.map-comp[symmetric])
  qed(auto dest: * )
qed

lemma pair-pair-spmf:
  pair-spmf (pair-spmf p q) r = map-spmf (λ(x, (y, z)). ((x, y), z)) (pair-spmf p
(pair-spmf q r))
by(simp add: pair-spmf-alt-def map-spmf-conv-bind-spmf)

lemma pair-commute-spmf:
  pair-spmf p q = map-spmf (λ(y, x). (x, y)) (pair-spmf q p)
unfolding pair-spmf-alt-def by(subst bind-commute-spmf)(simp add: map-spmf-conv-bind-spmf)
```

25.18 Assertions

```
definition assert-spmf :: bool ⇒ unit spmf
where assert-spmf b = (if b then return-spmf () else return-pmf None)

lemma assert-spmf-simps [simp]:
  assert-spmf True = return-spmf ()
  assert-spmf False = return-pmf None
by(simp-all add: assert-spmf-def)

lemma in-set-assert-spmf [simp]: x ∈ set-spmf (assert-spmf p) ⟷ p
by(cases p) simp-all

lemma set-spmf-assert-spmf-eq-empty [simp]: set-spmf (assert-spmf b) = {} ⟷
¬ b
by(cases b) simp-all
```

lemma *lossless-assert-spmf* [*iff*]: *lossless-spmf* (*assert-spmf b*) ⟷ *b*
by(*cases b*) *simp-all*

25.19 Try

definition *try-spmf* :: *'a spmf* ⇒ *'a spmf* ⇒ *'a spmf* (*TRY - ELSE - [0,60] 59*)
where *try-spmf p q* = *bind-pmf p* (λ*x*. *case x of None* ⇒ *q* | *Some y* ⇒ *return-spmf y*)

lemma *try-spmf-lossless* [*simp*]:
assumes *lossless-spmf p*
shows *TRY p ELSE q* = *p*
proof −
have *TRY p ELSE q* = *bind-pmf p return-pmf* **unfolding** *try-spmf-def* **using** *assms*
by(*auto simp add*: *lossless-iff-set-pmf-None split*: *option.split intro*: *bind-pmf-cong*)
thus *?thesis* **by**(*simp add*: *bind-return-pmf'*)
qed

lemma *try-spmf-return-spmf1*: *TRY return-spmf x ELSE q* = *return-spmf x*
by(*simp add*: *try-spmf-def bind-return-pmf*)

lemma *try-spmf-return-None* [*simp*]: *TRY return-pmf None ELSE q* = *q*
by(*simp add*: *try-spmf-def bind-return-pmf*)

lemma *try-spmf-return-pmf-None2* [*simp*]: *TRY p ELSE return-pmf None* = *p*
by(*simp add*: *try-spmf-def option.case-distrib*[*symmetric*] *bind-return-pmf' case-option-id*)

lemma *map-try-spmf*: *map-spmf f* (*try-spmf p q*) = *try-spmf* (*map-spmf f p*) (*map-spmf f q*)
by(*simp add*: *try-spmf-def map-bind-pmf bind-map-pmf option.case-distrib*[**where** *h*=*map-spmf f*] *o-def cong del*: *option.case-cong-weak*)

lemma *try-spmf-bind-pmf*: *TRY* (*bind-pmf p f*) *ELSE q* = *bind-pmf p* (λ*x*. *TRY* (*f x*) *ELSE q*)
by(*simp add*: *try-spmf-def bind-assoc-pmf*)

lemma *try-spmf-bind-spmf-lossless*:
lossless-spmf p ⟹ *TRY* (*bind-spmf p f*) *ELSE q* = *bind-spmf p* (λ*x*. *TRY* (*f x*) *ELSE q*)
by(*auto simp add*: *try-spmf-def bind-spmf-def bind-assoc-pmf bind-return-pmf loss-less-iff-set-pmf-None intro*!: *bind-pmf-cong split*: *option.split*)

lemma *try-spmf-bind-out*:
lossless-spmf p ⟹ *bind-spmf p* (λ*x*. *TRY* (*f x*) *ELSE q*) = *TRY* (*bind-spmf p f*) *ELSE q*
by(*simp add*: *try-spmf-bind-spmf-lossless*)

lemma *lossless-try-spmf* [*simp*]:
lossless-spmf (*TRY p ELSE q*) ⟷ *lossless-spmf p* ∨ *lossless-spmf q*
by(*auto simp add*: *try-spmf-def in-set-spmf lossless-iff-set-pmf-None split*: *option.splits*)

context includes *lifting-syntax*
begin

lemma *try-spmf-parametric* [*transfer-rule*]:
(*rel-spmf A* ===> *rel-spmf A* ===> *rel-spmf A*) *try-spmf try-spmf*
unfolding *try-spmf-def*[*abs-def*] **by** *transfer-prover*

end

lemma *try-spmf-cong*:
⟦ *p* = *p'*; ¬ *lossless-spmf p'* ⟹ *q* = *q'* ⟧ ⟹ *TRY p ELSE q* = *TRY p' ELSE q'*
unfolding *try-spmf-def*
by(*rule bind-pmf-cong*)(*auto split*: *option.split simp add*: *lossless-iff-set-pmf-None*)

lemma *rel-spmf-try-spmf*:
⟦ *rel-spmf R p p'*; ¬ *lossless-spmf p'* ⟹ *rel-spmf R q q'* ⟧
⟹ *rel-spmf R* (*TRY p ELSE q*) (*TRY p' ELSE q'*)
unfolding *try-spmf-def*
apply(*rule rel-pmf-bindI*[**where** *R*=λ*x y*. *rel-option R x y* ∧ *x* ∈ *set-pmf p* ∧ *y* ∈ *set-pmf p'*])
apply(*erule pmf.rel-mono-strong*; *simp*)
apply(*auto split*: *option.split simp add*: *lossless-iff-set-pmf-None*)
done

lemma *spmf-try-spmf*:
spmf (*TRY p ELSE q*) *x* = *spmf p x* + *pmf p None* ∗ *spmf q x*
proof −
have *ennreal* (*spmf* (*TRY p ELSE q*) *x*) = ∫⁺ *y*. *ennreal* (*spmf q x*) ∗ *indicator* {*None*} *y* + *indicator* {*Some x*} *y* ∂*measure-pmf p*
unfolding *try-spmf-def ennreal-pmf-bind* **by**(*rule nn-integral-cong*)(*simp split*: *option.split split-indicator*)
also have ... = (∫⁺ *y*. *ennreal* (*spmf q x*) ∗ *indicator* {*None*} *y* ∂*measure-pmf p*) + ∫⁺ *y*. *indicator* {*Some x*} *y* ∂*measure-pmf p*
by(*simp add*: *nn-integral-add*)
also have ... = *ennreal* (*spmf q x*) ∗ *pmf p None* + *spmf p x* **by**(*simp add*: *emeasure-pmf-single*)
finally show *?thesis* **by**(*simp add*: *ennreal-mult*[*symmetric*] *ennreal-plus*[*symmetric*] *del*: *ennreal-plus*)
qed

lemma *try-scale-spmf-same* [*simp*]: *lossless-spmf p* ⟹ *TRY scale-spmf k p ELSE p* = *p*
by(*rule spmf-eqI*)(*auto simp add*: *spmf-try-spmf spmf-scale-spmf pmf-scale-spmf-None lossless-iff-pmf-None weight-spmf-conv-pmf-None min-def max-def field-simps*)

lemma *pmf-try-spmf-None* [*simp*]: *pmf* (*TRY p ELSE q*) *None* = *pmf p None* ∗ *pmf q None* (**is** *?lhs* = *?rhs*)
proof −
have *?lhs* = ∫ *x*. *pmf q None* ∗ *indicator* {*None*} *x* ∂*measure-pmf p*
unfolding *try-spmf-def pmf-bind* **by**(*rule Bochner-Integration.integral-cong*)(*simp-all split*: *option.split*)
also have ... = *?rhs* **by**(*simp add*: *measure-pmf-single*)
finally show *?thesis* **.**
qed

lemma *try-bind-spmf-lossless2*:
lossless-spmf q ⟹ *TRY* (*bind-spmf p f*) *ELSE q* = *TRY* (*p* ⤜ (λ*x*. *TRY* (*f x*) *ELSE q*)) *ELSE q*
by(*rule spmf-eqI*)(*simp add*: *spmf-try-spmf pmf-bind-spmf-None spmf-bind field-simps measure-spmf.integrable-const-bound*[**where** *B*=*1*] *pmf-le-1 lossless-iff-pmf-None*)

lemma *try-bind-spmf-lossless2′*:
fixes *f* :: *′a* ⇒ *′b spmf* **shows**
⟦ *NO-MATCH* (λ*x* :: *′a*. *try-spmf* (*g x* :: *′b spmf*) (*h x*)) *f*; *lossless-spmf q* ⟧
⟹ *TRY* (*bind-spmf p f*) *ELSE q* = *TRY* (*p* ⤜ (λ*x* :: *′a*. *TRY* (*f x*) *ELSE q*)) *ELSE q*
by(*rule try-bind-spmf-lossless2*)

lemma *try-bind-assert-spmf*:
TRY (*assert-spmf b* ⤜ *f*) *ELSE q* = (*if b then TRY* (*f* ()) *ELSE q else q*)
by *simp*

25.20 Miscellaneous

lemma assumes *rel-spmf* (λ*x y*. *bad1 x* = *bad2 y* ∧ (¬ *bad2 y* ⟶ *A x* ⟷ *B y*)) *p q* (**is** *rel-spmf ?A* - -)
shows *fundamental-lemma-bad*: *measure* (*measure-spmf p*) {*x*. *bad1 x*} = *measure* (*measure-spmf q*) {*y*. *bad2 y*} (**is** *?bad*)
and *fundamental-lemma*: |*measure* (*measure-spmf p*) {*x*. *A x*} − *measure* (*measure-spmf q*) {*y*. *B y*}| ≤
measure (*measure-spmf p*) {*x*. *bad1 x*} (**is** *?fundamental*)
proof −
have *good*: *rel-fun ?A* (=) (λ*x*. *A x* ∧ ¬ *bad1 x*) (λ*y*. *B y* ∧ ¬ *bad2 y*) **by**(*auto simp add*: *rel-fun-def*)
from *assms* **have** *1*: *measure* (*measure-spmf p*) {*x*. *A x* ∧ ¬ *bad1 x*} = *measure* (*measure-spmf q*) {*y*. *B y* ∧ ¬ *bad2 y*}
by(*rule measure-spmf-parametric*[*THEN rel-funD*, *THEN rel-funD*])(*rule Collect-parametric*[*THEN rel-funD*, *OF good*])

have *bad*: *rel-fun ?A* (=) *bad1 bad2* **by**(*simp add*: *rel-fun-def*)
show *2*: *?bad* **using** *assms*
by(*rule measure-spmf-parametric*[*THEN rel-funD*, *THEN rel-funD*])(*rule Collect-parametric*[*THEN rel-funD*, *OF bad*])

let *?μp = measure (measure-spmf p)* **and** *?μq = measure (measure-spmf q)*
have *{x. A x ∧ bad1 x} ∪ {x. A x ∧ ¬ bad1 x} = {x. A x}*
and *{y. B y ∧ bad2 y} ∪ {y. B y ∧ ¬ bad2 y} = {y. B y}* **by** *auto*
then have *|?μp {x. A x} − ?μq {x. B x}| = |?μp ({x. A x ∧ bad1 x} ∪ {x. A x ∧ ¬ bad1 x}) − ?μq ({y. B y ∧ bad2 y} ∪ {y. B y ∧ ¬ bad2 y})|*
by *simp*
also have *... = |?μp {x. A x ∧ bad1 x} + ?μp {x. A x ∧ ¬ bad1 x} − ?μq {y. B y ∧ bad2 y} − ?μq {y. B y ∧ ¬ bad2 y}|*
by(*subst (1 2) measure-Union*)(*auto*)
also have *... = |?μp {x. A x ∧ bad1 x} − ?μq {y. B y ∧ bad2 y}|* **using** *1* **by** *simp*
also have *... ≤ max (?μp {x. A x ∧ bad1 x}) (?μq {y. B y ∧ bad2 y})*
by(*rule abs-leI*)(*auto simp add*: *max-def not-le*, *simp-all only*: *add-increasing measure-nonneg mult-2*)
also have *... ≤ max (?μp {x. bad1 x}) (?μq {y. bad2 y})*
by(*rule max.mono*; *rule measure-spmf.finite-measure-mono*; *auto*)
also note *2*[*symmetric*]
finally show *?fundamental* **by** *simp*
qed

end

26 Indexed products of PMFs

theory *Product-PMF*
imports *Probability-Mass-Function Independent-Family*
begin

26.1 Preliminaries

lemma *pmf-expectation-eq-infsetsum*: *measure-pmf.expectation p f = infsetsum (λx. pmf p x ∗ f x) UNIV*
unfolding *infsetsum-def measure-pmf-eq-density* **by** (*subst integral-density*) *simp-all*

lemma *measure-pmf-prob-product*:
assumes *countable A countable B*
shows *measure-pmf.prob (pair-pmf M N) (A × B) = measure-pmf.prob M A ∗ measure-pmf.prob N B*
proof −
have *measure-pmf.prob (pair-pmf M N) (A × B) = (∑a(a, b)∈A × B. pmf M a ∗ pmf N b)*
by (*auto intro*!: *infsetsum-cong simp add*: *measure-pmf-conv-infsetsum pmf-pair*)
also have *... = measure-pmf.prob M A ∗ measure-pmf.prob N B*
using *assms* **by** (*subst infsetsum-product*) (*auto simp add*: *measure-pmf-conv-infsetsum*)
finally show *?thesis*
by *simp*
qed

26.2 Definition

In analogy to Pi_M, we define an indexed product of PMFs. In the literature, this is typically called taking a vector of independent random variables. Note that the components do not have to be identically distributed.

The operation takes an explicit index set A and a function f that maps each element from A to a PMF and defines the product measure $\bigotimes_{i \in A} f(i)$, which is represented as a $('a \Rightarrow 'b)$ *pmf*.

Note that unlike Pi_M, this only works for *finite* index sets. It could be extended to countable sets and beyond, but the construction becomes somewhat more involved.

definition *Pi-pmf* :: *'a set* ⇒ *'b* ⇒ (*'a* ⇒ *'b pmf*) ⇒ (*'a* ⇒ *'b*) *pmf* **where**
Pi-pmf A dflt p =
embed-pmf (λ*f*. *if* (∀ *x*. *x* ∉ *A* ⟶ *f x* = *dflt*) *then* ∏ *x*∈*A*. *pmf* (*p x*) (*f x*) *else 0*)

A technical subtlety that needs to be addressed is this: Intuitively, the functions in the support of a product distribution have domain A. However, since HOL is a total logic, these functions must still return *some* value for inputs outside A. The product measure Pi_M simply lets these functions return *undefined* in these cases. We chose a different solution here, which is to supply a default value *dflt* that is returned in these cases.

As one possible application, one could model the result of n different independent coin tosses as *Pi-pmf* {*0*::*'a*..<*n*} *False* (λ-. *bernoulli-pmf* (*1* / *2*)). This returns a function of type *nat* ⇒ *bool* that maps every natural number below n to the result of the corresponding coin toss, and every other natural number to *False*.

lemma *pmf-Pi*:
assumes *A*: *finite A*
shows *pmf* (*Pi-pmf A dflt p*) *f* =
(*if* (∀ *x*. *x* ∉ *A* ⟶ *f x* = *dflt*) *then* ∏ *x*∈*A*. *pmf* (*p x*) (*f x*) *else 0*)
unfolding *Pi-pmf-def*
proof (*rule pmf-embed-pmf*, *goal-cases*)
case *2*
define *S* **where** *S* = {*f*. ∀ *x*. *x* ∉ *A* ⟶ *f x* = *dflt*}
define *B* **where** *B* = (λ*x*. *set-pmf* (*p x*))

have *neutral-left*: (∏ *xa*∈*A*. *pmf* (*p xa*) (*f xa*)) = *0*
if *f* ∈ *PiE A B* − (λ*f*. *restrict f A*) ‘ *S* **for** *f*
proof −
have *restrict* (λ*x*. *if x* ∈ *A then f x else dflt*) *A* ∈ (λ*f*. *restrict f A*) ‘ *S*
by (*intro imageI*) (*auto simp*: *S-def*)
also have *restrict* (λ*x*. *if x* ∈ *A then f x else dflt*) *A* = *f*
using *that* **by** (*auto simp*: *PiE-def Pi-def extensional-def fun-eq-iff*)
finally show *?thesis* **using** *that* **by** *blast*
qed

have *neutral-right*: $(\prod xa{\in}A.\ pmf\ (p\ xa)\ (f\ xa)) = 0$
if $f \in (\lambda f.\ restrict\ f\ A)\ `\ S - PiE\ A\ B$ **for** f
proof −
from *that* **obtain** f' **where** f': $f = restrict\ f'\ A\ f' \in S$ **by** *auto*
moreover from *this* **and** *that* **have** $restrict\ f'\ A \notin PiE\ A\ B$ **by** *simp*
then obtain x **where** $x \in A\ pmf\ (p\ x)\ (f'\ x) = 0$ **by** (*auto simp*: *B-def set-pmf-eq*)
with f' **and** A **show** *?thesis* **by** *auto*
qed

have $(\lambda f.\ \prod x{\in}A.\ pmf\ (p\ x)\ (f\ x))$ *abs-summable-on PiE A B*
by (*intro abs-summable-on-prod-PiE A*) (*auto simp*: *B-def*)
also have *?this* $\longleftrightarrow$ $(\lambda f.\ \prod x{\in}A.\ pmf\ (p\ x)\ (f\ x))$ *abs-summable-on* $(\lambda f.\ restrict\ f\ A)\ `\ S$
by (*intro abs-summable-on-cong-neutral neutral-left neutral-right*) *auto*
also have ... $\longleftrightarrow$ $(\lambda f.\ \prod x{\in}A.\ pmf\ (p\ x)\ (restrict\ f\ A\ x))$ *abs-summable-on S*
by (*rule abs-summable-on-reindex-iff* [*symmetric*]) (*force simp*: *inj-on-def fun-eq-iff S-def*)
also have ... $\longleftrightarrow$ $(\lambda f.\ if\ \forall x.\ x \notin A \longrightarrow f\ x = dflt\ then\ \prod x{\in}A.\ pmf\ (p\ x)\ (f\ x)\ else\ 0)$
abs-summable-on UNIV
by (*intro abs-summable-on-cong-neutral*) (*auto simp*: *S-def*)
finally have *summable*: ... **.**

have $1 = (\prod x{\in}A.\ 1{::}real)$ **by** *simp*
also have $(\prod x{\in}A.\ 1) = (\prod x{\in}A.\ \sum_a y{\in}B\ x.\ pmf\ (p\ x)\ y)$
unfolding *B-def* **by** (*subst infsetsum-pmf-eq-1*) *auto*
also have $(\prod x{\in}A.\ \sum_a y{\in}B\ x.\ pmf\ (p\ x)\ y) = (\sum_a f{\in}Pi_E\ A\ B.\ \prod x{\in}A.\ pmf\ (p\ x)\ (f\ x))$
by (*intro infsetsum-prod-PiE* [*symmetric*] *A*) (*auto simp*: *B-def*)
also have ... $= (\sum_a f{\in}(\lambda f.\ restrict\ f\ A)\ `\ S.\ \prod x{\in}A.\ pmf\ (p\ x)\ (f\ x))$ **using** *A*
by (*intro infsetsum-cong-neutral neutral-left neutral-right refl*)
also have ... $= (\sum_a f{\in}S.\ \prod x{\in}A.\ pmf\ (p\ x)\ (restrict\ f\ A\ x))$
by (*rule infsetsum-reindex*) (*force simp*: *inj-on-def fun-eq-iff S-def*)
also have ... $= (\sum_a f{\in}S.\ \prod x{\in}A.\ pmf\ (p\ x)\ (f\ x))$
by (*intro infsetsum-cong*) (*auto simp*: *S-def*)
also have ... $= (\sum_a f.\ if\ \forall x.\ x \notin A \longrightarrow f\ x = dflt\ then\ \prod x{\in}A.\ pmf\ (p\ x)\ (f\ x)\ else\ 0)$
by (*intro infsetsum-cong-neutral*) (*auto simp*: *S-def*)
also have *ennreal* ... $= (\int^+ f.\ ennreal\ (if\ \forall x.\ x \notin A \longrightarrow f\ x = dflt\ then\ \prod x{\in}A.\ pmf\ (p\ x)\ (f\ x)\ else\ 0)\ \partial count\text{-}space\ UNIV)$
by (*intro nn-integral-conv-infsetsum* [*symmetric*] *summable*) (*auto simp*: *prod-nonneg*)
finally show *?case* **by** *simp*
qed (*auto simp*: *prod-nonneg*)

lemma *Pi-pmf-cong*:
assumes $A = A'$ $dflt = dflt'$ $\bigwedge x.\ x \in A \Longrightarrow f\ x = f'\ x$
shows *Pi-pmf A dflt f* = *Pi-pmf A' dflt' f'*
proof −

have $(\lambda fa.\ \mathit{if}\ \forall x.\ x \notin A \longrightarrow fa\ x = \mathit{dflt}\ \mathit{then}\ \prod x{\in}A.\ \mathit{pmf}\ (f\ x)\ (fa\ x)\ \mathit{else}\ 0) =$
$(\lambda f.\ \mathit{if}\ \forall x.\ x \notin A' \longrightarrow f\ x = \mathit{dflt}'\ \mathit{then}\ \prod x{\in}A'.\ \mathit{pmf}\ (f'\ x)\ (f\ x)\ \mathit{else}\ 0)$
using *assms* **by** (*intro ext*) (*auto intro*!: *prod.cong*)
thus *?thesis*
by (*simp only*: *Pi-pmf-def*)
qed

lemma *pmf-Pi'*:
assumes *finite A* $\bigwedge x.\ x \notin A \Longrightarrow f\ x = \mathit{dflt}$
shows *pmf* (*Pi-pmf A dflt p*) *f* = $(\prod x{\in}A.\ \mathit{pmf}\ (p\ x)\ (f\ x))$
using *assms* **by** (*subst pmf-Pi*) *auto*

lemma *pmf-Pi-outside*:
assumes *finite A* $\exists x.\ x \notin A \wedge f\ x \neq \mathit{dflt}$
shows *pmf* (*Pi-pmf A dflt p*) *f* = *0*
using *assms* **by** (*subst pmf-Pi*) *auto*

lemma *pmf-Pi-empty* [*simp*]: *Pi-pmf* {} *dflt p* = *return-pmf* (λ-. *dflt*)
by (*intro pmf-eqI*, *subst pmf-Pi*) (*auto simp*: *indicator-def*)

lemma *set-Pi-pmf-subset*: *finite A* $\Longrightarrow$ *set-pmf* (*Pi-pmf A dflt p*) $\subseteq \{f.\ \forall x.\ x \notin A \longrightarrow f\ x = \mathit{dflt}\}$
by (*auto simp*: *set-pmf-eq pmf-Pi*)

26.3 Dependent product sets with a default

The following describes a dependent product of sets where the functions are required to return the default value *dflt* outside their domain, in analogy to Pi_E, which uses *undefined*.

definition *PiE-dflt*
where *PiE-dflt A dflt B* = $\{f.\ \forall x.\ (x \in A \longrightarrow f\ x \in B\ x) \wedge (x \notin A \longrightarrow f\ x = \mathit{dflt})\}$

lemma *restrict-PiE-dflt*: (λ*h*. *restrict h A*) ' *PiE-dflt A dflt B* = *PiE A B*
proof (*intro equalityI subsetI*)
fix *h* **assume** *h* ∈ (λ*h*. *restrict h A*) ' *PiE-dflt A dflt B*
thus *h* ∈ *PiE A B*
by (*auto simp*: *PiE-dflt-def*)
next
fix *h* **assume** *h*: *h* ∈ *PiE A B*
hence *restrict* (λ*x*. *if x* ∈ *A then h x else dflt*) *A* ∈ (λ*h*. *restrict h A*) ' *PiE-dflt A dflt B*
by (*intro imageI*) (*auto simp*: *PiE-def extensional-def PiE-dflt-def*)
also have *restrict* (λ*x*. *if x* ∈ *A then h x else dflt*) *A* = *h*
using *h* **by** (*auto simp*: *fun-eq-iff*)
finally show *h* ∈ (λ*h*. *restrict h A*) ' *PiE-dflt A dflt B* **.**
qed

lemma *dflt-image-PiE*: $(\lambda h\ x.\ \mathit{if}\ x \in A\ \mathit{then}\ h\ x\ \mathit{else}\ \mathit{dflt})\ `\ \mathit{PiE}\ A\ B = \mathit{PiE\text{-}dflt}\ A\ \mathit{dflt}\ B$
(**is** $?f\ `\ ?X = ?Y$)
proof (*intro equalityI subsetI*)
fix h **assume** $h \in ?f\ `\ ?X$
thus $h \in ?Y$
by (*auto simp*: *PiE-dflt-def PiE-def*)
next
fix h **assume** *h*: $h \in ?Y$
hence $?f\ (\mathit{restrict}\ h\ A) \in ?f\ `\ ?X$
by (*intro imageI*) (*auto simp*: *PiE-def extensional-def PiE-dflt-def*)
also have $?f\ (\mathit{restrict}\ h\ A) = h$
using h **by** (*auto simp*: *fun-eq-iff PiE-dflt-def*)
finally show $h \in ?f\ `\ ?X$ **.**
qed

lemma *finite-PiE-dflt* [*intro*]:
assumes *finite* A $\bigwedge x.\ x \in A \Longrightarrow \mathit{finite}\ (B\ x)$
shows *finite* (*PiE-dflt A d B*)
proof −
have $\mathit{PiE\text{-}dflt}\ A\ d\ B = (\lambda f\ x.\ \mathit{if}\ x \in A\ \mathit{then}\ f\ x\ \mathit{else}\ d)\ `\ \mathit{PiE}\ A\ B$
by (*rule dflt-image-PiE* [*symmetric*])
also have *finite* ...
by (*intro finite-imageI finite-PiE assms*)
finally show *?thesis* **.**
qed

lemma *card-PiE-dflt*:
assumes *finite* A $\bigwedge x.\ x \in A \Longrightarrow \mathit{finite}\ (B\ x)$
shows $\mathit{card}\ (\mathit{PiE\text{-}dflt}\ A\ d\ B) = (\prod x{\in}A.\ \mathit{card}\ (B\ x))$
proof −
from *assms* **have** $(\prod x{\in}A.\ \mathit{card}\ (B\ x)) = \mathit{card}\ (\mathit{PiE}\ A\ B)$
by (*intro card-PiE* [*symmetric*]) *auto*
also have $\mathit{PiE}\ A\ B = (\lambda f.\ \mathit{restrict}\ f\ A)\ `\ \mathit{PiE\text{-}dflt}\ A\ d\ B$
by (*rule restrict-PiE-dflt* [*symmetric*])
also have $\mathit{card}\ \ldots = \mathit{card}\ (\mathit{PiE\text{-}dflt}\ A\ d\ B)$
by (*intro card-image*) (*force simp*: *inj-on-def restrict-def fun-eq-iff PiE-dflt-def*)
finally show *?thesis* **..**
qed

lemma *PiE-dflt-empty-iff* [*simp*]: $\mathit{PiE\text{-}dflt}\ A\ \mathit{dflt}\ B = \{\} \longleftrightarrow (\exists x{\in}A.\ B\ x = \{\})$
by (*simp add*: *dflt-image-PiE* [*symmetric*] *PiE-eq-empty-iff*)

lemma *set-Pi-pmf-subset′*:
assumes *finite A*
shows $\mathit{set\text{-}pmf}\ (\mathit{Pi\text{-}pmf}\ A\ \mathit{dflt}\ p) \subseteq \mathit{PiE\text{-}dflt}\ A\ \mathit{dflt}\ (\mathit{set\text{-}pmf} \circ p)$
using *assms* **by** (*auto simp*: *set-pmf-eq pmf-Pi PiE-dflt-def*)

lemma *set-Pi-pmf*:

assumes *finite A*
shows *set-pmf (Pi-pmf A dflt p) = PiE-dflt A dflt (set-pmf ∘ p)*
proof (*rule equalityI*)
show *PiE-dflt A dflt (set-pmf ∘ p) ⊆ set-pmf (Pi-pmf A dflt p)*
proof *safe*
fix *f* **assume** *f*: *f ∈ PiE-dflt A dflt (set-pmf ∘ p)*
hence *pmf (Pi-pmf A dflt p) f = (∏ x∈A. pmf (p x) (f x))*
using *assms* **by** (*auto simp*: *pmf-Pi PiE-dflt-def*)
also have *... > 0*
using *f* **by** (*intro prod-pos*) (*auto simp*: *PiE-dflt-def set-pmf-eq*)
finally show *f ∈ set-pmf (Pi-pmf A dflt p)*
by (*auto simp*: *set-pmf-eq*)
qed
qed (*use set-Pi-pmf-subset'*[*OF assms*, *of dflt p*] **in** *auto*)

The probability of an independent combination of events is precisely the product of the probabilities of each individual event.

lemma *measure-Pi-pmf-PiE-dflt*:
assumes [*simp*]: *finite A*
shows *measure-pmf.prob (Pi-pmf A dflt p) (PiE-dflt A dflt B) =*
(∏ x∈A. measure-pmf.prob (p x) (B x))
proof −
define *B'* **where** *B' = (λx. B x ∩ set-pmf (p x))*
have *measure-pmf.prob (Pi-pmf A dflt p) (PiE-dflt A dflt B) =*
(∑ a h∈PiE-dflt A dflt B. pmf (Pi-pmf A dflt p) h)
by (*rule measure-pmf-conv-infsetsum*)
also have *... = (∑ a h∈PiE-dflt A dflt B. ∏ x∈A. pmf (p x) (h x))*
by (*intro infsetsum-cong*, *subst pmf-Pi'*) (*auto simp*: *PiE-dflt-def*)
also have *... = (∑ a h∈(λh. restrict h A) ‘ PiE-dflt A dflt B. ∏ x∈A. pmf (p x) (h x))*
by (*subst infsetsum-reindex*) (*force simp*: *inj-on-def PiE-dflt-def fun-eq-iff*)+
also have *(λh. restrict h A) ‘ PiE-dflt A dflt B = PiE A B*
by (*rule restrict-PiE-dflt*)
also have *(∑ a h∈PiE A B. ∏ x∈A. pmf (p x) (h x)) = (∑ a h∈PiE A B'. ∏ x∈A. pmf (p x) (h x))*
by (*intro infsetsum-cong-neutral*) (*auto simp*: *B'-def set-pmf-eq*)
also have *(∑ a h∈PiE A B'. ∏ x∈A. pmf (p x) (h x)) = (∏ x∈A. infsetsum (pmf (p x)) (B' x))*
by (*intro infsetsum-prod-PiE*) (*auto simp*: *B'-def*)
also have *... = (∏ x∈A. infsetsum (pmf (p x)) (B x))*
by (*intro prod.cong infsetsum-cong-neutral*) (*auto simp*: *B'-def set-pmf-eq*)
also have *... = (∏ x∈A. measure-pmf.prob (p x) (B x))*
by (*subst measure-pmf-conv-infsetsum*) (*rule refl*)
finally show *?thesis* **.**
qed

lemma *measure-Pi-pmf-Pi*:
fixes *t*::*nat*
assumes [*simp*]: *finite A*

```
  shows   measure-pmf.prob (Pi-pmf A dflt p) (Pi A B) =
          (∏ x∈A. measure-pmf.prob (p x) (B x)) (is ?lhs = ?rhs)
proof −
  have ?lhs = measure-pmf.prob (Pi-pmf A dflt p) (PiE-dflt A dflt B)
    by (intro measure-prob-cong-0)
       (auto simp: PiE-dflt-def PiE-def intro!: pmf-Pi-outside)+
  also have ... = ?rhs
    using assms by (simp add: measure-Pi-pmf-PiE-dflt)
  finally show ?thesis
    by simp
qed
```

26.4 Common PMF operations on products

Pi-pmf distributes over the 'bind' operation in the Giry monad:

```
lemma Pi-pmf-bind:
  assumes finite A
  shows   Pi-pmf A d (λx. bind-pmf (p x) (q x)) =
          do {f ← Pi-pmf A d' p; Pi-pmf A d (λx. q x (f x))} (is ?lhs = ?rhs)
proof (rule pmf-eqI, goal-cases)
  case (1 f)
  show ?case
  proof (cases ∃ x∈−A. f x ≠ d)
    case False
    define B where B = (λx. set-pmf (p x))
    have [simp]: countable (B x) for x by (auto simp: B-def)

    {
      fix x :: 'a
      have (λa. pmf (p x) a * 1) abs-summable-on B x
        by (simp add: pmf-abs-summable)
      moreover have norm (pmf (p x) a * 1) ≥ norm (pmf (p x) a * pmf (q x
a) (f x)) for a
        unfolding norm-mult by (intro mult-left-mono) (auto simp: pmf-le-1)
      ultimately have (λa. pmf (p x) a * pmf (q x a) (f x)) abs-summable-on B x
        by (rule abs-summable-on-comparison-test)
    } note summable = this

    have pmf ?rhs f = (∑ a g. pmf (Pi-pmf A d' p) g * (∏ x∈A. pmf (q x (g x))
(f x)))
      by (subst pmf-bind, subst pmf-Pi')
         (insert assms False, simp-all add: pmf-expectation-eq-infsetsum)
    also have ... = (∑ a g∈PiE-dflt A d' B.
                       pmf (Pi-pmf A d' p) g * (∏ x∈A. pmf (q x (g x)) (f x)))
unfolding B-def
      using assms by (intro infsetsum-cong-neutral) (auto simp: pmf-Pi PiE-dflt-def
set-pmf-eq)
    also have ... = (∑ a g∈PiE-dflt A d' B.
                     (∏ x∈A. pmf (p x) (g x) * pmf (q x (g x)) (f x)))
```

```
    using assms by (intro infsetsum-cong) (auto simp: pmf-Pi PiE-dflt-def prod.distrib)
  also have ... = (∑a g∈(λg. restrict g A) ' PiE-dflt A d' B.
                    (∏x∈A. pmf (p x) (g x) * pmf (q x (g x)) (f x)))
    by (subst infsetsum-reindex) (force simp: PiE-dflt-def inj-on-def fun-eq-iff)+
  also have (λg. restrict g A) ' PiE-dflt A d' B = PiE A B
    by (rule restrict-PiE-dflt)
  also have (∑a g∈.... (∏x∈A. pmf (p x) (g x) * pmf (q x (g x)) (f x))) =
              (∏x∈A. ∑a a∈B x. pmf (p x) a * pmf (q x a) (f x))
    using assms summable by (subst infsetsum-prod-PiE) simp-all
  also have ... = (∏x∈A. ∑a a. pmf (p x) a * pmf (q x a) (f x))
    by (intro prod.cong infsetsum-cong-neutral) (auto simp: B-def set-pmf-eq)
  also have ... = pmf ?lhs f
  using False assms by (subst pmf-Pi') (simp-all add: pmf-bind pmf-expectation-eq-infsetsum)
  finally show ?thesis ..
 next
  case True
  have pmf ?rhs f =
        measure-pmf.expectation (Pi-pmf A d' p) (λx. pmf (Pi-pmf A d (λxa. q xa (x xa))) f)
    using assms by (simp add: pmf-bind)
  also have ... = measure-pmf.expectation (Pi-pmf A d' p) (λx. 0)
   using assms True by (intro Bochner-Integration.integral-cong pmf-Pi-outside) auto
  also have ... = pmf ?lhs f
    using assms True by (subst pmf-Pi-outside) auto
  finally show ?thesis ..
 qed
qed

lemma Pi-pmf-return-pmf [simp]:
 assumes finite A
 shows  Pi-pmf A dflt (λx. return-pmf (f x)) = return-pmf (λx. if x ∈ A then f x else dflt)
  using assms by (intro pmf-eqI) (auto simp: pmf-Pi simp: indicator-def split: if-splits)
```

Analogously any componentwise mapping can be pulled outside the product:

```
lemma Pi-pmf-map:
 assumes [simp]: finite A and f dflt = dflt'
 shows  Pi-pmf A dflt' (λx. map-pmf f (g x)) = map-pmf (λh. f ∘ h) (Pi-pmf A dflt g)
proof −
 have Pi-pmf A dflt' (λx. map-pmf f (g x)) =
       Pi-pmf A dflt' (λx. g x ⤜ (λx. return-pmf (f x)))
   using assms by (simp add: map-pmf-def Pi-pmf-bind)
 also have ... = Pi-pmf A dflt g ⤜ (λh. return-pmf (λx. if x ∈ A then f (h x) else dflt'))
  by (subst Pi-pmf-bind[where d' = dflt]) (auto simp: )
```

```
  also have ... = map-pmf (λh. f ∘ h) (Pi-pmf A dflt g)
    unfolding map-pmf-def using set-Pi-pmf-subset'[of A dflt g]
    by (intro bind-pmf-cong refl arg-cong[of - - return-pmf])
       (auto dest: simp: fun-eq-iff PiE-dflt-def assms(2))
  finally show ?thesis .
qed
```

We can exchange the default value in a product of PMFs like this:

```
lemma Pi-pmf-default-swap:
  assumes finite A
  shows   map-pmf (λf x. if x ∈ A then f x else dflt') (Pi-pmf A dflt p) =
          Pi-pmf A dflt' p (is ?lhs = ?rhs)
proof (rule pmf-eqI, goal-cases)
  case (1 f)
  let ?B = (λf x. if x ∈ A then f x else dflt') -` {f} ∩ PiE-dflt A dflt (λ-. UNIV)
  show ?case
  proof (cases ∃x∈-A. f x ≠ dflt')
    case False
    let ?f' = λx. if x ∈ A then f x else dflt
    from False have pmf ?lhs f = measure-pmf.prob (Pi-pmf A dflt p) ?B
      using assms unfolding pmf-map
      by (intro measure-prob-cong-0) (auto simp: PiE-dflt-def pmf-Pi-outside)
    also from False have ?B = {?f'}
      by (auto simp: fun-eq-iff PiE-dflt-def)
    also have measure-pmf.prob (Pi-pmf A dflt p) {?f'} = pmf (Pi-pmf A dflt p)
?f'
      by (simp add: measure-pmf-single)
    also have ... = pmf ?rhs f
      using False assms by (subst (1 2) pmf-Pi) auto
    finally show ?thesis .
  next
    case True
    have pmf ?lhs f = measure-pmf.prob (Pi-pmf A dflt p) ?B
      using assms unfolding pmf-map
      by (intro measure-prob-cong-0) (auto simp: PiE-dflt-def pmf-Pi-outside)
    also from True have ?B = {} by auto
    also have measure-pmf.prob (Pi-pmf A dflt p) ... = 0
      by simp
    also have 0 = pmf ?rhs f
      using True assms by (intro pmf-Pi-outside [symmetric]) auto
    finally show ?thesis .
  qed
qed
```

The following rule allows reindexing the product:

```
lemma Pi-pmf-bij-betw:
  assumes finite A bij-betw h A B ⋀x. x ∉ A ⟹ h x ∉ B
  shows Pi-pmf A dflt (λ-. f) = map-pmf (λg. g ∘ h) (Pi-pmf B dflt (λ-. f))
    (is ?lhs = ?rhs)
```

proof −
have *B*: *finite B*
using *assms bij-betw-finite* **by** *auto*
have *pmf ?lhs g* = *pmf ?rhs g* **for** *g*
proof (*cases* ∀ *a*. *a* ∉ *A* ⟶ *g a* = *dflt*)
case *True*
define *h'* **where** *h'* = *the-inv-into A h*
have *h'*: *h'* (*h x*) = *x* **if** *x* ∈ *A* **for** *x*
unfolding *h'-def* **using** *that assms* **by** (*auto simp add*: *bij-betw-def the-inv-into-f-f*)
have *h*: *h* (*h'* *x*) = *x* **if** *x* ∈ *B* **for** *x*
unfolding *h'-def* **using** *that assms f-the-inv-into-f-bij-betw* **by** *fastforce*
have *pmf ?rhs g* = *measure-pmf.prob* (*Pi-pmf B dflt* (λ-. *f*)) ((λ*g*. *g* ∘ *h*) −‘ {*g*})
unfolding *pmf-map* **by** *simp*
also have ... = *measure-pmf.prob* (*Pi-pmf B dflt* (λ-. *f*))
(((λ*g*. *g* ∘ *h*) −‘ {*g*}) ∩ *PiE-dflt B dflt* (λ-. *UNIV*))
using *B* **by** (*intro measure-prob-cong-0*) (*auto simp*: *PiE-dflt-def pmf-Pi-outside*)
also have ... = *pmf* (*Pi-pmf B dflt* (λ-. *f*)) (λ*x*. *if x* ∈ *B then g* (*h'* *x*) *else dflt*)
proof −
have (*if h x* ∈ *B then g* (*h'* (*h x*)) *else dflt*) = *g x* **for** *x*
using *h' assms True* **by** (*cases x* ∈ *A*) (*auto simp add*: *bij-betwE*)
then have (λ*g*. *g* ∘ *h*) −‘ {*g*} ∩ *PiE-dflt B dflt* (λ-. *UNIV*) =
{(λ*x*. *if x* ∈ *B then g* (*h'* *x*) *else dflt*)}
using *assms h' h True* **unfolding** *PiE-dflt-def* **by** *auto*
then show *?thesis*
by (*simp add*: *measure-pmf-single*)
qed
also have ... = *pmf* (*Pi-pmf A dflt* (λ-. *f*)) *g*
using *B assms True h'-def*
by (*auto simp add*: *pmf-Pi intro*!: *prod.reindex-bij-betw bij-betw-the-inv-into*)
finally show *?thesis*
by *simp*
next
case *False*
have *pmf ?rhs g* = *infsetsum* (*pmf* (*Pi-pmf B dflt* (λ-. *f*))) ((λ*g*. *g* ∘ *h*) −‘ {*g*})
using *assms* **by** (*auto simp add*: *measure-pmf-conv-infsetsum pmf-map*)
also have ... = *infsetsum* (λ-. *0*) ((λ*g x*. *g* (*h x*)) −‘ {*g*})
using *B False assms* **by** (*intro infsetsum-cong pmf-Pi-outside*) *fastforce*+
also have ... = *0*
by *simp*
finally show *?thesis*
using *assms False* **by** (*auto simp add*: *pmf-Pi pmf-map*)
qed
then show *?thesis*
by (*rule pmf-eqI*)
qed

A product of uniform random choices is again a uniform distribution.

lemma *Pi-pmf-of-set*:
assumes *finite A* $\bigwedge x.\ x \in A \Longrightarrow$ *finite* $(B\ x)$ $\bigwedge x.\ x \in A \Longrightarrow B\ x \neq \{\}$
shows *Pi-pmf A d* $(\lambda x.$ *pmf-of-set* $(B\ x)) =$ *pmf-of-set* $($*PiE-dflt A d B*$)$ (**is** *?lhs = ?rhs*)
proof (*rule pmf-eqI*, *goal-cases*)
case (*1 f*)
show *?case*
proof (*cases* $\exists x.\ x \notin A \wedge f\ x \neq d$)
case *True*
hence *pmf ?lhs f = 0*
using *assms* **by** (*intro pmf-Pi-outside*) (*auto simp*: *PiE-dflt-def*)
also from *True* **have** *f* $\notin$ *PiE-dflt A d B*
by (*auto simp*: *PiE-dflt-def*)
hence *0 = pmf ?rhs f*
using *assms* **by** (*subst pmf-of-set*) *auto*
finally show *?thesis* **.**
next
case *False*
hence *pmf ?lhs f* $= (\prod x{\in}A.$ *pmf* $($*pmf-of-set* $(B\ x))\ (f\ x))$
using *assms* **by** (*subst pmf-Pi'*) *auto*
also have $\ldots = (\prod x{\in}A.$ *indicator* $(B\ x)\ (f\ x)$ */ real* $($*card* $(B\ x)))$
by (*intro prod.cong refl*, *subst pmf-of-set*) (*use assms False* **in** *auto*)
also have $\ldots = (\prod x{\in}A.$ *indicator* $(B\ x)\ (f\ x))$ */ real* $(\prod x{\in}A.$ *card* $(B\ x))$
by (*subst prod-dividef*) *simp-all*
also have $(\prod x{\in}A.$ *indicator* $(B\ x)\ (f\ x) ::$ *real*$) =$ *indicator* $($*PiE-dflt A d B*$)\ f$
using *assms False* **by** (*auto simp*: *indicator-def PiE-dflt-def*)
also have $(\prod x{\in}A.$ *card* $(B\ x)) =$ *card* $($*PiE-dflt A d B*$)$
using *assms* **by** (*intro card-PiE-dflt* [*symmetric*]) *auto*
also have *indicator* $($*PiE-dflt A d B*$)\ f$ */* $\ldots =$ *pmf ?rhs f*
using *assms* **by** (*intro pmf-of-set* [*symmetric*]) *auto*
finally show *?thesis* **.**
qed
qed

26.5 Merging and splitting PMF products

The following lemma shows that we can add a single PMF to a product:

lemma *Pi-pmf-insert*:
assumes *finite A x* $\notin$ *A*
shows *Pi-pmf* $($*insert x A*$)$ *dflt p = map-pmf* $(\lambda(y,f).\ f(x{:=}y))$ $($*pair-pmf* $(p\ x)$ $($*Pi-pmf A dflt p*$))$
proof (*intro pmf-eqI*)
fix *f*
let *?M = pair-pmf* $(p\ x)$ $($*Pi-pmf A dflt p*$)$
have *pmf* $($*map-pmf* $(\lambda(y, f).\ f(x := y))$ *?M*$)\ f =$
measure-pmf.prob ?M $((\lambda(y, f).\ f(x := y)) -` \{f\})$
by (*subst pmf-map*) *auto*
also have $((\lambda(y, f).\ f(x := y)) -` \{f\}) = (\bigcup y'.\ \{(f\ x, f(x := y'))\})$
by (*auto simp*: *fun-upd-def fun-eq-iff*)

also have *measure-pmf.prob ?M ... = measure-pmf.prob ?M* $\{(f\,x, f(x := dflt))\}$
using *assms* **by** (*intro measure-prob-cong-0*) (*auto simp*: *pmf-pair pmf-Pi split*: *if-splits*)
also have ... = *pmf* (*p x*) (*f x*) ∗ *pmf* (*Pi-pmf A dflt p*) ($f(x := dflt)$)
by (*simp add*: *measure-pmf-single pmf-pair pmf-Pi*)
also have ... = *pmf* (*Pi-pmf* (*insert x A*) *dflt p*) *f*
proof (*cases* ∀ *y*. *y* ∉ *insert x A* ⟶ *f y* = *dflt*)
case *True*
with *assms* **have** *pmf* (*p x*) (*f x*) ∗ *pmf* (*Pi-pmf A dflt p*) ($f(x := dflt)$) =
pmf (*p x*) (*f x*) ∗ ($\prod$ *xa*∈*A*. *pmf* (*p xa*) (($f(x := dflt)$) *xa*))
by (*subst pmf-Pi'*) *auto*
also have ($\prod$ *xa*∈*A*. *pmf* (*p xa*) (($f(x := dflt)$) *xa*)) = ($\prod$ *xa*∈*A*. *pmf* (*p xa*) (*f xa*))
using *assms* **by** (*intro prod.cong*) *auto*
also have *pmf* (*p x*) (*f x*) ∗ ... = *pmf* (*Pi-pmf* (*insert x A*) *dflt p*) *f*
using *assms True* **by** (*subst pmf-Pi'*) *auto*
finally show *?thesis* **.**
qed (*insert assms*, *auto simp*: *pmf-Pi*)
finally show ... = *pmf* (*map-pmf* (λ(*y*, *f*). $f(x := y)$) *?M*) *f* **..**
qed

lemma *Pi-pmf-insert'*:
assumes *finite A* *x* ∉ *A*
shows *Pi-pmf* (*insert x A*) *dflt p* =
do {*y* ← *p x*; *f* ← *Pi-pmf A dflt p*; *return-pmf* ($f(x := y)$)}
using *assms*
by (*subst Pi-pmf-insert*)
(*auto simp add*: *map-pmf-def pair-pmf-def case-prod-beta' bind-return-pmf bind-assoc-pmf*)

lemma *Pi-pmf-singleton*:
Pi-pmf {*x*} *dflt p* = *map-pmf* (λ*a b*. *if b* = *x then a else dflt*) (*p x*)
proof −
have *Pi-pmf* {*x*} *dflt p* = *map-pmf* (*fun-upd* (λ-. *dflt*) *x*) (*p x*)
by (*subst Pi-pmf-insert*) (*simp-all add*: *pair-return-pmf2 pmf.map-comp o-def*)
also have *fun-upd* (λ-. *dflt*) *x* = (λ*z y*. *if y* = *x then z else dflt*)
by (*simp add*: *fun-upd-def fun-eq-iff*)
finally show *?thesis* **.**
qed

Projecting a product of PMFs onto a component yields the expected result:

lemma *Pi-pmf-component*:
assumes *finite A*
shows *map-pmf* (λ*f*. *f x*) (*Pi-pmf A dflt p*) = (*if x* ∈ *A then p x else return-pmf dflt*)
proof (*cases x* ∈ *A*)
case *True*
define *A'* **where** *A'* = *A* − {*x*}
from *assms* **and** *True* **have** *A'*: *A* = *insert x A'*

```
    by (auto simp: A'-def)
  from assms have map-pmf (λf. f x) (Pi-pmf A dflt p) = p x unfolding A'
    by (subst Pi-pmf-insert)
       (auto simp: A'-def pmf.map-comp o-def case-prod-unfold map-fst-pair-pmf)
  with True show ?thesis by simp
next
  case False
  have map-pmf (λf. f x) (Pi-pmf A dflt p) = map-pmf (λ-. dflt) (Pi-pmf A dflt
p)
    using assms False set-Pi-pmf-subset[of A dflt p]
    by (intro pmf.map-cong refl) (auto simp: set-pmf-eq pmf-Pi-outside)
  with False show ?thesis by simp
qed
```

We can take merge two PMF products on disjoint sets like this:

```
lemma Pi-pmf-union:
  assumes finite A finite B A ∩ B = {}
  shows   Pi-pmf (A ∪ B) dflt p =
          map-pmf (λ(f,g) x. if x ∈ A then f x else g x)
          (pair-pmf (Pi-pmf A dflt p) (Pi-pmf B dflt p)) (is - = map-pmf (?h A)
(?q A))
  using assms(1,3)
proof (induction rule: finite-induct)
  case (insert x A)
  have map-pmf (?h (insert x A)) (?q (insert x A)) =
        do {v ← p x; (f, g) ← pair-pmf (Pi-pmf A dflt p) (Pi-pmf B dflt p);
            return-pmf (λy. if y ∈ insert x A then (f(x := v)) y else g y)}
    by (subst Pi-pmf-insert)
       (insert insert.hyps insert.prems,
          simp-all add: pair-pmf-def map-bind-pmf bind-map-pmf bind-assoc-pmf
bind-return-pmf)
  also have ... = do {v ← p x; (f, g) ← ?q A; return-pmf ((?h A (f,g))(x := v))}
    by (intro bind-pmf-cong refl) (auto simp: fun-eq-iff)
  also have ... = do {v ← p x; f ← map-pmf (?h A) (?q A); return-pmf (f(x :=
v))}
    by (simp add: bind-map-pmf map-bind-pmf case-prod-unfold cong: if-cong)
  also have ... = do {v ← p x; f ← Pi-pmf (A ∪ B) dflt p; return-pmf (f(x :=
v))}
   using insert.hyps and insert.prems by (intro bind-pmf-cong insert.IH [symmetric]
refl) auto
  also have ... = Pi-pmf (insert x (A ∪ B)) dflt p
    by (subst Pi-pmf-insert)
      (insert assms insert.hyps insert.prems, auto simp: pair-pmf-def map-bind-pmf)
  also have insert x (A ∪ B) = insert x A ∪ B
    by simp
  finally show ?case ..
qed (simp-all add: case-prod-unfold map-snd-pair-pmf)
```

We can also project a product to a subset of the indices by mapping all the

other indices to the default value:

lemma *Pi-pmf-subset*:
assumes *finite A* $A' \subseteq A$
shows *Pi-pmf A' dflt p = map-pmf* ($\lambda f\ x.$ *if* $x \in A'$ *then f x else dflt*) (*Pi-pmf A dflt p*)
proof −
let *?P = pair-pmf* (*Pi-pmf A' dflt p*) (*Pi-pmf* ($A - A'$) *dflt p*)
from *assms* **have** [*simp*]: *finite A'*
by (*blast dest*: *finite-subset*)
from *assms* **have** $A = A' \cup (A - A')$
by *blast*
also have *Pi-pmf ... dflt p = map-pmf* ($\lambda(f,g)\ x.$ *if* $x \in A'$ *then f x else g x*) *?P*
using *assms* **by** (*intro Pi-pmf-union*) *auto*
also have *map-pmf* ($\lambda f\ x.$ *if* $x \in A'$ *then f x else dflt*) ... = *map-pmf fst ?P*
unfolding *map-pmf-comp o-def case-prod-unfold*
using *set-Pi-pmf-subset*[*of A' dflt p*] **by** (*intro map-pmf-cong refl*) (*auto simp*: *fun-eq-iff*)
also have ... = *Pi-pmf A' dflt p*
by (*simp add*: *map-fst-pair-pmf*)
finally show *?thesis* **..**
qed

lemma *Pi-pmf-subset'*:
fixes $f :: 'a \Rightarrow 'b$ *pmf*
assumes *finite A* $B \subseteq A$ $\bigwedge x.\ x \in A - B \Longrightarrow f\ x =$ *return-pmf dflt*
shows *Pi-pmf A dflt f = Pi-pmf B dflt f*
proof −
have *Pi-pmf* ($B \cup (A - B)$) *dflt f* =
map-pmf ($\lambda(f,\ g)\ x.$ *if* $x \in B$ *then f x else g x*)
(*pair-pmf* (*Pi-pmf B dflt f*) (*Pi-pmf* ($A - B$) *dflt f*))
using *assms* **by** (*intro Pi-pmf-union*) (*auto dest*: *finite-subset*)
also have *Pi-pmf* ($A - B$) *dflt f = Pi-pmf* ($A - B$) *dflt* (λ-. *return-pmf dflt*)
using *assms* **by** (*intro Pi-pmf-cong*) *auto*
also have ... = *return-pmf* (λ-. *dflt*)
using *assms* **by** *simp*
also have *map-pmf* ($\lambda(f,\ g)\ x.$ *if* $x \in B$ *then f x else g x*)
(*pair-pmf* (*Pi-pmf B dflt f*) (*return-pmf* (λ-. *dflt*))) =
map-pmf ($\lambda f\ x.$ *if* $x \in B$ *then f x else dflt*) (*Pi-pmf B dflt f*)
by (*simp add*: *map-pmf-def pair-pmf-def bind-assoc-pmf bind-return-pmf bind-return-pmf'*)
also have ... = *Pi-pmf B dflt f*
using *assms* **by** (*intro Pi-pmf-default-swap*) (*auto dest*: *finite-subset*)
also have $B \cup (A - B) = A$
using *assms* **by** *auto*
finally show *?thesis* **.**
qed

lemma *Pi-pmf-if-set*:
assumes *finite A*
shows *Pi-pmf A dflt* ($\lambda x.$ *if b x then f x else return-pmf dflt*) =

Pi-pmf {*x*∈*A*. *b x*} *dflt f*
proof −
have *Pi-pmf A dflt* (λ*x*. *if b x then f x else return-pmf dflt*) =
Pi-pmf {*x*∈*A*. *b x*} *dflt* (λ*x*. *if b x then f x else return-pmf dflt*)
using *assms* **by** (*intro Pi-pmf-subset'*) *auto*
also have ... = *Pi-pmf* {*x*∈*A*. *b x*} *dflt f*
by (*intro Pi-pmf-cong*) *auto*
finally show *?thesis* **.**
qed

lemma *Pi-pmf-if-set'*:
assumes *finite A*
shows *Pi-pmf A dflt* (λ*x*. *if b x then return-pmf dflt else f x*) =
Pi-pmf {*x*∈*A*. ¬*b x*} *dflt f*
proof −
have *Pi-pmf A dflt* (λ*x*. *if b x then return-pmf dflt else f x*) =
Pi-pmf {*x*∈*A*. ¬*b x*} *dflt* (λ*x*. *if b x then return-pmf dflt else f x*)
using *assms* **by** (*intro Pi-pmf-subset'*) *auto*
also have ... = *Pi-pmf* {*x*∈*A*. ¬*b x*} *dflt f*
by (*intro Pi-pmf-cong*) *auto*
finally show *?thesis* **.**
qed

Lastly, we can delete a single component from a product:

lemma *Pi-pmf-remove*:
assumes *finite A*
shows *Pi-pmf* (*A* − {*x*}) *dflt p* = *map-pmf* (λ*f*. *f*(*x* := *dflt*)) (*Pi-pmf A dflt p*)
proof −
have *Pi-pmf* (*A* − {*x*}) *dflt p* =
map-pmf (λ*f xa*. *if xa* ∈ *A* − {*x*} *then f xa else dflt*) (*Pi-pmf A dflt p*)
using *assms* **by** (*intro Pi-pmf-subset*) *auto*
also have ... = *map-pmf* (λ*f*. *f*(*x* := *dflt*)) (*Pi-pmf A dflt p*)
using *set-Pi-pmf-subset*[*of A dflt p*] *assms*
by (*intro map-pmf-cong refl*) (*auto simp*: *fun-eq-iff*)
finally show *?thesis* **.**
qed

26.6 Additional properties

lemma *nn-integral-prod-Pi-pmf*:
assumes *finite A*
shows *nn-integral* (*Pi-pmf A dflt p*) (λ*y*. ∏ *x*∈*A*. *f x* (*y x*)) = (∏ *x*∈*A*. *nn-integral* (*p x*) (*f x*))
using *assms*
proof (*induction rule*: *finite-induct*)
case (*insert x A*)
have *nn-integral* (*Pi-pmf* (*insert x A*) *dflt p*) (λ*y*. ∏ *z*∈*insert x A*. *f z* (*y z*)) =
(∫⁺*a*. ∫⁺*b*. *f x a* ∗ (∏ *z*∈*A*. *f z* (*if z* = *x then a else b z*)) ∂*Pi-pmf A dflt*

p ∂p x)
using *insert* **by** (*auto simp*: *Pi-pmf-insert case-prod-unfold nn-integral-pair-pmf' cong*: *if-cong*)
also have (*λa b*. ∏ *z*∈*A*. *f z* (*if z* = *x then a else b z*)) = (*λa b*. ∏ *z*∈*A*. *f z* (*b z*))
by (*intro ext prod.cong*) (*use insert.hyps* **in** *auto*)
also have (∫⁺*a*. ∫⁺*b*. *f x a* ∗ (∏ *z*∈*A*. *f z* (*b z*)) *∂Pi-pmf A dflt p ∂p x*) =
(∫⁺*y*. *f x y ∂*(*p x*)) ∗ (∫⁺*y*. (∏ *z*∈*A*. *f z* (*y z*)) *∂*(*Pi-pmf A dflt p*))
by (*simp add*: *nn-integral-multc nn-integral-cmult*)
also have (∫⁺*y*. (∏ *z*∈*A*. *f z* (*y z*)) *∂*(*Pi-pmf A dflt p*)) = (∏ *x*∈*A*. *nn-integral* (*p x*) (*f x*))
by (*rule insert.IH*)
also have (∫⁺*y*. *f x y ∂*(*p x*)) ∗ ... = (∏ *x*∈*insert x A*. *nn-integral* (*p x*) (*f x*))
using *insert.hyps* **by** *simp*
finally show *?case* **.**
qed *auto*

lemma *integrable-prod-Pi-pmf*:
fixes *f* :: *'a* ⇒ *'b* ⇒ *'c* :: {*real-normed-field*, *second-countable-topology*, *banach*}
assumes *finite A* **and** ⋀*x*. *x* ∈ *A* ⟹ *integrable* (*measure-pmf* (*p x*)) (*f x*)
shows *integrable* (*measure-pmf* (*Pi-pmf A dflt p*)) (*λh*. ∏ *x*∈*A*. *f x* (*h x*))
proof (*intro integrableI-bounded*)
have (∫⁺ *x*. *ennreal* (*norm* (∏ *xa*∈*A*. *f xa* (*x xa*))) *∂measure-pmf* (*Pi-pmf A dflt p*)) =
(∫⁺ *x*. (∏ *y*∈*A*. *ennreal* (*norm* (*f y* (*x y*)))) *∂measure-pmf* (*Pi-pmf A dflt p*))
by (*simp flip*: *prod-norm prod-ennreal*)
also have ... = (∏ *x*∈*A*. ∫⁺ *a*. *ennreal* (*norm* (*f x a*)) *∂measure-pmf* (*p x*))
by (*intro nn-integral-prod-Pi-pmf*) *fact*
also have (∫⁺*a*. *ennreal* (*norm* (*f i a*)) *∂measure-pmf* (*p i*)) ≠ *top* **if** *i*: *i* ∈ *A* **for** *i*
using *assms*(*2*)[*OF i*] **by** (*simp add*: *integrable-iff-bounded*)
hence (∏ *x*∈*A*. ∫⁺ *a*. *ennreal* (*norm* (*f x a*)) *∂measure-pmf* (*p x*)) ≠ *top*
by (*subst ennreal-prod-eq-top*) *auto*
finally show (∫⁺ *x*. *ennreal* (*norm* (∏ *xa*∈*A*. *f xa* (*x xa*))) *∂measure-pmf* (*Pi-pmf A dflt p*)) < ∞
by (*simp add*: *top.not-eq-extremum*)
qed *auto*

lemma *expectation-prod-Pi-pmf*:
fixes *f* :: - ⇒ - ⇒ *real*
assumes *finite A*
assumes ⋀*x*. *x* ∈ *A* ⟹ *integrable* (*measure-pmf* (*p x*)) (*f x*)
assumes ⋀*x y*. *x* ∈ *A* ⟹ *y* ∈ *set-pmf* (*p x*) ⟹ *f x y* ≥ *0*
shows *measure-pmf.expectation* (*Pi-pmf A dflt p*) (*λy*. ∏ *x*∈*A*. *f x* (*y x*)) =
(∏ *x*∈*A*. *measure-pmf.expectation* (*p x*) (*λv*. *f x v*))
proof −
have *nonneg*: *measure-pmf.expectation* (*p x*) (*f x*) ≥ *0* **if** *x* ∈ *A* **for** *x*
using *that* **by** (*intro Bochner-Integration.integral-nonneg-AE AE-pmfI assms*)

```
  have nonneg': 0 ≤ measure-pmf.expectation (Pi-pmf A dflt p) (λy. ∏ x∈A. f x
(y x))
    by (intro Bochner-Integration.integral-nonneg-AE AE-pmfI assms prod-nonneg)
       (use assms in ‹auto simp: set-Pi-pmf PiE-dflt-def›)

  have ennreal (measure-pmf.expectation (Pi-pmf A dflt p) (λy. ∏ x∈A. f x (y x)))
=
         nn-integral (Pi-pmf A dflt p) (λy. ennreal (∏ x∈A. f x (y x))) using assms
    by (intro nn-integral-eq-integral [symmetric] assms integrable-prod-Pi-pmf)
       (auto simp: AE-measure-pmf-iff set-Pi-pmf PiE-dflt-def prod-nonneg)
  also have ... = nn-integral (Pi-pmf A dflt p) (λy. (∏ x∈A. ennreal (f x (y x))))
    by (intro nn-integral-cong-AE AE-pmfI prod-ennreal [symmetric])
       (use assms(1) in ‹auto simp: set-Pi-pmf PiE-dflt-def intro!: assms(3)›)
  also have ... = (∏ x∈A. ∫⁺ a. ennreal (f x a) ∂measure-pmf (p x))
    by (rule nn-integral-prod-Pi-pmf) fact+
  also have ... = (∏ x∈A. ennreal (measure-pmf.expectation (p x) (f x)))
    by (intro prod.cong nn-integral-eq-integral assms AE-pmfI) auto
  also have ... = ennreal (∏ x∈A. measure-pmf.expectation (p x) (f x))
    by (intro prod-ennreal nonneg)
  finally show ?thesis
    using nonneg nonneg' by (subst (asm) ennreal-inj) (auto intro!: prod-nonneg)
qed

lemma indep-vars-Pi-pmf:
  assumes fin: finite I
  shows   prob-space.indep-vars (measure-pmf (Pi-pmf I dflt p))
            (λ-. count-space UNIV) (λx f. f x) I
proof (cases I = {})
  case True
  show ?thesis
    by (subst prob-space.indep-vars-def [OF measure-pmf.prob-space-axioms],
      subst prob-space.indep-sets-def [OF measure-pmf.prob-space-axioms]) (simp-all
add: True)
next
  case [simp]: False
  show ?thesis
  proof (subst prob-space.indep-vars-iff-distr-eq-PiM')
    show distr (measure-pmf (Pi-pmf I dflt p)) (Pi_M I (λi. count-space UNIV))
(λx. restrict x I) =
         Pi_M I (λi. distr (measure-pmf (Pi-pmf I dflt p)) (count-space UNIV) (λf.
f i))
    proof (rule product-sigma-finite.PiM-eqI, goal-cases)
      case 1
        interpret product-prob-space λi. distr (measure-pmf (Pi-pmf I dflt p))
(count-space UNIV) (λf. f i)
      by (intro product-prob-spaceI prob-space.prob-space-distr measure-pmf.prob-space-axioms)
          simp-all
      show ?case by unfold-locales
    next
```

```
    case 3
     have sets (Pi_M I (λi. distr (measure-pmf (Pi-pmf I dflt p)) (count-space
UNIV) (λf. f i))) =
           sets (Pi_M I (λ-. count-space UNIV))
      by (intro sets-PiM-cong) simp-all
    thus ?case by simp
  next
    case (4 A)
    have Pi_E I A ∈ sets (Pi_M I (λi. count-space UNIV))
      using 4 by (intro sets-PiM-I-finite fin) auto
   hence emeasure (distr (measure-pmf (Pi-pmf I dflt p)) (Pi_M I (λi. count-space
UNIV))
            (λx. restrict x I)) (Pi_E I A) =
          emeasure (measure-pmf (Pi-pmf I dflt p)) ((λx. restrict x I) -` Pi_E I A)
      using 4 by (subst emeasure-distr) (auto simp: space-PiM)
    also have ... = emeasure (measure-pmf (Pi-pmf I dflt p)) (PiE-dflt I dflt A)
     by (intro emeasure-eq-AE AE-pmfI) (auto simp: PiE-dflt-def set-Pi-pmf fin)
    also have ... = (∏ i∈I. emeasure (measure-pmf (p i)) (A i))
      by (simp add: measure-pmf.emeasure-eq-measure measure-Pi-pmf-PiE-dflt
fin prod-ennreal)
    also have ... = (∏ i∈I. emeasure (measure-pmf (map-pmf (λf. f i) (Pi-pmf
I dflt p))) (A i))
      by (intro prod.cong refl, subst Pi-pmf-component) (auto simp: fin)
    finally show ?case
      by (simp add: map-pmf-rep-eq)
  qed fact+
 qed (simp-all add: measure-pmf.prob-space-axioms)
qed

lemma
 fixes h :: 'a :: comm-monoid-add ⇒ 'b::{banach, second-countable-topology}
 assumes fin: finite I
 assumes integrable: ⋀i. i ∈ I ⟹ integrable (measure-pmf (D i)) h
 shows   integrable-sum-Pi-pmf: integrable (Pi-pmf I dflt D) (λg. ∑ i∈I. h (g i))
  and    expectation-sum-Pi-pmf:
          measure-pmf.expectation (Pi-pmf I dflt D) (λg. ∑ i∈I. h (g i)) =
          (∑ i∈I. measure-pmf.expectation (D i) h)
proof −
 have integrable': integrable (Pi-pmf I dflt D) (λg. h (g i)) if i: i ∈ I for i
 proof −
  have integrable (D i) h
    using i by (rule assms)
  also have D i = map-pmf (λg. g i) (Pi-pmf I dflt D)
    by (subst Pi-pmf-component) (use fin i in auto)
  finally show integrable (measure-pmf (Pi-pmf I dflt D)) (λx. h (x i))
    by simp
 qed
 thus integrable (Pi-pmf I dflt D) (λg. ∑ i∈I. h (g i))
  by (intro Bochner-Integration.integrable-sum)
```

have *measure-pmf.expectation* (*Pi-pmf I dflt D*) (λ*x*. ∑ *i*∈*I*. *h* (*x i*)) =
(∑ *i*∈*I*. *measure-pmf.expectation* (*map-pmf* (λ*x*. *x i*) (*Pi-pmf I dflt D*)) *h*)
using *integrable'* **by** (*subst Bochner-Integration.integral-sum*) *auto*
also have ... = (∑ *i*∈*I*. *measure-pmf.expectation* (*D i*) *h*)
by (*intro sum.cong refl*, *subst Pi-pmf-component*) (*use fin* **in** *auto*)
finally show *measure-pmf.expectation* (*Pi-pmf I dflt D*) (λ*g*. ∑ *i*∈*I*. *h* (*g i*)) =
(∑ *i*∈*I*. *measure-pmf.expectation* (*D i*) *h*) **.**
qed

26.7 Applications

Choosing a subset of a set uniformly at random is equivalent to tossing a fair coin independently for each element and collecting all the elements that came up heads.

lemma *pmf-of-set-Pow-conv-bernoulli*:
assumes *finite* (*A* :: *'a set*)
shows *map-pmf* (λ*b*. {*x*∈*A*. *b x*}) (*Pi-pmf A P* (λ-. *bernoulli-pmf* (*1/2*))) = *pmf-of-set* (*Pow A*)
proof −
have *Pi-pmf A P* (λ-. *bernoulli-pmf* (*1/2*)) = *pmf-of-set* (*PiE-dflt A P* (λ*x*. *UNIV*))
using *assms* **by** (*simp add*: *bernoulli-pmf-half-conv-pmf-of-set Pi-pmf-of-set*)
also have *map-pmf* (λ*b*. {*x*∈*A*. *b x*}) ... = *pmf-of-set* (*Pow A*)
proof −
have *bij-betw* (λ*b*. {*x* ∈ *A*. *b x*}) (*PiE-dflt A P* (λ-. *UNIV*)) (*Pow A*)
by (*rule bij-betwI*[*of - - - λB b. if b* ∈ *A then b* ∈ *B else P*]) (*auto simp add*: *PiE-dflt-def*)
then show *?thesis*
using *assms* **by** (*intro map-pmf-of-set-bij-betw*) *auto*
qed
finally show *?thesis*
by *simp*
qed

A binomial distribution can be seen as the number of successes in n independent coin tosses.

lemma *binomial-pmf-altdef'*:
fixes *A* :: *'a set*
assumes *finite A* **and** *card A* = *n* **and** *p*: *p* ∈ {*0..1*}
shows *binomial-pmf n p* =
map-pmf (λ*f*. *card* {*x*∈*A*. *f x*}) (*Pi-pmf A dflt* (λ-. *bernoulli-pmf p*)) (**is** *?lhs* = *?rhs*)
proof −
from *assms* **have** *?lhs* = *binomial-pmf* (*card A*) *p*
by *simp*
also have ... = *?rhs*

```
  using assms(1)
  proof (induction rule: finite-induct)
    case empty
    with p show ?case by (simp add: binomial-pmf-0)
  next
    case (insert x A)
    from insert.hyps have card (insert x A) = Suc (card A)
      by simp
    also have binomial-pmf ... p = do {
                                     b ← bernoulli-pmf p;
                                     f ← Pi-pmf A dflt (λ-. bernoulli-pmf p);
                                    return-pmf ((if b then 1 else 0) + card {y ∈ A. f y})
                                    }
      using p by (simp add: binomial-pmf-Suc insert.IH bind-map-pmf)
    also have ... = do {
                      b ← bernoulli-pmf p;
                      f ← Pi-pmf A dflt (λ-. bernoulli-pmf p);
                      return-pmf (card {y ∈ insert x A. (f(x := b)) y})
                    }
    proof (intro bind-pmf-cong refl, goal-cases)
      case (1 b f)
      have (if b then 1 else 0) + card {y∈A. f y} = card ((if b then {x} else {}) ∪
{y∈A. f y})
        using insert.hyps by auto
      also have (if b then {x} else {}) ∪ {y∈A. f y} = {y∈insert x A. (f(x := b))
y}
        using insert.hyps by auto
      finally show ?case by simp
    qed
    also have ... = map-pmf (λf. card {y∈insert x A. f y})
                    (Pi-pmf (insert x A) dflt (λ-. bernoulli-pmf p))
    using insert.hyps by (subst Pi-pmf-insert) (simp-all add: pair-pmf-def map-bind-pmf)
    finally show ?case .
  qed
  finally show ?thesis .
qed

end
```

27 Hoeffding's Lemma and Hoeffding's Inequality

```
theory Hoeffding
  imports Product-PMF Independent-Family
begin
```

Hoeffding's inequality shows that a sum of bounded independent random variables is concentrated around its mean, with an exponential decay of the tail probabilities.

27.1 Hoeffding's Lemma

lemma *convex-on-exp*:
  fixes $l :: real$
  assumes $l \geq 0$
  shows *convex-on UNIV* $(\lambda x.\ exp(l*x))$
  using *assms*
  by (*intro convex-on-realI*[**where** $f' = \lambda x.\ l * exp\ (l * x)$])
    (*auto intro*!: *derivative-eq-intros mult-left-mono*)

lemma *mult-const-minus-self-real-le*:
  fixes $x :: real$
  shows $x * (c - x) \leq c^2\ /\ 4$
proof −
  have $x * (c - x) = -(x - c\ /\ 2)^2 + c^2\ /\ 4$
    by (*simp add*: *field-simps power2-eq-square*)
  also have $\ldots \leq 0 + c^2\ /\ 4$
    by (*intro add-mono*) *auto*
  finally show *?thesis* **by** *simp*
qed

lemma *Hoeffdings-lemma-aux*:
  fixes $h\ p :: real$
  assumes $h \geq 0$ **and** $p \geq 0$
  defines $L \equiv (\lambda h.\ -h * p + ln\ (1 + p * (exp\ h - 1)))$
  shows $L\ h \leq h^2\ /\ 8$
proof (*cases* $h = 0$)
  case *False*
  hence *h*: $h > 0$
    using ‹$h \geq 0$› **by** *simp*
  define L' **where** $L' = (\lambda h.\ -p + p * exp\ h\ /\ (1 + p * (exp\ h - 1)))$
  define L'' **where** $L'' = (\lambda h.\ -(p^2) * exp\ h * exp\ h\ /\ (1 + p * (exp\ h - 1))^2 +$
                $p * exp\ h\ /\ (1 + p * (exp\ h - 1)))$
  define *Ls* **where** $Ls = (\lambda n.\ [L,\ L',\ L'']\ !\ n)$

  have [*simp*]: $L\ 0 = 0\ L'\ 0 = 0$
    by (*auto simp*: *L-def L'-def*)

  have L': $(L\ has\text{-}real\text{-}derivative\ L'\ x)\ (at\ x)$ **if** $x \in \{0..h\}$ **for** x
  proof −
    have $1 + p * (exp\ x - 1) > 0$
      using ‹$p \geq 0$› *that* **by** (*intro add-pos-nonneg mult-nonneg-nonneg*) *auto*
    thus *?thesis*
      unfolding *L-def L'-def* **by** (*auto intro*!: *derivative-eq-intros*)
  qed

  have L'': $(L'\ has\text{-}real\text{-}derivative\ L''\ x)\ (at\ x)$ **if** $x \in \{0..h\}$ **for** x
  proof −
    have ∗: $1 + p * (exp\ x - 1) > 0$
      using ‹$p \geq 0$› *that* **by** (*intro add-pos-nonneg mult-nonneg-nonneg*) *auto*

show *?thesis*
unfolding *L′-def L″-def*
by (*insert* ∗, (*rule derivative-eq-intros refl* | *simp*)+) (*auto simp*: *divide-simps*; *algebra*)
qed

have *diff*: ∀ *m t*. *m* < *2* ∧ *0* ≤ *t* ∧ *t* ≤ *h* ⟶ (*Ls m has-real-derivative Ls* (*Suc m*) *t*) (*at t*)
using *L′ L″* **by** (*auto simp*: *Ls-def nth-Cons split*: *nat.splits*)
from *Taylor*[*of 2 Ls L 0 h 0 h*, *OF - - diff*]
obtain *t* **where** *t*: *t* ∈ {*0*<..<*h*} *L h* = *L″ t* ∗ *h*² / *2*
using ‹*h* > *0*› **by** (*auto simp*: *Ls-def lessThan-nat-numeral*)
define *u* **where** *u* = *p* ∗ *exp t* / (*1* + *p* ∗ (*exp t* − *1*))

have *L″ t* = *u* ∗ (*1* − *u*)
by (*simp add*: *L″-def u-def divide-simps*; *algebra*)
also have ... ≤ *1* / *4*
using *mult-const-minus-self-real-le*[*of u 1*] **by** *simp*
finally have *L″ t* ≤ *1* / *4* **.**

note *t*(*2*)
also have *L″ t* ∗ *h*² / *2* ≤ (*1* / *4*) ∗ *h*² / *2*
using ‹*L″ t* ≤ *1* / *4*› **by** (*intro mult-right-mono divide-right-mono*) *auto*
finally show *L h* ≤ *h*² / *8* **by** *simp*
qed (*auto simp*: *L-def*)

locale *interval-bounded-random-variable* = *prob-space* +
fixes *f* :: *'a* ⇒ *real* **and** *a b* :: *real*
assumes *random-variable* [*measurable*]: *random-variable borel f*
assumes *AE-in-interval*: *AE x in M*. *f x* ∈ {*a*..*b*}
begin

lemma *integrable* [*intro*]: *integrable M f*
proof (*rule integrable-const-bound*)
show *AE x in M*. *norm* (*f x*) ≤ *max* |*a*| |*b*|
by (*intro eventually-mono*[*OF AE-in-interval*]) *auto*
qed (*fact random-variable*)

We first show Hoeffding's lemma for distributions whose expectation is 0. The general case will easily follow from this later.

lemma *Hoeffdings-lemma-nn-integral-0*:
assumes *l* > *0* **and** *E0*: *expectation f* = *0*
shows *nn-integral M* (λ*x*. *exp* (*l* ∗ *f x*)) ≤ *ennreal* (*exp* (*l*² ∗ (*b* − *a*)² / *8*))
proof (*cases AE x in M*. *f x* = *0*)
case *True*
hence *nn-integral M* (λ*x*. *exp* (*l* ∗ *f x*)) = *nn-integral M* (λ*x*. *ennreal 1*)
by (*intro nn-integral-cong-AE*) *auto*
also have ... = *ennreal* (*expectation* (λ*-*. *1*))

```
    by (intro nn-integral-eq-integral) auto
  finally show ?thesis by (simp add: prob-space)
next
  case False
  have a < 0
  proof (rule ccontr)
    assume a: ¬(a < 0)
    have AE x in M. f x = 0
    proof (subst integral-nonneg-eq-0-iff-AE [symmetric])
      show AE x in M. f x ≥ 0
        using AE-in-interval by eventually-elim (use a in auto)
    qed (use E0 in ‹auto simp: id-def integrable›)
    with False show False by contradiction
  qed

  have b > 0
  proof (rule ccontr)
    assume b: ¬(b > 0)
    have AE x in M. −f x = 0
    proof (subst integral-nonneg-eq-0-iff-AE [symmetric])
      show AE x in M. −f x ≥ 0
        using AE-in-interval by eventually-elim (use b in auto)
    qed (use E0 in ‹auto simp: id-def integrable›)
    with False show False by simp
  qed

  have a < b
    using ‹a < 0› ‹b > 0› by linarith

  define p where p = −a / (b − a)
  define L where L = (λt. −t∗ p + ln (1 − p + p ∗ exp t))
  define z where z = l ∗ (b − a)
  have z > 0
    unfolding z-def using ‹a < b› ‹l > 0› by auto
  have p > 0
    using ‹a < 0› ‹a < b› unfolding p-def by (intro divide-pos-pos) auto

  have (∫+x. exp (l ∗ f x) ∂M) ≤
        (∫+x. (b − f x) / (b − a) ∗ exp (l ∗ a) + (f x − a) / (b − a) ∗ exp (l ∗ b)
∂M)
  proof (intro nn-integral-mono-AE eventually-mono[OF AE-in-interval] ennreal-leI)
    fix x assume x: f x ∈ {a..b}
    define y where y = (b − f x) / (b−a)
    have y: y ∈ {0..1}
      using x ‹a < b› by (auto simp: y-def)
    have conv: convex-on UNIV (λx. exp(l∗x))
      using ‹l > 0› by (intro convex-on-exp) auto
    have exp (l ∗ ((1 − y) ∗R b + y ∗R a)) ≤ (1 − y) ∗ exp (l ∗ b) + y ∗ exp (l
∗ a)
```

```
    using y ‹l > 0› by (intro convex-onD[OF convex-on-exp]) auto
  also have (1 − y) *R b + y *R a = f x
    using ‹a < b› by (simp add: y-def divide-simps) (simp add: algebra-simps)?
  also have 1 − y = (f x − a) / (b − a)
    using ‹a < b› by (simp add: field-simps y-def)
  finally show exp (l * f x) ≤ (b − f x) / (b − a) * exp (l*a) + (f x − a)/(b−a)
* exp (l*b)
    by (simp add: y-def)
 qed
 also have ... = (∫+x. ennreal (b − f x) * exp (l * a) / (b − a) +
                       ennreal (f x − a) * exp (l * b) / (b − a) ∂M)
  using ‹a < 0› ‹b > 0›
  by (intro nn-integral-cong-AE eventually-mono[OF AE-in-interval])
     (simp add: ennreal-plus ennreal-mult flip: divide-ennreal)
 also have ... = ((∫+ x. ennreal (b − f x) ∂M) * ennreal (exp (l * a)) +
                (∫+ x. ennreal (f x − a) ∂M) * ennreal (exp (l * b))) / ennreal (b
− a)
  by (simp add: nn-integral-add nn-integral-divide nn-integral-multc add-divide-distrib-ennreal)
 also have (∫+ x. ennreal (b − f x) ∂M) = ennreal (expectation (λx. b − f x))
  by (intro nn-integral-eq-integral Bochner-Integration.integrable-diff
          eventually-mono[OF AE-in-interval] integrable-const integrable) auto
 also have expectation (λx. b − f x) = b
  using assms by (subst Bochner-Integration.integral-diff) (auto simp: prob-space)
 also have (∫+ x. ennreal (f x − a) ∂M) = ennreal (expectation (λx. f x − a))
  by (intro nn-integral-eq-integral Bochner-Integration.integrable-diff
          eventually-mono[OF AE-in-interval] integrable-const integrable) auto
 also have expectation (λx. f x − a) = (−a)
  using assms by (subst Bochner-Integration.integral-diff) (auto simp: prob-space)
 also have (ennreal b * (exp (l * a)) + ennreal (−a) * (exp (l * b))) / (b − a) =
          ennreal (b * exp (l * a) − a * exp (l * b)) / ennreal (b − a)
  using ‹a < 0› ‹b > 0›
  by (simp flip: ennreal-mult ennreal-plus add: mult-nonpos-nonneg divide-ennreal
mult-mono)
 also have b * exp (l * a) − a * exp (l * b) = exp (L z) * (b − a)
 proof −
  have pos: 1 − p + p * exp z > 0
  proof −
   have exp z > 1 using ‹l > 0› and ‹a < b›
    by (subst one-less-exp-iff) (auto simp: z-def intro!: mult-pos-pos)
   hence (exp z − 1) * p ≥ 0
    unfolding p-def using ‹a < 0› and ‹a < b›
    by (intro mult-nonneg-nonneg divide-nonneg-pos) auto
   thus ?thesis
    by (simp add: algebra-simps)
  qed

  have exp (L z) * (b − a) = exp (−z * p) * (1 − p + p * exp z) * (b − a)
   using pos by (simp add: exp-add L-def exp-diff exp-minus divide-simps)
  also have ... = b * exp (l * a) − a * exp (l * b) using ‹a < b›
```

by (*simp add*: *p-def z-def divide-simps*) (*simp add*: *exp-diff algebra-simps*)*?*
finally show *?thesis* **by** *simp*
qed
also have *ennreal* (*exp* (*L z*) ∗ (*b* − *a*)) / *ennreal* (*b* − *a*) = *ennreal* (*exp* (*L z*))
using ‹*a* < *b*› **by** (*simp add*: *divide-ennreal*)
also have *L z* = −*z* ∗ *p* + *ln* (*1* + *p* ∗ (*exp z* − *1*))
by (*simp add*: *L-def algebra-simps*)
also have ... ≤ z^2 / *8*
unfolding *L-def* **by** (*rule Hoeffdings-lemma-aux*[**where** *p* = *p*]) (*use* ‹*z* > *0*› ‹*p* > *0*› **in** *simp-all*)
hence *ennreal* (*exp* (−*z* ∗ *p* + *ln* (*1* + *p* ∗ (*exp z* − *1*)))) ≤ *ennreal* (*exp* (z^2 / *8*))
by (*intro ennreal-leI*) *auto*
finally show *?thesis*
by (*simp add*: *z-def power-mult-distrib*)
qed

context
begin

interpretation *shift*: *interval-bounded-random-variable M* λ*x*. *f x* − *μ a* − *μ b* − *μ*
rewrites *b* − *μ* − (*a* − *μ*) ≡ *b* − *a*
by *unfold-locales* (*auto intro*!: *eventually-mono*[*OF AE-in-interval*])

lemma *expectation-shift*: *expectation* (λ*x*. *f x* − *expectation f*) = *0*
by (*subst Bochner-Integration.integral-diff*) (*auto simp*: *integrable prob-space*)

lemmas *Hoeffdings-lemma-nn-integral* = *shift.Hoeffdings-lemma-nn-integral-0*[*OF - expectation-shift*]

end

end

27.2 Hoeffding's Inequality

Consider n independent real random variables $X_1, \ldots, X_n$ that each almost surely lie in a compact interval $[a_i, b_i]$. Hoeffding's inequality states that the distribution of the sum of the X_i is tightly concentrated around the sum of the expected values: the probability of it being above or below the sum of the expected values by more than some ε decreases exponentially with ε.

locale *indep-interval-bounded-random-variables* = *prob-space* +
fixes *I* :: *'b set* **and** *X* :: *'b* ⇒ *'a* ⇒ *real*
fixes *a b* :: *'b* ⇒ *real*
assumes *fin*: *finite I*
assumes *indep*: *indep-vars* (λ-. *borel*) *X I*
assumes *AE-in-interval*: ⋀*i*. *i* ∈ *I* ⟹ *AE x in M*. *X i x* ∈ {*a i*..*b i*}

begin

lemma *random-variable* [*measurable*]:
assumes *i*: $i \in I$
shows *random-variable borel* $(X\ i)$
using *i indep* **unfolding** *indep-vars-def* **by** *blast*

lemma *bounded-random-variable* [*intro*]:
assumes *i*: $i \in I$
shows *interval-bounded-random-variable M* $(X\ i)$ $(a\ i)$ $(b\ i)$
by *unfold-locales* (*use AE-in-interval*[*OF i*] *i* **in** *auto*)

end

locale *Hoeffding-ineq* = *indep-interval-bounded-random-variables* +
fixes μ :: *real*
defines $\mu \equiv (\sum i{\in}I.\ expectation\ (X\ i))$
begin

theorem *Hoeffding-ineq-ge*:
assumes $\varepsilon \geq 0$
assumes $(\sum i{\in}I.\ (b\ i - a\ i)^2) > 0$
shows *prob* $\{x{\in}space\ M.\ (\sum i{\in}I.\ X\ i\ x) \geq \mu + \varepsilon\} \leq exp\ (-2 * \varepsilon^2\ /\ (\sum i{\in}I.\ (b\ i - a\ i)^2))$
proof (*cases* $\varepsilon = 0$)
case [*simp*]: *True*
have *prob* $\{x{\in}space\ M.\ (\sum i{\in}I.\ X\ i\ x) \geq \mu + \varepsilon\} \leq 1$
by *simp*
thus *?thesis* **by** *simp*
next
case *False*
with ‹$\varepsilon \geq 0$› **have** ε: $\varepsilon > 0$
by *auto*

define *d* **where** $d = (\sum i{\in}I.\ (b\ i - a\ i)^2)$
define *l* :: *real* **where** $l = 4 * \varepsilon\ /\ d$
have *d*: $d > 0$
using *assms* **by** (*simp add*: *d-def*)
have *l*: $l > 0$
using ε *d* **by** (*simp add*: *l-def*)
define μ' **where** $\mu' = (\lambda i.\ expectation\ (X\ i))$

have $\{x{\in}space\ M.\ (\sum i{\in}I.\ X\ i\ x) \geq \mu + \varepsilon\} = \{x{\in}space\ M.\ (\sum i{\in}I.\ X\ i\ x) - \mu \geq \varepsilon\}$
by (*simp add*: *algebra-simps*)
hence *ennreal* (*prob* $\{x{\in}space\ M.\ (\sum i{\in}I.\ X\ i\ x) \geq \mu + \varepsilon\}$) = *emeasure M* ...
by (*simp add*: *emeasure-eq-measure*)
also have ... $\leq$ *ennreal* $(exp\ (-l{*}\varepsilon)) * (\int^+ x{\in}space\ M.\ exp\ (l * ((\sum i{\in}I.\ X\ i$

$x) - \mu)) \ \partial M)$
by (*intro Chernoff-ineq-nn-integral-ge l*) *auto*
also have $(\lambda x.\ (\sum i{\in}I.\ X\ i\ x) - \mu) = (\lambda x.\ (\sum i{\in}I.\ X\ i\ x - \mu'\ i))$
by (*simp add*: *μ-def sum-subtractf μ'-def*)
also have $(\int^+ x{\in}space\ M.\ exp\ (l * ((\sum i{\in}I.\ X\ i\ x - \mu'\ i)))\ \partial M) =$
$(\int^+ x.\ (\prod i{\in}I.\ ennreal\ (exp\ (l * (X\ i\ x - \mu'\ i))))\ \partial M)$
by (*intro nn-integral-cong*)
(*simp-all add*: *sum-distrib-left ring-distribs exp-diff exp-sum fin prod-ennreal*)
also have $\ldots = (\prod i{\in}I.\ \int^+ x.\ ennreal\ (exp\ (l * (X\ i\ x - \mu'\ i)))\ \partial M)$
by (*intro indep-vars-nn-integral fin indep-vars-compose2*[*OF indep*]) *auto*
also have $ennreal\ (exp\ (-l * \varepsilon)) * \ldots \leq$
$ennreal\ (exp\ (-l * \varepsilon)) * (\prod i{\in}I.\ ennreal\ (exp\ (l^2 * (b\ i - a\ i)^2\ /\ 8)))$
proof (*intro mult-left-mono prod-mono-ennreal*)
fix *i* **assume** *i*: $i \in I$
from *i* **interpret** *interval-bounded-random-variable M X i a i b i* **..**
show $(\int^+ x.\ ennreal\ (exp\ (l * (X\ i\ x - \mu'\ i)))\ \partial M) \leq ennreal\ (exp\ (l^2 * (b\ i - a\ i)^2\ /\ 8))$
unfolding *μ'-def* **by** (*rule Hoeffdings-lemma-nn-integral*) *fact+*
qed *auto*
also have $\ldots = ennreal\ (exp\ (-l{*}\varepsilon) * (\prod i{\in}I.\ exp\ (l^2 * (b\ i - a\ i)^2\ /\ 8)))$
by (*simp add*: *prod-ennreal prod-nonneg flip*: *ennreal-mult*)
also have $exp\ (-l{*}\varepsilon) * (\prod i{\in}I.\ exp\ (l^2 * (b\ i - a\ i)^2\ /\ 8)) = exp\ (d * l^2\ /\ 8 - l * \varepsilon)$
by (*simp add*: *exp-diff exp-minus sum-divide-distrib sum-distrib-left*
sum-distrib-right exp-sum fin divide-simps mult-ac d-def)
also have $d * l^2\ /\ 8 - l * \varepsilon = -2 * \varepsilon^2\ /\ d$
using *d* **by** (*simp add*: *l-def field-simps power2-eq-square*)
finally show *?thesis*
by (*subst* (*asm*) *ennreal-le-iff*) (*simp-all add*: *d-def*)
qed

corollary *Hoeffding-ineq-le*:
assumes *ε*: $\varepsilon \geq 0$
assumes $(\sum i{\in}I.\ (b\ i - a\ i)^2) > 0$
shows $prob\ \{x{\in}space\ M.\ (\sum i{\in}I.\ X\ i\ x) \leq \mu - \varepsilon\} \leq exp\ (-2 * \varepsilon^2\ /\ (\sum i{\in}I.\ (b\ i - a\ i)^2))$
proof −
interpret *flip*: *Hoeffding-ineq M I* $\lambda i\ x.\ -X\ i\ x\ \lambda i.\ -b\ i\ \lambda i.\ -a\ i\ -\mu$
proof *unfold-locales*
fix *i* **assume** $i \in I$
then interpret *interval-bounded-random-variable M X i a i b i* **..**
show *AE x in M*. $- X\ i\ x \in \{-\ b\ i..-\ a\ i\}$
by (*intro eventually-mono*[*OF AE-in-interval*]) *auto*
qed (*auto simp*: *fin μ-def sum-negf intro*: *indep-vars-compose2*[*OF indep*])

have $prob\ \{x{\in}space\ M.\ (\sum i{\in}I.\ X\ i\ x) \leq \mu - \varepsilon\} = prob\ \{x{\in}space\ M.\ (\sum i{\in}I.\ -X\ i\ x) \geq -\mu + \varepsilon\}$
by (*simp add*: *sum-negf algebra-simps*)
also have $\ldots \leq exp\ (-\ 2 * \varepsilon^2\ /\ (\sum i{\in}I.\ (b\ i - a\ i)^2))$

using *flip.Hoeffding-ineq-ge*[*OF* ε] *assms*(*2*) **by** *simp*
finally show *?thesis* **.**
qed

corollary *Hoeffding-ineq-abs-ge*:
assumes ε: $\varepsilon \geq 0$
assumes $(\sum i{\in}I.\ (b\ i - a\ i)^2) > 0$
shows *prob* $\{x{\in}space\ M.\ |(\sum i{\in}I.\ X\ i\ x) - \mu| \geq \varepsilon\} \leq 2 * exp\ (-2 * \varepsilon^2\ /\ (\sum i{\in}I.\ (b\ i - a\ i)^2))$
proof −
have $\{x{\in}space\ M.\ |(\sum i{\in}I.\ X\ i\ x) - \mu| \geq \varepsilon\} =$
$\{x{\in}space\ M.\ (\sum i{\in}I.\ X\ i\ x) \geq \mu + \varepsilon\} \cup \{x{\in}space\ M.\ (\sum i{\in}I.\ X\ i\ x) \leq \mu - \varepsilon\}$
by *auto*
also have *prob* ... $\leq$ *prob* $\{x{\in}space\ M.\ (\sum i{\in}I.\ X\ i\ x) \geq \mu + \varepsilon\}$ +
prob $\{x{\in}space\ M.\ (\sum i{\in}I.\ X\ i\ x) \leq \mu - \varepsilon\}$
by (*intro measure-Un-le*) *auto*
also have ... $\leq$ *exp* $(-2 * \varepsilon^2\ /\ (\sum i{\in}I.\ (b\ i - a\ i)^2))$ + *exp* $(-2 * \varepsilon^2\ /\ (\sum i{\in}I.\ (b\ i - a\ i)^2))$
by (*intro add-mono Hoeffding-ineq-ge Hoeffding-ineq-le assms*)
finally show *?thesis* **by** *simp*
qed

end

27.3 Hoeffding's inequality for i.i.d. bounded random variables

If we have n even identically-distributed random variables, the statement of Hoeffding's lemma simplifies a bit more: it shows that the probability that the average of the X_i is more than ε above the expected value is no greater than $e^{\frac{-2ny^2}{(b-a)^2}}$.

This essentially gives us a more concrete version of the weak law of large numbers: the law states that the probability vanishes for $n \to \infty$ for any $\varepsilon > 0$. Unlike Hoeffding's inequality, it does not assume the variables to have bounded support, but it does not provide concrete bounds.

locale *iid-interval-bounded-random-variables* = *prob-space* +
fixes I :: *'b set* **and** X :: *'b* ⇒ *'a* ⇒ *real* **and** Y :: *'a* ⇒ *real*
fixes $a\ b$ :: *real*
assumes *fin*: *finite I*
assumes *indep*: *indep-vars* (λ-. *borel*) $X\ I$
assumes *distr-X*: $i \in I \Longrightarrow$ *distr M borel* $(X\ i)$ = *distr M borel Y*
assumes *rv-Y* [*measurable*]: *random-variable borel Y*
assumes *AE-in-interval*: *AE x in M.* $Y\ x \in \{a..b\}$
begin

lemma *random-variable* [*measurable*]:

assumes *i*: $i \in I$
shows *random-variable borel* $(X\ i)$
using *i indep* **unfolding** *indep-vars-def* **by** *blast*

sublocale *X*: *indep-interval-bounded-random-variables M I X* λ-. *a* λ-. *b*
proof
fix *i* **assume** *i*: $i \in I$
have *AE x in M*. $Y\ x \in \{a..b\}$
by (*fact AE-in-interval*)
also have *?this* $\longleftrightarrow$ (*AE x in distr M borel Y*. $x \in \{a..b\}$)
by (*subst AE-distr-iff*) *auto*
also have *distr M borel Y* = *distr M borel* $(X\ i)$
using *i* **by** (*simp add*: *distr-X*)
also have (*AE x in* .... $x \in \{a..b\}$) $\longleftrightarrow$ (*AE x in M*. $X\ i\ x \in \{a..b\}$)
using *i* **by** (*subst AE-distr-iff*) *auto*
finally show *AE x in M*. $X\ i\ x \in \{a..b\}$ **.**
qed (*simp-all add*: *fin indep*)

lemma *expectation-X* [*simp*]:
assumes *i*: $i \in I$
shows *expectation* $(X\ i)$ = *expectation Y*
proof −
have *expectation* $(X\ i)$ = *lebesgue-integral* (*distr M borel* $(X\ i)$) $(\lambda x.\ x)$
using *i* **by** (*intro integral-distr* [*symmetric*]) *auto*
also have *distr M borel* $(X\ i)$ = *distr M borel Y*
using *i* **by** (*rule distr-X*)
also have *lebesgue-integral* ... $(\lambda x.\ x)$ = *expectation Y*
by (*rule integral-distr*) *auto*
finally show *expectation* $(X\ i)$ = *expectation Y* **.**
qed

end

locale *Hoeffding-ineq-iid* = *iid-interval-bounded-random-variables* +
fixes μ :: *real*
defines $\mu \equiv$ *expectation Y*
begin

sublocale *X*: *Hoeffding-ineq M I X* λ-. *a* λ-. *b real* (*card I*) $*$ μ
by *unfold-locales* (*simp-all add*: μ-*def*)

corollary
assumes ε: $\varepsilon \geq 0$
assumes $a < b$ $I \neq \{\}$
defines $n \equiv$ *card I*
shows *Hoeffding-ineq-ge*:
prob $\{x \in$ *space M*. $(\sum i \in I.\ X\ i\ x) \geq n * \mu + \varepsilon\} \leq$
exp $(-2 * \varepsilon^2\ /\ (n * (b - a)^2))$ (**is** *?le*)

and *Hoeffding-ineq-le*:
prob $\{x \in space\ M.\ (\sum i \in I.\ X\ i\ x) \leq n * \mu - \varepsilon\} \leq$
exp $(-2 * \varepsilon^2 / (n * (b - a)^2))$ (**is** *?ge*)
and *Hoeffding-ineq-abs-ge*:
prob $\{x \in space\ M.\ |(\sum i \in I.\ X\ i\ x) - n * \mu| \geq \varepsilon\} \leq$
$2 *$ *exp* $(-2 * \varepsilon^2 / (n * (b - a)^2))$ (**is** *?abs-ge*)
proof −
have *pos*: $(\sum i \in I.\ (b - a)^2) > 0$
using ‹$a < b$› ‹$I \neq \{\}$› *fin* **by** (*intro sum-pos*) *auto*
show *?le*
using *X.Hoeffding-ineq-ge*[*OF* ε *pos*] **by** (*simp add*: *n-def*)
show *?ge*
using *X.Hoeffding-ineq-le*[*OF* ε *pos*] **by** (*simp add*: *n-def*)
show *?abs-ge*
using *X.Hoeffding-ineq-abs-ge*[*OF* ε *pos*] **by** (*simp add*: *n-def*)
qed

lemma
assumes ε: $\varepsilon \geq 0$
assumes $a < b$ $I \neq \{\}$
defines $n \equiv card\ I$
shows *Hoeffding-ineq-ge′*:
prob $\{x \in space\ M.\ (\sum i \in I.\ X\ i\ x) / n \geq \mu + \varepsilon\} \leq$
exp $(-2 * n * \varepsilon^2 / (b - a)^2)$ (**is** *?ge*)
and *Hoeffding-ineq-le′*:
prob $\{x \in space\ M.\ (\sum i \in I.\ X\ i\ x) / n \leq \mu - \varepsilon\} \leq$
exp $(-2 * n * \varepsilon^2 / (b - a)^2)$ (**is** *?le*)
and *Hoeffding-ineq-abs-ge′*:
prob $\{x \in space\ M.\ |(\sum i \in I.\ X\ i\ x) / n - \mu| \geq \varepsilon\} \leq$
$2 *$ *exp* $(-2 * n * \varepsilon^2 / (b - a)^2)$ (**is** *?abs-ge*)
proof −
have $n > 0$
using *assms fin* **by** (*auto simp*: *field-simps*)
have ε': $\varepsilon * n \geq 0$
using ‹$n > 0$› ‹$\varepsilon \geq 0$› **by** *auto*
have *eq*: $-(2 * (\varepsilon * real\ n)^2 / (real\ (card\ I) * (b - a)^2)) =$
$-(2 * real\ n * \varepsilon^2 / (b - a)^2)$
using ‹$n > 0$› **by** (*simp add*: *power2-eq-square divide-simps n-def*)

have $\{x \in space\ M.\ (\sum i \in I.\ X\ i\ x) / n \geq \mu + \varepsilon\} =$
$\{x \in space\ M.\ (\sum i \in I.\ X\ i\ x) \geq \mu * n + \varepsilon * n\}$
using ‹$n > 0$› **by** (*intro Collect-cong conj-cong refl*) (*auto simp*: *field-simps*)
with *Hoeffding-ineq-ge*[*OF* ε' ‹$a < b$› ‹$I \neq \{\}$›] ‹$n > 0$› *eq* **show** *?ge*
by (*simp add*: *n-def mult-ac*)

have $\{x \in space\ M.\ (\sum i \in I.\ X\ i\ x) / n \leq \mu - \varepsilon\} =$
$\{x \in space\ M.\ (\sum i \in I.\ X\ i\ x) \leq \mu * n - \varepsilon * n\}$
using ‹$n > 0$› **by** (*intro Collect-cong conj-cong refl*) (*auto simp*: *field-simps*)
with *Hoeffding-ineq-le*[*OF* ε' ‹$a < b$› ‹$I \neq \{\}$›] ‹$n > 0$› *eq* **show** *?le*

```
    by (simp add: n-def mult-ac)

  have {x∈space M. |(∑ i∈I. X i x) / n − μ| ≥ ε} =
        {x∈space M. |(∑ i∈I. X i x) − μ * n| ≥ ε * n}
    using ‹n > 0› by (intro Collect-cong conj-cong refl) (auto simp: field-simps)
  with Hoeffding-ineq-abs-ge[OF ε' ‹a < b› ‹I ≠ {}›] ‹n > 0› eq show ?abs-ge
    by (simp add: n-def mult-ac)
qed

end
```

27.4 Hoeffding's Inequality for the Binomial distribution

We can now apply Hoeffding's inequality to the Binomial distribution, which can be seen as the sum of n i.i.d. coin flips (the support of each of which is contained in $[0, 1]$).

```
locale binomial-distribution =
  fixes n :: nat and p :: real
  assumes p: p ∈ {0..1}
begin

context
  fixes coins :: (nat ⇒ bool) pmf and μ
  assumes n: n > 0
  defines coins ≡ Pi-pmf {..<n} False (λ-. bernoulli-pmf p)
begin

lemma coins-component:
  assumes i: i < n
  shows   distr (measure-pmf coins) borel (λf. if f i then 1 else 0) =
          distr (measure-pmf (bernoulli-pmf p)) borel (λb. if b then 1 else 0)
proof −
  have distr (measure-pmf coins) borel (λf. if f i then 1 else 0) =
       distr (measure-pmf (map-pmf (λf. f i) coins)) borel (λb. if b then 1 else 0)
    unfolding map-pmf-rep-eq by (subst distr-distr) (auto simp: o-def)
  also have map-pmf (λf. f i) coins = bernoulli-pmf p
    unfolding coins-def using i by (subst Pi-pmf-component) auto
  finally show ?thesis
    unfolding map-pmf-rep-eq .
qed

lemma prob-binomial-pmf-conv-coins:
  measure-pmf.prob (binomial-pmf n p) {x. P (real x)} =
   measure-pmf.prob coins {x. P (∑ i<n. if x i then 1 else 0)}
proof −
  have eq1: (∑ i<n. if x i then 1 else 0) = real (card {i∈{..<n}. x i}) for x
  proof −
    have (∑ i<n. if x i then 1 else (0::real)) = (∑ i∈{i∈{..<n}. x i}. 1)
      by (intro sum.mono-neutral-cong-right) auto
```

thus *?thesis* **by** *simp*
qed
have *eq2*: *binomial-pmf n p = map-pmf* (λ*v*. *card* {*i*∈{..<*n*}. *v i*}) *coins*
unfolding *coins-def* **by** (*rule binomial-pmf-altdef'*) (*use p* **in** *auto*)
show *?thesis*
by (*subst eq2*) (*simp-all add*: *eq1*)
qed

interpretation *Hoeffding-ineq-iid*
coins {..<*n*} λ*i f*. *if f i then 1 else 0* λ*f*. *if f 0 then 1 else 0 0 1 p*
proof *unfold-locales*
show *prob-space.indep-vars* (*measure-pmf coins*) (λ-. *borel*) (λ*i f*. *if f i then 1 else 0*) {..<*n*}
unfolding *coins-def*
by (*intro prob-space.indep-vars-compose2*[*OF - indep-vars-Pi-pmf*])
(*auto simp*: *measure-pmf.prob-space-axioms*)
next
have *measure-pmf.expectation coins* (λ*f*. *if f 0 then 1 else 0* :: *real*) =
measure-pmf.expectation (*map-pmf* (λ*f*. *f 0*) *coins*) (λ*b*. *if b then 1 else 0* :: *real*)
by (*simp add*: *coins-def*)
also have *map-pmf* (λ*f*. *f 0*) *coins* = *bernoulli-pmf p*
using *n* **by** (*simp add*: *coins-def Pi-pmf-component*)
also have *measure-pmf.expectation* ... (λ*b*. *if b then 1 else 0*) = *p*
using *p* **by** *simp*
finally show *p* ≡ *measure-pmf.expectation coins* (λ*f*. *if f 0 then 1 else 0*) **by** *simp*
qed (*auto simp*: *coins-component*)

corollary
fixes ε :: *real*
assumes ε: ε ≥ *0*
shows *prob-ge*: *measure-pmf.prob* (*binomial-pmf n p*) {*x*. *x* ≥ *n* ∗ *p* + ε} ≤ *exp* (−*2* ∗ ε^2 / *n*)
and *prob-le*: *measure-pmf.prob* (*binomial-pmf n p*) {*x*. *x* ≤ *n* ∗ *p* − ε} ≤ *exp* (−*2* ∗ ε^2 / *n*)
and *prob-abs-ge*:
measure-pmf.prob (*binomial-pmf n p*) {*x*. |*x* − *n* ∗ *p*| ≥ ε} ≤ *2* ∗ *exp* (−*2* ∗ ε^2 / *n*)
proof −
have [*simp*]: {..<*n*} ≠ {}
using *n* **by** *auto*
show *measure-pmf.prob* (*binomial-pmf n p*) {*x*. *x* ≥ *n* ∗ *p* + ε} ≤ *exp* (−*2* ∗ ε^2 / *n*)
using *Hoeffding-ineq-ge*[*of* ε] **by** (*subst prob-binomial-pmf-conv-coins*) (*use assms* **in** *simp-all*)
show *measure-pmf.prob* (*binomial-pmf n p*) {*x*. *x* ≤ *n* ∗ *p* − ε} ≤ *exp* (−*2* ∗ ε^2 / *n*)
using *Hoeffding-ineq-le*[*of* ε] **by** (*subst prob-binomial-pmf-conv-coins*) (*use*

assms **in** *simp-all*)
show *measure-pmf.prob* (*binomial-pmf n p*) {*x*. |*x* − *n* ∗ *p*| ≥ *ε*} ≤ *2* ∗ *exp* (−*2* ∗ *ε*^{2} / *n*)
using *Hoeffding-ineq-abs-ge*[*of ε*]
by (*subst prob-binomial-pmf-conv-coins*) (*use assms* **in** *simp-all*)
qed

corollary
fixes *ε* :: *real*
assumes *ε*: *ε* ≥ *0*
shows *prob-ge′*: *measure-pmf.prob* (*binomial-pmf n p*) {*x*. *x* / *n* ≥ *p* + *ε*} ≤ *exp* (−*2* ∗ *n* ∗ *ε*^{2})
and *prob-le′*: *measure-pmf.prob* (*binomial-pmf n p*) {*x*. *x* / *n* ≤ *p* − *ε*} ≤ *exp* (−*2* ∗ *n* ∗ *ε*^{2})
and *prob-abs-ge′*:
measure-pmf.prob (*binomial-pmf n p*) {*x*. |*x* / *n* − *p*| ≥ *ε*} ≤ *2* ∗ *exp* (−*2* ∗ *n* ∗ *ε*^{2})
proof −
have [*simp*]: {..<*n*} ≠ {}
using *n* **by** *auto*
show *measure-pmf.prob* (*binomial-pmf n p*) {*x*. *x* / *n* ≥ *p* + *ε*} ≤ *exp* (−*2* ∗ *n* ∗ *ε*^{2})
using *Hoeffding-ineq-ge′*[*of ε*] **by** (*subst prob-binomial-pmf-conv-coins*) (*use assms* **in** *simp-all*)
show *measure-pmf.prob* (*binomial-pmf n p*) {*x*. *x* / *n* ≤ *p* − *ε*} ≤ *exp* (−*2* ∗ *n* ∗ *ε*^{2})
using *Hoeffding-ineq-le′*[*of ε*] **by** (*subst prob-binomial-pmf-conv-coins*) (*use assms* **in** *simp-all*)
show *measure-pmf.prob* (*binomial-pmf n p*) {*x*. |*x* / *n* − *p*| ≥ *ε*} ≤ *2* ∗ *exp* (−*2* ∗ *n* ∗ *ε*^{2})
using *Hoeffding-ineq-abs-ge′*[*of ε*]
by (*subst prob-binomial-pmf-conv-coins*) (*use assms* **in** *simp-all*)
qed

end

end

27.5 Tail bounds for the negative binomial distribution

Since the tail probabilities of a negative Binomial distribution are equal to the tail probabilities of some Binomial distribution, we can obtain tail bounds for the negative Binomial distribution through the Hoeffding tail bounds for the Binomial distribution.

context
fixes *p q* :: *real*
assumes *p*: *p* ∈ {*0*<..<*1*}
defines *q* ≡ *1* − *p*

begin

lemma *prob-neg-binomial-pmf-ge-bound*:
fixes n :: *nat* **and** k :: *real*
defines $\mu \equiv$ *real* $n * q$ / p
assumes k: $k \geq 0$
shows *measure-pmf.prob* (*neg-binomial-pmf n p*) {x. *real* $x \geq \mu + k$}
$\leq$ *exp* ($-$ *2* $*$ p ^ *3* $*$ k^2 / (n + p $*$ k))
proof −
consider $n = 0 \mid p = 1 \mid n > 0\ p \neq 1$
by *blast*
thus *?thesis*
proof *cases*
assume [*simp*]: $n = 0$
show *?thesis* **using** k
by (*simp add*: *indicator-def* μ-*def*)
next
assume [*simp*]: $p = 1$
show *?thesis* **using** k
by (*auto simp add*: *indicator-def* μ-*def q-def*)
next
assume n: $n > 0$ **and** $p \neq 1$
from ‹$p \neq 1$› **and** p **have** p: $p \in \{0<..<1\}$
by *auto*
from p **have** q: $q \in \{0<..<1\}$
by (*auto simp*: *q-def*)

define *k1* **where** *k1* = $\mu + k$
have *k1*: *k1* $\geq \mu$
using k **by** (*simp add*: *k1-def*)
have *k1* > *0*
by (*rule less-le-trans*[*OF* - *k1*]) (*use* p n **in** ‹*auto simp*: *q-def* μ-*def*›)

define *k1'* **where** *k1'* = *nat* (*ceiling k1*)
have $\mu \geq 0$ **using** p
by (*auto simp*: μ-*def q-def*)
have $\neg(x < k1') \longleftrightarrow$ *real* $x \geq k1$ **for** x
unfolding *k1'-def* **by** *linarith*
hence *eq*: *UNIV* $-$ {..<*k1'*} = {x. $x \geq k1$}
by *auto*
hence *measure-pmf.prob* (*neg-binomial-pmf n p*) {n. $n \geq k1$} =
1 $-$ *measure-pmf.prob* (*neg-binomial-pmf n p*) {..<*k1'*}
using *measure-pmf.prob-compl*[*of* {..<*k1'*} *neg-binomial-pmf n p*] **by** *simp*
also have *measure-pmf.prob* (*neg-binomial-pmf n p*) {..<*k1'*} =
measure-pmf.prob (*binomial-pmf* (*n* + *k1'* − *1*) *q*) {..<*k1'*}
unfolding *q-def* **using** p **by** (*intro prob-neg-binomial-pmf-lessThan*) *auto*
also have *1* $-$ ... = *measure-pmf.prob* (*binomial-pmf* (*n* + *k1'* − *1*) *q*) {n.
$n \geq k1$}
using *measure-pmf.prob-compl*[*of* {..<*k1'*} *binomial-pmf* (*n* + *k1'* − *1*) *q*]

eq **by** *simp*
also have $\{x.\ real\ x \geq k1\} = \{x.\ x \geq real\ (n + k1' - 1) * q + (k1 - real\ (n + k1' - 1) * q)\}$
by *simp*
also have *measure-pmf.prob* (*binomial-pmf* $(n + k1' - 1)\ q) \ldots \leq$
$exp\ (-2 * (k1 - real\ (n + k1' - 1) * q)^2\ /\ real\ (n + k1' - 1))$
proof (*rule binomial-distribution.prob-ge*)
show *binomial-distribution q*
by *unfold-locales* (*use q* **in** *auto*)
next
show $n + k1' - 1 > 0$
using ‹$k1 > 0$› n **unfolding** *k1'-def* **by** *linarith*
next
have $real\ (n + nat\ \lceil k1 \rceil - 1) \leq real\ n + k1$
using ‹$k1 > 0$› **by** *linarith*
hence $real\ (n + k1' - 1) * q \leq (real\ n + k1) * q$
unfolding *k1'-def* **by** (*intro mult-right-mono*) (*use p* **in** ‹*simp-all add*: *q-def*›)
also have $\ldots \leq k1$
using *k1 p* **by** (*simp add*: *q-def field-simps* μ*-def*)
finally show $0 \leq k1 - real\ (n + k1' - 1) * q$
by *simp*
qed
also have $\{x.\ real\ (n + k1' - 1) * q + (k1 - real\ (n + k1' - 1) * q) \leq real\ x\} = \{x.\ real\ x \geq k1\}$
by *simp*
also have $exp\ (-2 * (k1 - real\ (n + k1' - 1) * q)^2\ /\ real\ (n + k1' - 1)) \leq$
$exp\ (-2 * (k1 - (n + k1) * q)^2\ /\ (n + k1))$
proof −
have $real\ (n + k1' - Suc\ 0) \leq real\ n + k1$
unfolding *k1'-def* **using** ‹$k1 > 0$› **by** *linarith*
moreover have $(real\ n + k1) * q \leq k1$
using *k1 p* **by** (*auto simp*: *q-def field-simps* μ*-def*)
moreover have $1 < n + k1'$
using n ‹$k1 > 0$› **unfolding** *k1'-def* **by** *linarith*
ultimately have $2 * (k1 - real\ (n + k1' - 1) * q)^2\ /\ real\ (n + k1' - 1) \geq 2 * (k1 - (n + k1) * q)^2\ /\ (n + k1)$
by (*intro frac-le mult-left-mono power-mono mult-nonneg-nonneg mult-right-mono diff-mono*)
(*use q* **in** *simp-all*)
thus *?thesis*
by *simp*
qed
also have $\ldots = exp\ (-2 * (p * k1 - q * n)^2\ /\ (k1 + n))$
by (*simp add*: *q-def algebra-simps*)
also have $-2 * (p * k1 - q * n)^2 = -2 * p^2 * (k1 - \mu)^2$
using *p* **by** (*auto simp*: *field-simps* μ*-def*)
also have $k1 - \mu = k$
by (*simp add*: *k1-def* μ*-def*)

```
    also note k1-def
    also have μ + k + real n = real n / p + k
      using p by (simp add: μ-def q-def field-simps)
    also have − 2 * p² * k² / (real n / p + k) = − 2 * p ^ 3 * k² / (p * k + n)
      using p by (simp add: field-simps power3-eq-cube power2-eq-square)
    finally show ?thesis by (simp add: add-ac)
  qed
qed

lemma prob-neg-binomial-pmf-le-bound:
  fixes n :: nat and k :: real
  defines μ ≡ real n * q / p
  assumes k: k ≥ 0
  shows measure-pmf.prob (neg-binomial-pmf n p) {x. real x ≤ μ − k}
        ≤ exp (−2 * p ^ 3 * k² / (n − p * k))
proof −
  consider n = 0 | p = 1 | k > μ | n > 0 p ≠ 1 k ≤ μ
    by force
  thus ?thesis
  proof cases
    assume [simp]: n = 0
    show ?thesis using k
      by (simp add: indicator-def μ-def)
  next
    assume [simp]: p = 1
    show ?thesis using k
      by (auto simp add: indicator-def μ-def q-def)
  next
    assume k > μ
    hence {x. real x ≤ μ − k} = {}
      by auto
    thus ?thesis by simp
  next
    assume n: n > 0 and p ≠ 1 and k ≤ μ
    from ‹p ≠ 1› and p have p: p ∈ {0<..<1}
      by auto
    from p have q: q ∈ {0<..<1}
      by (auto simp: q-def)

    define f :: real ⇒ real where f = (λx. (p * x − q * n)² / (x + n))
    have f-mono: f x ≥ f y if x ≥ 0 y ≤ n * q / p x ≤ y for x y :: real
      using that(3)
    proof (rule DERIV-nonpos-imp-nonincreasing)
      fix t assume t: t ≥ x t ≤ y
      have x > −n
        using n ‹x ≥ 0› by linarith
      hence (f has-field-derivative ((p * t − q * n) * (n * (1 + p) + p * t) / (n +
t) ^ 2)) (at t)
        unfolding f-def using t
```

```
    by (auto intro!: derivative-eq-intros simp: algebra-simps q-def power2-eq-square)
    moreover {
      have p * t ≤ p * y
        using p by (intro mult-left-mono t) auto
      also have p * y ≤ q * n
        using ‹y ≤ n * q / p› p by (simp add: field-simps)
      finally have p * t ≤ q * n .
    }
    hence (p * t - q * n) * (n * (1 + p) + p * t) / (n + t) ^ 2 ≤ 0
      using p ‹x ≥ 0› t
        by (intro mult-nonpos-nonneg divide-nonpos-nonneg add-nonneg-nonneg
mult-nonneg-nonneg) auto
    ultimately show ∃ y. (f has-real-derivative y) (at t) ∧ y ≤ 0
      by blast
  qed

  define k1 where k1 = μ - k
  have k1: k1 ≤ real n * q / p
    using assms by (simp add: μ-def k1-def)
  have k1 ≥ 0
    using k ‹k ≤ μ› by (simp add: μ-def k1-def)

  define k1' where k1' = nat (floor k1)
  have μ ≥ 0 using p
    by (auto simp: μ-def q-def)
  have (x ≤ k1') ⟷ real x ≤ k1 for x
    unfolding k1'-def not-less using ‹k1 ≥ 0› by linarith
  hence eq: {n. n ≤ k1}  = {..k1'}
    by auto
  hence measure-pmf.prob (neg-binomial-pmf n p) {n. n ≤ k1} =
        measure-pmf.prob (neg-binomial-pmf n p) {..k1'}
    by simp
  also have measure-pmf.prob (neg-binomial-pmf n p) {..k1'} =
          measure-pmf.prob (binomial-pmf (n + k1') q) {..k1'}
    unfolding q-def using p by (intro prob-neg-binomial-pmf-atMost) auto
  also note eq [symmetric]
  also have {x. real x ≤ k1} = {x. x ≤ real (n + k1') * q - (real (n + k1') *
q - real k1')}
    using eq by auto
  also have measure-pmf.prob (binomial-pmf (n + k1') q) ... ≤
            exp (-2 * (real (n + k1') * q - real k1')^2 / real (n + k1'))
  proof (rule binomial-distribution.prob-le)
    show binomial-distribution q
      by unfold-locales (use q in auto)
  next
    show n + k1' > 0
      using ‹k1 ≥ 0› n unfolding k1'-def by linarith
  next
    have p * k1' ≤ p * k1
```

```
      using p ‹k1 ≥ 0› by (intro mult-left-mono) (auto simp: k1'-def)
    also have ... ≤ q * n
      using k1 p by (simp add: field-simps)
    finally show 0 ≤ real (n + k1') * q - real k1'
      by (simp add: algebra-simps q-def)
  qed
  also have {x. real x ≤ real (n + k1') * q - (real (n + k1') * q - k1')} =
{..k1'}
    by auto
  also have real (n + k1') * q - k1' = -(p * k1' - q * n)
    by (simp add: q-def algebra-simps)
  also have ... ^ 2 = (p * k1' - q * n) ^ 2
    by algebra
  also have - 2 * (p * real k1' - q * real n)² / real (n + k1') = -2 * f (real
k1')
    by (simp add: f-def)
  also have f (real k1') ≥ f k1
    by (rule f-mono) (use ‹k1 ≥ 0› k1 in ‹auto simp: k1'-def›)
  hence exp (-2 * f (real k1')) ≤ exp (-2 * f k1)
    by simp
  also have ... = exp (-2 * (p * k1 - q * n)² / (k1 + n))
    by (simp add: f-def)

  also have -2 * (p * k1 - q * n)² = -2 * p² * (k1 - μ)²
    using p by (auto simp: field-simps μ-def)
  also have (k1 - μ) ^ 2 = k ^ 2
    by (simp add: k1-def μ-def)
  also note k1-def
  also have μ - k + real n = real n / p - k
    using p by (simp add: μ-def q-def field-simps)
  also have - 2 * p² * k² / (real n / p - k) = - 2 * p ^ 3 * k² / (n - p * k)
    using p by (simp add: field-simps power3-eq-cube power2-eq-square)
  also have {..k1'} = {x. real x ≤ μ - k}
    using eq by (simp add: k1-def)
  finally show ?thesis .
 qed
qed
```

Due to the function $exp(-l/x)$ being concave for $x \geq \frac{l}{2}$, the above two bounds can be combined into the following one for moderate values of k. (cf. https://math.stackexchange.com/questions/1565559)

```
lemma prob-neg-binomial-pmf-abs-ge-bound:
  fixes n :: nat and k :: real
  defines μ ≡ real n * q / p
  assumes k ≥ 0 and n-ge: n ≥ p * k * (p² * k + 1)
  shows measure-pmf.prob (neg-binomial-pmf n p) {x. |real x - μ| ≥ k} ≤
        2 * exp (-2 * p ^ 3 * k ^ 2 / n)
proof (cases k = 0)
  case False
```

```
  with ‹k ≥ 0› have k: k > 0
    by auto
  define l :: real where l = 2 * p ^ 3 * k ^ 2
  have l: l > 0
    using p k by (auto simp: l-def)
  define f :: real ⇒ real where f = (λx. exp (−l / x))
  define f' where f' = (λx. −l * exp (−l / x) / x ^ 2)

  have f'-mono: f' x ≤ f' y if x ≥ l / 2 x ≤ y for x y :: real
    using that(2)
  proof (rule DERIV-nonneg-imp-nondecreasing)
    fix t assume t: x ≤ t t ≤ y
    have t > 0
      using that l t by auto
    have (f' has-field-derivative (l * (2 * t − l) / (exp (l / t) * t ^ 4))) (at t)
      unfolding f'-def using t that ‹t > 0›
       by (auto intro!: derivative-eq-intros simp: field-simps exp-minus simp flip:
power-Suc)
    moreover have l * (2 * t − l) / (exp (l / t) * t ^ 4) ≥ 0
      using that t l by (intro divide-nonneg-pos mult-nonneg-nonneg) auto
    ultimately show ∃ y. (f' has-real-derivative y) (at t) ∧ 0 ≤ y by blast
  qed

  have convex: convex-on {l/2..} (λx. −f x) unfolding f-def
  proof (intro convex-on-realI[where f' = f'])
    show ((λx. − exp (− l / x)) has-real-derivative f' x) (at x) if x ∈ {l/2..} for x
      using that l
     by (auto intro!: derivative-eq-intros simp: f'-def power2-eq-square algebra-simps)
  qed (use l in ‹auto intro!: f'-mono›)

  have eq: {x. |real x − μ| ≥ k} = {x. real x ≤ μ − k} ∪ {x. real x ≥ μ + k}
    by auto
  have measure-pmf.prob (neg-binomial-pmf n p) {x. |real x − μ| ≥ k} ≤
        measure-pmf.prob (neg-binomial-pmf n p) {x. real x ≤ μ − k} +
        measure-pmf.prob (neg-binomial-pmf n p) {x. real x ≥ μ + k}
    by (subst eq, rule measure-Un-le) auto
  also have ... ≤ exp (−2 * p ^ 3 * k^2 / (n − p * k)) + exp (−2 * p ^ 3 * k^2 /
(n + p * k))
    unfolding μ-def
   by (intro prob-neg-binomial-pmf-le-bound prob-neg-binomial-pmf-ge-bound add-mono
‹k ≥ 0›)
  also have ... = 2 * (1/2 * f (n − p * k) + 1/2 * f (n + p * k))
    by (simp add: f-def l-def)
  also have 1/2 * f (n − p * k) + 1/2 * f (n + p * k) ≤ f (1/2 * (n − p * k)
+ 1/2 * (n + p * k))
  proof −
    let ?x = n − p * k and ?y = n + p * k
    have le1: l / 2 ≤ ?x using n-ge
      by (simp add: l-def power2-eq-square power3-eq-cube algebra-simps)
```

```
    also have ... ≤ ?y
      using p k by simp
    finally have le2: l / 2 ≤ ?y .
    have −f ((1 − 1 / 2) *R ?x + (1 / 2) *R ?y) ≤ (1 − 1 / 2) * − f ?x + 1 /
2 * − f ?y
      using le1 le2 by (intro convex-onD[OF convex]) auto
    thus ?thesis by simp
  qed
  also have 1/2 * (n − p * k) + 1/2 * (n + p * k) = n
    by (simp add: algebra-simps)
  also have 2 * f n = 2 * exp (−l / n)
    by (simp add: f-def)
  finally show ?thesis
    by (simp add: l-def)
qed auto

end

end


theory Stream-Space
imports
  Infinite-Product-Measure
  HOL−Library.Stream
  HOL−Library.Linear-Temporal-Logic-on-Streams
begin

lemma stream-eq-Stream-iff: s = x ## t ⟷ (shd s = x ∧ stl s = t)
  by (cases s) simp

lemma Stream-snth: (x ## s) !! n = (case n of 0 ⇒ x | Suc n ⇒ s !! n)
  by (cases n) simp-all

definition to-stream :: (nat ⇒ 'a) ⇒ 'a stream where
  to-stream X = smap X nats

lemma to-stream-nat-case: to-stream (case-nat x X) = x ## to-stream X
  unfolding to-stream-def
   by (subst siterate.ctr) (simp add: smap-siterate[symmetric] stream.map-comp
comp-def)

lemma to-stream-in-streams: to-stream X ∈ streams S ⟷ (∀ n. X n ∈ S)
  by (simp add: to-stream-def streams-iff-snth)

definition stream-space :: 'a measure ⇒ 'a stream measure where
  stream-space M =
   distr (Π_M i∈UNIV. M) (vimage-algebra (streams (space M)) snth (Π_M i∈UNIV.
M)) to-stream
```

lemma *space-stream-space*: *space* (*stream-space M*) = *streams* (*space M*)
by (*simp add*: *stream-space-def*)

lemma *streams-stream-space*[*intro*]: *streams* (*space M*) ∈ *sets* (*stream-space M*)
using *sets.top*[*of stream-space M*] **by** (*simp add*: *space-stream-space*)

lemma *stream-space-Stream*:
x ## ω ∈ *space* (*stream-space M*) ⟷ *x* ∈ *space M* ∧ ω ∈ *space* (*stream-space M*)
by (*simp add*: *space-stream-space streams-Stream*)

lemma *stream-space-eq-distr*: *stream-space M* = *distr* (Π$_M$ *i*∈*UNIV*. *M*) (*stream-space M*) *to-stream*
unfolding *stream-space-def* **by** (*rule distr-cong*) *auto*

lemma *sets-stream-space-cong*[*measurable-cong*]:
sets M = *sets N* ⟹ *sets* (*stream-space M*) = *sets* (*stream-space N*)
using *sets-eq-imp-space-eq*[*of M N*] **by** (*simp add*: *stream-space-def vimage-algebra-def cong*: *sets-PiM-cong*)

lemma *measurable-snth-PiM*: (λω *n*. ω !! *n*) ∈ *measurable* (*stream-space M*) (Π$_M$ *i*∈*UNIV*. *M*)
by (*auto intro*!: *measurable-vimage-algebra1*
simp: *space-PiM streams-iff-sset sset-range image-subset-iff stream-space-def*)

lemma *measurable-snth*[*measurable*]: (λω. ω !! *n*) ∈ *measurable* (*stream-space M*) *M*
using *measurable-snth-PiM measurable-component-singleton* **by** (*rule measurable-compose*) *simp*

lemma *measurable-shd*[*measurable*]: *shd* ∈ *measurable* (*stream-space M*) *M*
using *measurable-snth*[*of 0*] **by** *simp*

lemma *measurable-stream-space2*:
assumes *f-snth*: ⋀*n*. (λ*x*. *f x* !! *n*) ∈ *measurable N M*
shows *f* ∈ *measurable N* (*stream-space M*)
unfolding *stream-space-def measurable-distr-eq2*
proof (*rule measurable-vimage-algebra2*)
show *f* ∈ *space N* → *streams* (*space M*)
using *f-snth*[*THEN measurable-space*] **by** (*auto simp add*: *streams-iff-sset sset-range*)
show (λ*x*. (!!) (*f x*)) ∈ *measurable N* (*Pi$_M$ UNIV* (λ*i*. *M*))
proof (*rule measurable-PiM-single′*)
show (λ*x*. (!!) (*f x*)) ∈ *space N* → *UNIV* →$_E$ *space M*
using *f-snth*[*THEN measurable-space*] **by** *auto*
qed (*rule f-snth*)
qed

lemma *measurable-stream-coinduct*[*consumes 1*, *case-names shd stl*, *coinduct set*:

measurable]:
assumes *F f*
assumes *h*: $\bigwedge f.\ F\ f \Longrightarrow (\lambda x.\ shd\ (f\ x)) \in measurable\ N\ M$
assumes *t*: $\bigwedge f.\ F\ f \Longrightarrow F\ (\lambda x.\ stl\ (f\ x))$
shows $f \in measurable\ N\ (stream\text{-}space\ M)$
proof (*rule measurable-stream-space2*)
fix *n* **show** $(\lambda x.\ f\ x\ !!\ n) \in measurable\ N\ M$
using ‹*F f*› **by** (*induction n arbitrary*: *f*) (*auto intro*: *h t*)
qed

lemma *measurable-sdrop*[*measurable*]: $sdrop\ n \in measurable\ (stream\text{-}space\ M)\ (stream\text{-}space\ M)$
by (*rule measurable-stream-space2*) (*simp add*: *sdrop-snth*)

lemma *measurable-stl*[*measurable*]: $(\lambda\omega.\ stl\ \omega) \in measurable\ (stream\text{-}space\ M)\ (stream\text{-}space\ M)$
by (*rule measurable-stream-space2*) (*simp del*: *snth.simps add*: *snth.simps*[*symmetric*])

lemma *measurable-to-stream*[*measurable*]: *to-stream* $\in measurable\ (\Pi_M\ i{\in}UNIV.\ M)\ (stream\text{-}space\ M)$
by (*rule measurable-stream-space2*) (*simp add*: *to-stream-def*)

lemma *measurable-Stream*[*measurable* (*raw*)]:
assumes *f*[*measurable*]: $f \in measurable\ N\ M$
assumes *g*[*measurable*]: $g \in measurable\ N\ (stream\text{-}space\ M)$
shows $(\lambda x.\ f\ x\ \#\#\ g\ x) \in measurable\ N\ (stream\text{-}space\ M)$
by (*rule measurable-stream-space2*) (*simp add*: *Stream-snth*)

lemma *measurable-smap*[*measurable*]:
assumes *X*[*measurable*]: $X \in measurable\ N\ M$
shows $smap\ X \in measurable\ (stream\text{-}space\ N)\ (stream\text{-}space\ M)$
by (*rule measurable-stream-space2*) *simp*

lemma *measurable-stake*[*measurable*]:
$stake\ i \in measurable\ (stream\text{-}space\ (count\text{-}space\ UNIV))\ (count\text{-}space\ (UNIV :: {'a}{::}countable\ list\ set))$
by (*induct i*) *auto*

lemma *measurable-shift*[*measurable*]:
assumes *f*: $f \in measurable\ N\ (stream\text{-}space\ M)$
assumes [*measurable*]: $g \in measurable\ N\ (stream\text{-}space\ M)$
shows $(\lambda x.\ stake\ n\ (f\ x)\ @{-}\ g\ x) \in measurable\ N\ (stream\text{-}space\ M)$
using *f* **by** (*induction n arbitrary*: *f*) *simp-all*

lemma *measurable-case-stream-replace*[*measurable* (*raw*)]:
$(\lambda x.\ f\ x\ (shd\ (g\ x))\ (stl\ (g\ x))) \in measurable\ M\ N \Longrightarrow (\lambda x.\ case\text{-}stream\ (f\ x)\ (g\ x)) \in measurable\ M\ N$
unfolding *stream.case-eq-if* **.**

lemma *measurable-ev-at*[*measurable*]:
assumes [*measurable*]: *Measurable.pred* (*stream-space M*) *P*
shows *Measurable.pred* (*stream-space M*) (*ev-at P n*)
by (*induction n*) *auto*

lemma *measurable-alw*[*measurable*]:
Measurable.pred (*stream-space M*) *P* ⟹ *Measurable.pred* (*stream-space M*) (*alw P*)
unfolding *alw-def*
by (*coinduction rule*: *measurable-gfp-coinduct*) (*auto simp*: *inf-continuous-def*)

lemma *measurable-ev*[*measurable*]:
Measurable.pred (*stream-space M*) *P* ⟹ *Measurable.pred* (*stream-space M*) (*ev P*)
unfolding *ev-def*
by (*coinduction rule*: *measurable-lfp-coinduct*) (*auto simp*: *sup-continuous-def*)

lemma *measurable-until*:
assumes [*measurable*]: *Measurable.pred* (*stream-space M*) *φ Measurable.pred* (*stream-space M*) *ψ*
shows *Measurable.pred* (*stream-space M*) (*φ until ψ*)
unfolding *UNTIL-def*
by (*coinduction rule*: *measurable-gfp-coinduct*) (*simp-all add*: *inf-continuous-def fun-eq-iff*)

lemma *measurable-holds* [*measurable*]: *Measurable.pred M P* ⟹ *Measurable.pred* (*stream-space M*) (*holds P*)
unfolding *holds.simps*[*abs-def*]
by (*rule measurable-compose*[*OF measurable-shd*]) *simp*

lemma *measurable-hld*[*measurable*]: **assumes** [*measurable*]: *t* ∈ *sets M* **shows** *Measurable.pred* (*stream-space M*) (*HLD t*)
unfolding *HLD-def* **by** *measurable*

lemma *measurable-nxt*[*measurable* (*raw*)]:
Measurable.pred (*stream-space M*) *P* ⟹ *Measurable.pred* (*stream-space M*) (*nxt P*)
unfolding *nxt.simps*[*abs-def*] **by** *simp*

lemma *measurable-suntil*[*measurable*]:
assumes [*measurable*]: *Measurable.pred* (*stream-space M*) *Q Measurable.pred* (*stream-space M*) *P*
shows *Measurable.pred* (*stream-space M*) (*Q suntil P*)
unfolding *suntil-def* **by** (*coinduction rule*: *measurable-lfp-coinduct*) (*auto simp*: *sup-continuous-def*)

lemma *measurable-szip*:
(λ(*ω1*, *ω2*). *szip ω1 ω2*) ∈ *measurable* (*stream-space M* ⨂$_M$ *stream-space N*) (*stream-space* (*M* ⨂$_M$ *N*))

proof (*rule measurable-stream-space2*)
fix *n*
have (λ*x*. (*case x of* (*ω1*, *ω2*) ⇒ *szip ω1 ω2*) !! *n*) = (λ(*ω1*, *ω2*). (*ω1* !! *n*, *ω2* !! *n*))
by *auto*
also have ... ∈ *measurable* (*stream-space M* ⨂$_M$ *stream-space N*) (*M* ⨂$_M$ *N*)
by *measurable*
finally show (λ*x*. (*case x of* (*ω1*, *ω2*) ⇒ *szip ω1 ω2*) !! *n*) ∈ *measurable* (*stream-space M* ⨂$_M$ *stream-space N*) (*M* ⨂$_M$ *N*)
.
qed

lemma (**in** *prob-space*) *prob-space-stream-space*: *prob-space* (*stream-space M*)
proof −
interpret *product-prob-space* λ-. *M UNIV* **..**
show *?thesis*
by (*subst stream-space-eq-distr*) (*auto intro*!: *P.prob-space-distr*)
qed

lemma (**in** *prob-space*) *nn-integral-stream-space*:
assumes [*measurable*]: *f* ∈ *borel-measurable* (*stream-space M*)
shows (∫$^+$*X*. *f X* ∂*stream-space M*) = (∫$^+$*x*. (∫$^+$*X*. *f* (*x* ## *X*) ∂*stream-space M*) ∂*M*)
proof −
interpret *S*: *sequence-space M* **..**
interpret *P*: *pair-sigma-finite M* Π$_M$ *i*::*nat*∈*UNIV*. *M* **..**

have (∫$^+$*X*. *f X* ∂*stream-space M*) = (∫$^+$*X*. *f* (*to-stream X*) ∂*S.S*)
by (*subst stream-space-eq-distr*) (*simp add*: *nn-integral-distr*)
also have ... = (∫$^+$*X*. *f* (*to-stream* ((λ(*s*, *ω*). *case-nat s ω*) *X*)) ∂(*M* ⨂$_M$ *S.S*))
by (*subst S.PiM-iter*[*symmetric*]) (*simp add*: *nn-integral-distr*)
also have ... = (∫$^+$*x*. ∫$^+$*X*. *f* (*to-stream* ((λ(*s*, *ω*). *case-nat s ω*) (*x*, *X*))) ∂*S.S* ∂*M*)
by (*subst S.nn-integral-fst*) *simp-all*
also have ... = (∫$^+$*x*. ∫$^+$*X*. *f* (*x* ## *to-stream X*) ∂*S.S* ∂*M*)
by (*auto intro*!: *nn-integral-cong simp*: *to-stream-nat-case*)
also have ... = (∫$^+$*x*. ∫$^+$*X*. *f* (*x* ## *X*) ∂*stream-space M* ∂*M*)
by (*subst stream-space-eq-distr*)
(*simp add*: *nn-integral-distr cong*: *nn-integral-cong*)
finally show *?thesis* **.**
qed

lemma (**in** *prob-space*) *emeasure-stream-space*:
assumes *X*[*measurable*]: *X* ∈ *sets* (*stream-space M*)
shows *emeasure* (*stream-space M*) *X* = (∫$^+$*t*. *emeasure* (*stream-space M*) {*x*∈*space* (*stream-space M*). *t* ## *x* ∈ *X* } ∂*M*)
proof −
have *eq*: ⋀*x xs*. *xs* ∈ *space* (*stream-space M*) ⟹ *x* ∈ *space M* ⟹

indicator X (x ## xs) = indicator {xs∈space (stream-space M). x ## xs ∈ X } xs
by (*auto split*: *split-indicator*)
show *?thesis*
using *nn-integral-stream-space*[*of indicator X*]
apply (*auto intro*!: *nn-integral-cong*)
apply (*subst nn-integral-cong*)
apply (*rule eq*)
apply *simp-all*
done
qed

lemma (**in** *prob-space*) *prob-stream-space*:
assumes *P*[*measurable*]: {*x*∈*space* (*stream-space M*). *P x*} ∈ *sets* (*stream-space M*)
shows $\mathcal{P}$(*x in stream-space M*. *P x*) = ($\int^+ t$. $\mathcal{P}$(*x in stream-space M*. *P* (*t ## x*)) ∂*M*)
proof −
interpret *S*: *prob-space stream-space M*
by (*rule prob-space-stream-space*)
show *?thesis*
unfolding *S.emeasure-eq-measure*[*symmetric*]
by (*subst emeasure-stream-space*) (*auto simp*: *stream-space-Stream intro*!: *nn-integral-cong*)
qed

lemma (**in** *prob-space*) *AE-stream-space*:
assumes [*measurable*]: *Measurable.pred* (*stream-space M*) *P*
shows (*AE X in stream-space M*. *P X*) = (*AE x in M*. *AE X in stream-space M*. *P* (*x ## X*))
proof −
interpret *stream*: *prob-space stream-space M*
by (*rule prob-space-stream-space*)

have *eq*: ⋀*x X*. *indicator* {*x*. ¬ *P x*} (*x ## X*) = *indicator* {*X*. ¬ *P* (*x ## X*)} *X*
by (*auto split*: *split-indicator*)
show *?thesis*
apply (*subst AE-iff-nn-integral*, *simp*)
apply (*subst nn-integral-stream-space*, *simp*)
apply (*subst eq*)
apply (*subst nn-integral-0-iff-AE*, *simp*)
apply (*simp add*: *AE-iff-nn-integral*[*symmetric*])
done
qed

lemma (**in** *prob-space*) *AE-stream-all*:
assumes [*measurable*]: *Measurable.pred M P* **and** *P*: *AE x in M*. *P x*
shows *AE x in stream-space M*. *stream-all P x*
proof −

```
  { fix n have AE x in stream-space M. P (x !! n)
    proof (induct n)
      case 0 with P show ?case
        by (subst AE-stream-space) (auto elim!: eventually-mono)
    next
      case (Suc n) then show ?case
        by (subst AE-stream-space) auto
    qed }
  then show ?thesis
    unfolding stream-all-def by (simp add: AE-all-countable)
qed

lemma streams-sets:
  assumes X[measurable]: X ∈ sets M shows streams X ∈ sets (stream-space M)
proof −
  have streams X = {x∈space (stream-space M). x ∈ streams X}
   using streams-mono[OF - sets.sets-into-space[OF X]] by (auto simp: space-stream-space)
  also have ... = {x∈space (stream-space M). gfp (λp x. shd x ∈ X ∧ p (stl x))
x}
    apply (simp add: set-eq-iff streams-def streamsp-def)
    apply (intro allI conj-cong refl arg-cong2[where f=gfp] ext)
    apply (case-tac xa)
    apply auto
    done
  also have ... ∈ sets (stream-space M)
    apply (intro predE)
    apply (coinduction rule: measurable-gfp-coinduct)
    apply (auto simp: inf-continuous-def)
    done
  finally show ?thesis .
qed

lemma sets-stream-space-in-sets:
  assumes space: space N = streams (space M)
  assumes sets: ⋀i. (λx. x !! i) ∈ measurable N M
  shows sets (stream-space M) ⊆ sets N
  unfolding stream-space-def sets-distr
   by (auto intro!: sets-image-in-sets measurable-Sup2 measurable-vimage-algebra2
del: subsetI equalityI
          simp add: sets-PiM-eq-proj snth-in space sets cong: measurable-cong-sets)

lemma sets-stream-space-eq: sets (stream-space M) =
    sets (SUP i∈UNIV. vimage-algebra (streams (space M)) (λs. s !! i) M)
  by (auto intro!: sets-stream-space-in-sets sets-Sup-in-sets sets-image-in-sets
                   measurable-Sup1 snth-in measurable-vimage-algebra1 del: subsetI
          simp: space-Sup-eq-UN space-stream-space)

lemma sets-restrict-stream-space:
  assumes S[measurable]: S ∈ sets M
```

shows *sets* (*restrict-space* (*stream-space M*) (*streams S*)) = *sets* (*stream-space* (*restrict-space M S*))
using *S*[*THEN sets.sets-into-space*]
apply (*subst restrict-space-eq-vimage-algebra*)
apply (*simp add*: *space-stream-space streams-mono2*)
apply (*subst vimage-algebra-cong*[*OF refl refl sets-stream-space-eq*])
apply (*subst sets-stream-space-eq*)
apply (*subst sets-vimage-Sup-eq*[**where** *Y*=*streams* (*space M*)])
apply *simp*
apply *auto* []
apply (*auto intro*: *streams-mono*) []
apply *auto* []
apply (*simp add*: *image-image space-restrict-space*)
apply (*simp add*: *vimage-algebra-cong*[*OF refl refl restrict-space-eq-vimage-algebra*])
apply (*subst* (*1 2*) *vimage-algebra-vimage-algebra-eq*)
apply (*auto simp*: *streams-mono snth-in*)
done

primrec *sstart* :: *'a set* ⇒ *'a list* ⇒ *'a stream set* **where**
sstart S [] = *streams S*
| [*simp del*]: *sstart S* (*x* # *xs*) = (##) *x* ‘ *sstart S xs*

lemma *in-sstart*[*simp*]: *s* ∈ *sstart S* (*x* # *xs*) ⟷ *shd s* = *x* ∧ *stl s* ∈ *sstart S xs*
by (*cases s*) (*auto simp*: *sstart.simps*(*2*))

lemma *sstart-in-streams*: *xs* ∈ *lists S* ⟹ *sstart S xs* ⊆ *streams S*
by (*induction xs*) (*auto simp*: *sstart.simps*(*2*))

lemma *sstart-eq*: *x* ∈ *streams S* ⟹ *x* ∈ *sstart S xs* = (∀ *i*<*length xs*. *x* !! *i* = *xs* ! *i*)
by (*induction xs arbitrary*: *x*) (*auto simp*: *nth-Cons streams-stl split*: *nat.splits*)

lemma *sstart-sets*: *sstart S xs* ∈ *sets* (*stream-space* (*count-space UNIV*))
proof (*induction xs*)
case (*Cons x xs*)
note *Cons*[*measurable*]
have *sstart S* (*x* # *xs*) =
{*s*∈*space* (*stream-space* (*count-space UNIV*)). *shd s* = *x* ∧ *stl s* ∈ *sstart S xs*}
by (*simp add*: *set-eq-iff space-stream-space*)
also have ... ∈ *sets* (*stream-space* (*count-space UNIV*))
by *measurable*
finally show *?case* **.**
qed (*simp add*: *streams-sets*)

lemma *sigma-sets-singletons*:
assumes *countable S*
shows *sigma-sets S* ((λ*s*. {*s*})‘*S*) = *Pow S*
proof *safe*
interpret *sigma-algebra S sigma-sets S* ((λ*s*. {*s*})‘*S*)

```
    by (rule sigma-algebra-sigma-sets) auto
  fix A assume A ⊆ S
  with assms have (⋃ a∈A. {a}) ∈ sigma-sets S ((λs. {s}) ‘S)
    by (intro countable-UN′) (auto dest: countable-subset)
  then show A ∈ sigma-sets S ((λs. {s}) ‘S)
    by simp
qed (auto dest: sigma-sets-into-sp[rotated])

lemma sets-count-space-eq-sigma:
  countable S ⟹ sets (count-space S) = sets (sigma S ((λs. {s}) ‘S))
  by (subst sets-measure-of) (auto simp: sigma-sets-singletons)

lemma sets-stream-space-sstart:
  assumes S[simp]: countable S
  shows sets (stream-space (count-space S)) = sets (sigma (streams S) (sstart
S‘lists S ∪ {{}}))
proof
  have [simp]: sstart S ‘ lists S ⊆ Pow (streams S)
    by (simp add: image-subset-iff sstart-in-streams)

  let ?S = sigma (streams S) (sstart S ‘ lists S ∪ {{}})
  { fix i a assume a ∈ S
    { fix x have (x !! i = a ∧ x ∈ streams S) ⟷ (∃ xs∈lists S. length xs = i ∧ x
∈ sstart S (xs @ [a]))
      proof (induction i arbitrary: x)
        case (Suc i) from this[of stl x] show ?case
          by (simp add: length-Suc-conv Bex-def ex-simps[symmetric] del: ex-simps)
             (metis stream.collapse streams-Stream)
      qed (insert ‹a ∈ S›, auto intro: streams-stl in-streams) }
    then have (λx. x !! i) −‘ {a} ∩ streams S = (⋃ xs∈{xs∈lists S. length xs =
i}. sstart S (xs @ [a]))
      by (auto simp add: set-eq-iff)
    also have ... ∈ sets ?S
      using ‹a∈S› by (intro sets.countable-UN′) (auto intro!: sigma-sets.Basic
image-eqI)
    finally have (λx. x !! i) −‘ {a} ∩ streams S ∈ sets ?S . }
  then show sets (stream-space (count-space S)) ⊆ sets (sigma (streams S) (sstart
S‘lists S ∪ {{}}))
  by (intro sets-stream-space-in-sets) (auto simp: measurable-count-space-eq-countable
snth-in)

  have sigma-sets (space (stream-space (count-space S))) (sstart S‘lists S ∪ {{}})
⊆ sets (stream-space (count-space S))
  proof (safe intro!: sets.sigma-sets-subset)
    fix xs assume ∀ x∈set xs. x ∈ S
    then have sstart S xs = {x∈space (stream-space (count-space S)). ∀ i<length
xs. x !! i = xs ! i}
      by (induction xs)
         (auto simp: space-stream-space nth-Cons split: nat.split intro: in-streams
```

streams-stl)
also have ... ∈ *sets* (*stream-space* (*count-space S*))
by *measurable*
finally show *sstart S xs* ∈ *sets* (*stream-space* (*count-space S*)) **.**
qed
then show *sets* (*sigma* (*streams S*) (*sstart S*‘*lists S* ∪ {{}})) ⊆ *sets* (*stream-space* (*count-space S*))
by (*simp add*: *space-stream-space*)
qed

lemma *Int-stable-sstart*: *Int-stable* (*sstart S*‘*lists S* ∪ {{}})
proof −
{ fix *xs ys* **assume** *xs* ∈ *lists S ys* ∈ *lists S*
then have *sstart S xs* ∩ *sstart S ys* ∈ *sstart S* ‘ *lists S* ∪ {{}}
proof (*induction xs ys rule*: *list-induct2*′)
case (*4 x xs y ys*)
show *?case*
proof *cases*
assume *x* = *y*
then have *sstart S* (*x* # *xs*) ∩ *sstart S* (*y* # *ys*) = (##) *x* ‘ (*sstart S xs* ∩ *sstart S ys*)
by (*auto simp*: *image-iff intro*!: *stream.collapse*[*symmetric*])
also have ... ∈ *sstart S* ‘ *lists S* ∪ {{}}
using *4* **by** (*auto simp*: *sstart.simps*(*2*)[*symmetric*] *del*: *in-listsD*)
finally show *?case* **.**
qed *auto*
qed (*simp-all add*: *sstart-in-streams inf.absorb1 inf.absorb2 image-eqI*[**where** *x*=[]]) **}**
then show *?thesis*
by (*auto simp*: *Int-stable-def*)
qed

lemma *stream-space-eq-sstart*:
assumes *S*[*simp*]: *countable S*
assumes *P*: *prob-space M prob-space N*
assumes *ae*: *AE x in M*. *x* ∈ *streams S AE x in N*. *x* ∈ *streams S*
assumes *sets-M*: *sets M* = *sets* (*stream-space* (*count-space UNIV*))
assumes *sets-N*: *sets N* = *sets* (*stream-space* (*count-space UNIV*))
assumes ∗: ⋀*xs*. *xs* ≠ [] ⟹ *xs* ∈ *lists S* ⟹ *emeasure M* (*sstart S xs*) = *emeasure N* (*sstart S xs*)
shows *M* = *N*
proof (*rule measure-eqI-restrict-generator*[*OF Int-stable-sstart*])
have [*simp*]: *sstart S* ‘ *lists S* ⊆ *Pow* (*streams S*)
by (*simp add*: *image-subset-iff sstart-in-streams*)

interpret *M*: *prob-space M* **by** *fact*

show *sstart S* ‘ *lists S* ∪ {{}} ⊆ *Pow* (*streams S*)
by (*auto dest*: *sstart-in-streams del*: *in-listsD*)

{ fix M :: *'a stream measure* **assume** M: *sets* M = *sets* (*stream-space* (*count-space UNIV*))
have *sets* (*restrict-space* M (*streams* S)) = *sigma-sets* (*streams* S) (*sstart* S ‘ *lists* $S \cup \{\{\}\}$)
by (*subst sets-restrict-space-cong*[*OF* M])
(*simp add*: *sets-restrict-stream-space restrict-count-space sets-stream-space-sstart*)
}
from *this*[*OF sets-M*] *this*[*OF sets-N*]
show *sets* (*restrict-space* M (*streams* S)) = *sigma-sets* (*streams* S) (*sstart* S ‘ *lists* $S \cup \{\{\}\}$)
sets (*restrict-space* N (*streams* S)) = *sigma-sets* (*streams* S) (*sstart* S ‘ *lists* $S \cup \{\{\}\}$)
by *auto*
show $\{$*streams* $S\} \subseteq$ *sstart* S ‘ *lists* $S \cup \{\{\}\}$
$\bigcup\{$*streams* $S\}$ = *streams* S $\bigwedge s.\ s \in \{$*streams* $S\} \Longrightarrow$ *emeasure* M $s \neq \infty$
using M.*emeasure-space-1 space-stream-space*[*of count-space* S] *sets-eq-imp-space-eq*[*OF sets-M*]
by (*auto simp add*: *image-eqI*[**where** x=[]])
show *sets* M = *sets* N
by (*simp add*: *sets-M sets-N*)
next
fix X **assume** $X \in$ *sstart* S ‘ *lists* $S \cup \{\{\}\}$
then obtain *xs* **where** $X = \{\} \vee (xs \in$ *lists* $S \wedge X =$ *sstart* S *xs*)
by *auto*
moreover have *emeasure* M (*streams* S) = *1*
using *ae* **by** (*intro prob-space.emeasure-eq-1-AE*[*OF P(1)*]) (*auto simp*: *sets-M streams-sets*)
moreover have *emeasure* N (*streams* S) = *1*
using *ae* **by** (*intro prob-space.emeasure-eq-1-AE*[*OF P(2)*]) (*auto simp*: *sets-N streams-sets*)
ultimately show *emeasure* M X = *emeasure* N X
using P[*THEN prob-space.emeasure-space-1*]
by (*cases xs* = []) (*auto simp*: ∗ *space-stream-space del*: *in-listsD*)
qed (*auto simp*: ∗ *ae sets-M del*: *in-listsD intro*!: *streams-sets*)

lemma *sets-sstart*[*measurable*]: *sstart* Ω *xs* ∈ *sets* (*stream-space* (*count-space UNIV*))
proof (*induction xs*)
case (*Cons x xs*)
note *this*[*measurable*]
have *sstart* Ω (*x* # *xs*) = {ω∈*space* (*stream-space* (*count-space UNIV*)). ω ∈ *sstart* Ω (*x* # *xs*)}
by (*auto simp*: *space-stream-space*)
also have ... ∈ *sets* (*stream-space* (*count-space UNIV*))
unfolding *in-sstart* **by** *measurable*
finally show *?case* **.**
qed (*auto intro*!: *streams-sets*)

primrec *scylinder* :: *'a set* ⇒ *'a set list* ⇒ *'a stream set*

where
scylinder S [] = *streams S*
| *scylinder S* (*A* # *As*) = {ω∈*streams S*. *shd* ω ∈ *A* ∧ *stl* ω ∈ *scylinder S As*}

lemma *scylinder-streams*: *scylinder S xs* ⊆ *streams S*
by (*induction xs*) *auto*

lemma *sets-scylinder*: (∀ *x*∈*set xs*. *x* ∈ *sets S*) ⟹ *scylinder* (*space S*) *xs* ∈ *sets* (*stream-space S*)
by (*induction xs*) (*auto simp*: *space-stream-space*[*symmetric*])

lemma *stream-space-eq-scylinder*:
assumes *P*: *prob-space M prob-space N*
assumes *Int-stable G* **and** *S*: *sets S* = *sets* (*sigma* (*space S*) *G*)
and *C*: *countable C C* ⊆ *G* ⋃ *C* = *space S* **and** *G*: *G* ⊆ *Pow* (*space S*)
assumes *sets-M*: *sets M* = *sets* (*stream-space S*)
assumes *sets-N*: *sets N* = *sets* (*stream-space S*)
assumes ∗: ⋀*xs*. *xs* ≠ [] ⟹ *xs* ∈ *lists G* ⟹ *emeasure M* (*scylinder* (*space S*) *xs*) = *emeasure N* (*scylinder* (*space S*) *xs*)
shows *M* = *N*
proof (*rule measure-eqI-generator-eq*)
interpret *M*: *prob-space M* **by** *fact*
interpret *N*: *prob-space N* **by** *fact*

let *?G* = *scylinder* (*space S*) ‘ *lists G*
show *sc-Pow*: *?G* ⊆ *Pow* (*streams* (*space S*))
using *scylinder-streams* **by** *auto*

have *sets* (*stream-space S*) = *sets* (*sigma* (*streams* (*space S*)) *?G*)
(**is** *?S* = *sets ?R*)
proof (*rule antisym*)
let *?V* = λ*i*. *vimage-algebra* (*streams* (*space S*)) (λ*s*. *s* !! *i*) *S*
show *?S* ⊆ *sets ?R*
unfolding *sets-stream-space-eq*
proof (*safe intro*!: *sets-Sup-in-sets del*: *subsetI equalityI*)
fix *i* :: *nat*
show *space* (*?V i*) = *space ?R*
using *scylinder-streams* **by** (*subst space-measure-of*) (*auto simp*:)
{ fix *A* **assume** *A* ∈ *G*
then have *scylinder* (*space S*) (*replicate i* (*space S*) @ [*A*]) = (λ*s*. *s* !! *i*) −‘ *A* ∩ *streams* (*space S*)
by (*induction i*) (*auto simp add*: *streams-shd streams-stl cong*: *conj-cong*)
also have *scylinder* (*space S*) (*replicate i* (*space S*) @ [*A*]) = (⋃ *xs*∈{*xs*∈*lists C*. *length xs* = *i*}. *scylinder* (*space S*) (*xs* @ [*A*]))
apply (*induction i*)
apply *auto* []
apply (*simp add*: *length-Suc-conv set-eq-iff ex-simps*(*1*,*2*)[*symmetric*] *cong*: *conj-cong del*: *ex-simps*(*1*,*2*))
apply *rule*

```
        subgoal for i x
          apply (cases x)
          apply (subst (2) C(3)[symmetric])
         apply (simp del: ex-simps(1,2) add: ex-simps(1,2)[symmetric] ac-simps
Bex-def)
          apply auto
          done
        done
       finally have (λs. s !! i) −‘ A ∩ streams (space S) = (⋃xs∈{xs∈lists C.
length xs = i}. scylinder (space S) (xs @ [A]))
        ..
      also have ... ∈ ?R
        using C(2) ‹A∈G›
        by (intro sets.countable-UN′ countable-Collect countable-lists C)
           (auto intro!: in-measure-of[OF sc-Pow] imageI)
      finally have (λs. s !! i) −‘ A ∩ streams (space S) ∈ ?R . }
    then show sets (?V i) ⊆ ?R
      apply (subst vimage-algebra-cong[OF refl refl S])
      apply (subst vimage-algebra-sigma[OF G])
      apply (simp add: streams-iff-snth) []
      apply (subst sigma-le-sets)
      apply auto
      done
  qed
  have G ⊆ sets S
    unfolding S using G by auto
  with C(2) show sets ?R ⊆ ?S
    unfolding sigma-le-sets[OF sc-Pow] by (auto intro!: sets-scylinder)
 qed
 then show sets M = sigma-sets (streams (space S)) (scylinder (space S) ‘ lists
G)
   sets N = sigma-sets (streams (space S)) (scylinder (space S) ‘ lists G)
   unfolding sets-M sets-N by (simp-all add: sc-Pow)

 show Int-stable ?G
 proof (rule Int-stableI-image)
   fix xs ys assume xs ∈ lists G ys ∈ lists G
    then show ∃zs∈lists G. scylinder (space S) xs ∩ scylinder (space S) ys =
scylinder (space S) zs
   proof (induction xs arbitrary: ys)
     case Nil then show ?case
       by (auto simp add: Int-absorb1 scylinder-streams)
   next
     case xs: (Cons x xs)
     show ?case
     proof (cases ys)
       case Nil with xs.hyps show ?thesis
         by (auto simp add: Int-absorb2 scylinder-streams intro!: bexI[of - x#xs])
     next
```

```
      case ys: (Cons y ys')
      with xs.IH[of ys'] xs.prems obtain zs where
        zs ∈ lists G and eq: scylinder (space S) xs ∩ scylinder (space S) ys' =
scylinder (space S) zs
       by auto
      show ?thesis
      proof (intro bexI[of - (x ∩ y)#zs])
       show x ∩ y # zs ∈ lists G
       using ‹zs∈lists G› ‹x∈G› ‹ys∈lists G› ys ‹Int-stable G›[THEN Int-stableD,
of x y] by auto
        show scylinder (space S) (x # xs) ∩ scylinder (space S) ys = scylinder
(space S) (x ∩ y # zs)
         by (auto simp add: eq[symmetric] ys)
       qed
     qed
   qed
  qed

  show range (λ-::nat. streams (space S)) ⊆ scylinder (space S) ‘ lists G
   (⋃i. streams (space S)) = streams (space S)
   emeasure M (streams (space S)) ≠ ∞
   by (auto intro!: image-eqI[of - - []])

  fix X assume X ∈ scylinder (space S) ‘ lists G
  then obtain xs where xs: xs ∈ lists G and eq: X = scylinder (space S) xs
   by auto
  then show emeasure M X = emeasure N X
  proof (cases xs = [])
   assume xs = [] then show ?thesis
    unfolding eq
    using sets-M[THEN sets-eq-imp-space-eq] sets-N[THEN sets-eq-imp-space-eq]
      M.emeasure-space-1 N.emeasure-space-1
    by (simp add: space-stream-space[symmetric])
  next
   assume xs ≠ [] with xs show ?thesis
    unfolding eq by (intro *)
  qed
qed

lemma stream-space-coinduct:
  fixes R :: 'a stream measure ⇒ 'a stream measure ⇒ bool
  assumes R A B
  assumes R: ⋀A B. R A B ⟹ ∃K∈space (prob-algebra M).
   ∃A'∈M →M prob-algebra (stream-space M). ∃B'∈M →M prob-algebra (stream-space
M).
    (AE y in K. R (A' y) (B' y) ∨ A' y = B' y) ∧
    A = do { y ← K; ω ← A' y; return (stream-space M) (y ## ω) } ∧
    B = do { y ← K; ω ← B' y; return (stream-space M) (y ## ω) }
  shows A = B
```

proof (*rule stream-space-eq-scylinder*)
let *?step* = $\lambda K\ L.$ *do* { $y \leftarrow K$; $\omega \leftarrow L\ y$; *return* (*stream-space M*) (y ## ω) }
{ fix $K\ A\ A'$ **assume** K: $K \in$ *space* (*prob-algebra M*)
and A'[*measurable*]: $A' \in M \rightarrow_M$ *prob-algebra* (*stream-space M*) **and** *A-eq*: A = *?step* $K\ A'$
have *ps*: *prob-space A*
unfolding *A-eq* **by** (*rule prob-space-bind'*[*OF K*]) *measurable*
have *sets A* = *sets* (*stream-space M*)
unfolding *A-eq* **by** (*rule sets-bind'*[*OF K*]) *measurable*
note *ps this* **}**
note ** = *this*

{ fix *A B* **assume** *R A B*
obtain *K A' B'* **where** *K*: $K \in$ *space* (*prob-algebra M*)
and *A'*: $A' \in M \rightarrow_M$ *prob-algebra* (*stream-space M*) A = *?step K A'*
and *B'*: $B' \in M \rightarrow_M$ *prob-algebra* (*stream-space M*) B = *?step K B'*
using *R*[*OF* ‹*R A B*›] **by** *blast*
have *prob-space A prob-space B sets A* = *sets* (*stream-space M*) *sets B* = *sets* (*stream-space M*)
using **[*OF K A'*] **[*OF K B'*] **by** *auto* **}**
note *R-D* = *this*

show *prob-space A prob-space B sets A* = *sets* (*stream-space M*) *sets B* = *sets* (*stream-space M*)
using *R-D*[*OF* ‹*R A B*›] **by** *auto*

show *Int-stable* (*sets M*) *sets M* = *sets* (*sigma* (*space M*) (*sets M*)) *countable* {*space M*}
{*space M*} $\subseteq$ *sets M* $\bigcup$ {*space M*} = *space M sets M* $\subseteq$ *Pow* (*space M*)
using *sets.space-closed*[*of M*] **by** (*auto simp*: *Int-stable-def*)

{ fix *A As L K* **assume** *K*[*measurable*]: $K \in$ *space* (*prob-algebra M*) **and** *A*: $A \in$ *sets M As* $\in$ *lists* (*sets M*)
and *L*[*measurable*]: $L \in M \rightarrow_M$ *prob-algebra* (*stream-space M*)
from *A* **have** [*measurable*]: $\forall x \in$ *set* (*A # As*). $x \in$ *sets M* $\forall x \in$ *set As*. $x \in$ *sets M*
by *auto*
have [*simp*]: *space K* = *space M sets K* = *sets M*
using *K* **by** (*auto simp*: *space-prob-algebra intro*!: *sets-eq-imp-space-eq*)
have [*simp*]: $x \in$ *space M* $\Longrightarrow$ *sets* (*L x*) = *sets* (*stream-space M*) **for** *x*
using *measurable-space*[*OF L*] **by** (*auto simp*: *space-prob-algebra*)
note *sets-scylinder*[*measurable*]
have $*$: *indicator* (*scylinder* (*space M*) (*A # As*)) (*x* ## ω) =
(*indicator A x* $*$ *indicator* (*scylinder* (*space M*) *As*) ω :: *ennreal*) **for** ω *x*
using *scylinder-streams*[*of space M As*] ‹*A* $\in$ *sets M*›[*THEN sets.sets-into-space*]
by (*auto split*: *split-indicator*)
have *emeasure* (*?step K L*) (*scylinder* (*space M*) (*A # As*)) = ($\int^+ y.$ *L y* (*scylinder* (*space M*) *As*) $*$ *indicator A y* ∂K)
apply (*subst emeasure-bind-prob-algebra*[*OF K*])

apply *measurable*
apply (*rule nn-integral-cong*)
apply (*subst emeasure-bind-prob-algebra*[*OF L*[*THEN measurable-space*]])
apply (*simp-all add*: *ac-simps* ∗ *nn-integral-cmult-indicator del*: *scylinder.simps*)
apply *measurable*
done }
note *emeasure-step* = *this*

fix *Xs* **assume** *Xs* ∈ *lists* (*sets M*)
from *this* ‹*R A B*› **show** *emeasure A* (*scylinder* (*space M*) *Xs*) = *emeasure B* (*scylinder* (*space M*) *Xs*)
proof (*induction Xs arbitrary*: *A B*)
case (*Cons X Xs*)
obtain *K A′ B′* **where** *K*: *K* ∈ *space* (*prob-algebra M*)
and *A′*[*measurable*]: *A′* ∈ *M* →$_M$ *prob-algebra* (*stream-space M*) **and** *A*: *A* = *?step K A′*
and *B′*[*measurable*]: *B′* ∈ *M* →$_M$ *prob-algebra* (*stream-space M*) **and** *B*: *B* = *?step K B′*
and *AE-R*: *AE x in K*. *R* (*A′ x*) (*B′ x*) ∨ *A′ x* = *B′ x*
using *R*[*OF* ‹*R A B*›] **by** *blast*

show *?case*
unfolding *A B emeasure-step*[*OF K Cons.hyps A′*] *emeasure-step*[*OF K Cons.hyps B′*]
apply (*rule nn-integral-cong-AE*)
using *AE-R* **by** *eventually-elim* (*auto simp add*: *Cons.IH*)
next
case *Nil*
note *R-D*[*OF this*]
from *this*(*1*,*2*)[*THEN prob-space.emeasure-space-1*] *this*(*3*,*4*)[*THEN sets-eq-imp-space-eq*]
show *?case*
by (*simp add*: *space-stream-space*)
qed
qed

end

theory *Tree-Space*
imports *HOL−Analysis.Analysis HOL−Library.Tree*
begin

lemma *countable-lfp*:
assumes *step*: ⋀*Y*. *countable Y* ⟹ *countable* (*F Y*)
and *cont*: *Order-Continuity.sup-continuous F*
shows *countable* (*lfp F*)
by(*subst sup-continuous-lfp*[*OF cont*])(*simp add*: *countable-funpow*[*OF step*])

lemma *countable-lfp-apply*:
assumes *step*: $\bigwedge Y\ x.\ (\bigwedge x.\ countable\ (Y\ x)) \Longrightarrow countable\ (F\ Y\ x)$
and *cont*: *Order-Continuity.sup-continuous F*
shows *countable* (*lfp F x*)
proof −
{ fix *n*
have $\bigwedge x.\ countable\ ((F \frown\frown n)\ bot\ x)$
by(*induct n*)(*auto intro*: *step*) **}**
thus *?thesis* **using** *cont* **by**(*simp add*: *sup-continuous-lfp*)
qed

inductive-set *trees* :: *'a set* ⇒ *'a tree set* **for** *S* :: *'a set* **where**
[*intro*!]: *Leaf* ∈ *trees S*
| *l* ∈ *trees S* ⟹ *r* ∈ *trees S* ⟹ *v* ∈ *S* ⟹ *Node l v r* ∈ *trees S*

lemma *Node-in-trees-iff*[*simp*]: *Node l v r* ∈ *trees S* ⟷ (*l* ∈ *trees S* ∧ *v* ∈ *S* ∧ *r* ∈ *trees S*)
by (*subst trees.simps*) *auto*

lemma *trees-sub-lfp*: *trees S* ⊆ *lfp* (λ*T*. *T* ∪ {*Leaf*} ∪ (⋃ *l*∈*T*. (⋃ *v*∈*S*. (⋃ *r*∈*T*. {*Node l v r*}))))
proof
have *mono*: *mono* (λ*T*. *T* ∪ {*Leaf*} ∪ (⋃ *l*∈*T*. (⋃ *v*∈*S*. (⋃ *r*∈*T*. {*Node l v r*})))) **by** (*auto simp*: *mono-def*)
fix *t* **assume** *t* ∈ *trees S* **then show** *t* ∈ *lfp* (λ*T*. *T* ∪ {*Leaf*} ∪ (⋃ *l*∈*T*. (⋃ *v*∈*S*. (⋃ *r*∈*T*. {*Node l v r*}))))
proof *induction*
case *1* **then show** *?case*
by (*subst lfp-unfold*[*OF mono*]) *auto*
next
case *2* **then show** *?case*
by (*subst lfp-unfold*[*OF mono*]) *auto*
qed
qed

lemma *countable-trees*: *countable A* ⟹ *countable* (*trees A*)
proof (*intro countable-subset*[*OF trees-sub-lfp*] *countable-lfp*
sup-continuous-sup sup-continuous-const sup-continuous-id)
show *sup-continuous* (λ*T*. (⋃ *l*∈*T*. ⋃ *v*∈*A*. ⋃ *r*∈*T*. {⟨*l*, *v*, *r*⟩}))
unfolding *sup-continuous-def*
proof (*intro allI impI equalityI subsetI*, *goal-cases*)
case (*1 M t*)
then obtain *i j* :: *nat* **and** *l x r* **where** *t* = *Node l x r x* ∈ *A l* ∈ *M i r* ∈ *M j*
by *auto*
hence *l* ∈ *M* (*max i j*) *r* ∈ *M* (*max i j*)
using *incseqD*[*OF* ‹*incseq M*›, *of i max i j*] *incseqD*[*OF* ‹*incseq M*›, *of j max i j*] **by** *auto*
with ‹*t* = *Node l x r*› **and** ‹*x* ∈ *A*› **show** *?case* **by** *auto*
qed *auto*

qed *auto*

lemma *trees-UNIV*[*simp*]: *trees UNIV* = *UNIV*
proof −
  have *t* ∈ *trees UNIV* **for** *t* :: *'a tree*
    by (*induction t*) (*auto intro*: *trees.intros*(*2*))
  then show *?thesis* **by** *auto*
qed

instance *tree* :: (*countable*) *countable*
proof
  have *countable* (*UNIV* :: *'a tree set*)
    by (*subst trees-UNIV*[*symmetric*]) (*intro countable-trees*[*OF countableI-type*])
  then show ∃ *to-nat*::*'a tree* ⇒ *nat*. *inj to-nat*
    by (*auto simp*: *countable-def*)
qed

lemma *map-in-trees*[*intro*]: (⋀*x*. *x* ∈ *set-tree t* ⟹ *f x* ∈ *S*) ⟹ *map-tree f t* ∈ *trees S*
  by (*induction t*) (*auto intro*: *trees.intros*(*2*))

primrec *trees-cyl* :: *'a set tree* ⇒ *'a tree set* **where**
  trees-cyl Leaf = {*Leaf*}
| *trees-cyl* (*Node l v r*) = (⋃*l'*∈*trees-cyl l*. (⋃*v'*∈*v*. (⋃*r'*∈*trees-cyl r*. {*Node l' v' r'*}))))

definition *tree-sigma* :: *'a measure* ⇒ *'a tree measure*
where
  tree-sigma M = *sigma* (*trees* (*space M*)) (*trees-cyl* ‘ *trees* (*sets M*))

lemma *Node-in-trees-cyl*: *Node l' v' r'* ∈ *trees-cyl t* ⟷
  (∃ *l v r*. *t* = *Node l v r* ∧ *l'* ∈ *trees-cyl l* ∧ *r'* ∈ *trees-cyl r* ∧ *v'* ∈ *v*)
  by (*cases t*) *auto*

lemma *trees-cyl-sub-trees*:
  assumes *t* ∈ *trees A A* ⊆ *Pow B* **shows** *trees-cyl t* ⊆ *trees B*
  using *assms*(*1*)
proof *induction*
  case (*2 l v r*) **with** ‹*A* ⊆ *Pow B*› **show** *?case*
    by (*auto intro*!: *trees.intros*(*2*))
qed *auto*

lemma *trees-cyl-sets-in-space*: *trees-cyl* ‘ *trees* (*sets M*) ⊆ *Pow* (*trees* (*space M*))
  using *trees-cyl-sub-trees*[*OF - sets.space-closed*, *of - M*] **by** *auto*

lemma *space-tree-sigma*: *space* (*tree-sigma M*) = *trees* (*space M*)
  unfolding *tree-sigma-def* **by** (*rule space-measure-of-conv*)

lemma *sets-tree-sigma-eq*: *sets* (*tree-sigma M*) = *sigma-sets* (*trees* (*space M*))

(*trees-cyl* ‘ *trees* (*sets M*))
unfolding *tree-sigma-def* **by** (*rule sets-measure-of*) (*rule trees-cyl-sets-in-space*)

lemma *Leaf-in-space-tree-sigma* [*measurable*, *simp*, *intro*]: *Leaf* ∈ *space* (*tree-sigma M*)
by (*auto simp*: *space-tree-sigma*)

lemma *Leaf-in-tree-sigma* [*measurable*, *simp*, *intro*]: {*Leaf*} ∈ *sets* (*tree-sigma M*)
unfolding *sets-tree-sigma-eq*
by (*rule sigma-sets.Basic*) (*auto intro*: *trees.intros*(*2*) *image-eqI*[**where** *x*=*Leaf*])

lemma *trees-cyl-map-treeI*: *t* ∈ *trees-cyl* (*map-tree* (λ*x*. *A*) *t*) **if** ∗: *t* ∈ *trees A*
using ∗ **by** *induction auto*

lemma *trees-cyl-map-in-sets*:
(⋀*x*. *x* ∈ *set-tree t* ⟹ *f x* ∈ *sets M*) ⟹ *trees-cyl* (*map-tree f t*) ∈ *sets* (*tree-sigma M*)
by (*subst sets-tree-sigma-eq*) *auto*

lemma *Node-in-tree-sigma*:
assumes *L*: *X* ∈ *sets* (*M* ⨂$_M$ (*tree-sigma M* ⨂$_M$ *tree-sigma M*))
shows {*Node l v r* | *l v r*. (*v*, *l*, *r*) ∈ *X*} ∈ *sets* (*tree-sigma M*)
proof −
let *?E* = λ*s*::*unit tree*. *trees-cyl* (*map-tree* (λ-. *space M*) *s*)
have *1*: *countable* (*range ?E*)
by (*intro countable-image countableI-type*)
have *2*: *trees-cyl* ‘ *trees* (*sets M*) ⊆ *Pow* (*space* (*tree-sigma M*))
using *trees-cyl-sets-in-space*[*of M*] **by** (*simp add*: *space-tree-sigma*)
have *3*: *sets* (*tree-sigma M*) = *sigma-sets* (*space* (*tree-sigma M*)) (*trees-cyl* ‘ *trees* (*sets M*))
unfolding *sets-tree-sigma-eq* **by** (*simp add*: *space-tree-sigma*)
have *4*: (⋃*s*. *?E s*) = *space* (*tree-sigma M*)
proof (*safe*; *clarsimp simp*: *space-tree-sigma*)
fix *t s* **assume** *t* ∈ *trees-cyl* (*map-tree* (λ-::*unit*. *space M*) *s*)
then show *t* ∈ *trees* (*space M*)
by (*induction s arbitrary*: *t*) *auto*
next
fix *t* **assume** *t* ∈ *trees* (*space M*)
then show ∃ *t′*. *t* ∈ *?E t′*
by (*intro exI*[*of* - *map-tree* (λ-. ()) *t*])
(*auto simp*: *tree.map-comp comp-def intro*: *trees-cyl-map-treeI*)
qed
have *5*: *range ?E* ⊆ *trees-cyl* ‘ *trees* (*sets M*) **by** *auto*
let *?P* = {*A* × *B* | *A B*. *A* ∈ *trees-cyl* ‘ *trees* (*sets M*) ∧ *B* ∈ *trees-cyl* ‘ *trees* (*sets M*)}
have *P*: *sets* (*tree-sigma M* ⨂$_M$ *tree-sigma M*) = *sets* (*sigma* (*space* (*tree-sigma M*) × *space* (*tree-sigma M*)) *?P*)
by (*rule sets-pair-eq*[*OF 2 3 1 5 4 2 3 1 5 4*])

have *sets* $(M \bigotimes_M (\textit{tree-sigma}\ M \bigotimes_M \textit{tree-sigma}\ M)) =$
sets (*sigma* (*space M* × *space* (*tree-sigma M* $\bigotimes_M$ *tree-sigma M*)) {*A* × *BC* |
A BC. *A* ∈ *sets M* ∧ *BC* ∈ *?P*})
proof (*rule sets-pair-eq*)
show *sets M* ⊆ *Pow* (*space M*) *sets M* = *sigma-sets* (*space M*) (*sets M*)
by (*auto simp*: *sets.sigma-sets-eq sets.space-closed*)
show *countable* {*space M*} {*space M*} ⊆ *sets M* ⋃{*space M*} = *space M*
by *auto*
show *?P* ⊆ *Pow* (*space* (*tree-sigma M* $\bigotimes_M$ *tree-sigma M*))
using *trees-cyl-sets-in-space*[*of M*]
by (*auto simp*: *space-pair-measure space-tree-sigma subset-eq*)
then show *sets* (*tree-sigma M* $\bigotimes_M$ *tree-sigma M*) =
sigma-sets (*space* (*tree-sigma M* $\bigotimes_M$ *tree-sigma M*)) *?P*
by (*subst P*, *subst sets-measure-of*) (*auto simp*: *space-tree-sigma space-pair-measure*)
show *countable* ((λ(*a*, *b*). *a* × *b*) ‘ (*range ?E* × *range ?E*))
by (*intro countable-image countable-SIGMA countableI-type*)
show (λ(*a*, *b*). *a* × *b*) ‘ (*range ?E* × *range ?E*) ⊆ *?P*
by *auto*
qed (*insert 4*, *auto simp*: *space-pair-measure space-tree-sigma set-eq-iff*)
also have ... = *sigma-sets* (*space M* × *trees* (*space M*) × *trees* (*space M*))
{*A* × *BC* |*A BC*. *A* ∈ *sets M* ∧ *BC* ∈ {*A* × *B* |*A B*.
A ∈ *trees-cyl* ‘ *trees* (*sets M*) ∧ *B* ∈ *trees-cyl* ‘ *trees* (*sets M*)}}
(**is** - = *sigma-sets ?X ?Y*) **using** *sets.space-closed*[*of M*] *trees-cyl-sub-trees*[*of* - *sets M space M*]
by (*subst sets-measure-of*)
(*auto simp*: *space-pair-measure space-tree-sigma*)
also have *?Y* = {*A* × *trees-cyl B* × *trees-cyl C* | *A B C*. *A* ∈ *sets M* ∧
B ∈ *trees* (*sets M*) ∧ *C* ∈ *trees* (*sets M*)} **by** *blast*
finally have *X* ∈ *sigma-sets* (*space M* × *trees* (*space M*) × *trees* (*space M*))
{*A* × *trees-cyl B* × *trees-cyl C* | *A B C*. *A* ∈ *sets M* ∧ *B* ∈ *trees* (*sets M*) ∧
C ∈ *trees* (*sets M*) }
using *assms* **by** *blast*
then show *?thesis*
proof *induction*
case (*Basic A'*)
then obtain *A B C* **where** *A'* = *A* × *trees-cyl B* × *trees-cyl C*
and ∗: *A* ∈ *sets M B* ∈ *trees* (*sets M*) *C* ∈ *trees* (*sets M*)
by *auto*
then have {*Node l v r* |*l v r*. (*v*, *l*, *r*) ∈ *A'*} = *trees-cyl* (*Node B A C*)
by *auto*
then show *?case*
by (*auto simp del*: *trees-cyl.simps simp*: *sets-tree-sigma-eq intro*!: *sigma-sets.Basic* ∗)
next
case *Empty* **show** *?case* **by** *auto*
next
case (*Compl A*)
have {*Node l v r* |*l v r*. (*v*, *l*, *r*) ∈ *space M* × *trees* (*space M*) × *trees* (*space M*) − *A*} =

```
    (space (tree-sigma M) − {Node l v r |l v r. (v, l, r) ∈ A}) − {Leaf}
     by (auto simp: space-tree-sigma elim: trees.cases)
   also have ... ∈ sets (tree-sigma M)
     by (intro sets.Diff Compl) auto
   finally show ?case .
 next
   case (Union I)
   have ∗: {Node l v r |l v r. (v, l, r) ∈ ⋃(I ‘ UNIV)} =
     (⋃i. {Node l v r |l v r. (v, l, r) ∈ I i}) by auto
   show ?case unfolding ∗ using Union(2) by (intro sets.countable-UN) auto
 qed
qed

lemma measurable-left[measurable]: left ∈ tree-sigma M →M tree-sigma M
proof (rule measurableI)
 show t ∈ space (tree-sigma M) ⟹ left t ∈ space (tree-sigma M) for t
   by (cases t) (auto simp: space-tree-sigma)
 fix A assume A: A ∈ sets (tree-sigma M)
 from sets.sets-into-space[OF this]
 have ∗: left −‘ A ∩ space (tree-sigma M) =
   (if Leaf ∈ A then {Leaf} else {}) ∪
   {Node a v r | a v r. (v, a, r) ∈ space M × A × space (tree-sigma M)}
   by (auto simp: space-tree-sigma elim: trees.cases)
 show left −‘ A ∩ space (tree-sigma M) ∈ sets (tree-sigma M)
  unfolding ∗ using A by (intro sets.Un Node-in-tree-sigma pair-measureI) auto
qed

lemma measurable-right[measurable]: right ∈ tree-sigma M →M tree-sigma M
proof (rule measurableI)
 show t ∈ space (tree-sigma M) ⟹ right t ∈ space (tree-sigma M) for t
   by (cases t) (auto simp: space-tree-sigma)
 fix A assume A: A ∈ sets (tree-sigma M)
 from sets.sets-into-space[OF this]
 have ∗: right −‘ A ∩ space (tree-sigma M) =
   (if Leaf ∈ A then {Leaf} else {}) ∪
   {Node l v a | l v a. (v, l, a) ∈ space M × space (tree-sigma M) × A}
   by (auto simp: space-tree-sigma elim: trees.cases)
 show right −‘ A ∩ space (tree-sigma M) ∈ sets (tree-sigma M)
  unfolding ∗ using A by (intro sets.Un Node-in-tree-sigma pair-measureI) auto
qed

lemma measurable-value′: value ∈ restrict-space (tree-sigma M) (−{Leaf}) →M
M
proof (rule measurableI)
 show t ∈ space (restrict-space (tree-sigma M) (− {Leaf})) ⟹ value t ∈ space
M for t
   by (cases t) (auto simp: space-restrict-space space-tree-sigma)
 fix A assume A: A ∈ sets M
 from sets.sets-into-space[OF this]
```

have *value* $-`$ $A \cap$ *space* (*restrict-space* (*tree-sigma M*) ($-$ {*Leaf*})) =
{*Node l a r* | *l a r*. (*a*, *l*, *r*) ∈ *A* × *space* (*tree-sigma M*) × *space* (*tree-sigma M*)}
by (*auto simp*: *space-tree-sigma space-restrict-space elim*: *trees.cases*)
also have ... ∈ *sets* (*tree-sigma M*)
using *A* **by** (*intro sets.Un Node-in-tree-sigma pair-measureI*) *auto*
finally show *value* $-`$ *A* ∩ *space* (*restrict-space* (*tree-sigma M*) ($-$ {*Leaf*})) ∈
sets (*restrict-space* (*tree-sigma M*) ($-$ {*Leaf*}))
by (*auto simp*: *sets-restrict-space-iff space-restrict-space*)
qed

lemma *measurable-value*[*measurable* (*raw*)]:
assumes $f \in X \rightarrow_M$ *tree-sigma M*
and $\bigwedge x.\ x \in$ *space X* $\Longrightarrow f\ x \neq$ *Leaf*
shows ($\lambda\omega.$ *value* (*f* ω)) ∈ $X \rightarrow_M$ *M*
proof −
from *assms* **have** $f \in X \rightarrow_M$ *restrict-space* (*tree-sigma M*) ($-$ {*Leaf*})
by (*intro measurable-restrict-space2*) *auto*
from *this* **and** *measurable-value′* **show** *?thesis* **by** (*rule measurable-compose*)
qed

lemma *measurable-Node* [*measurable*]:
(λ(*l*,*x*,*r*). *Node l x r*) ∈ *tree-sigma M* $\bigotimes_M$ *M* $\bigotimes_M$ *tree-sigma M* $\rightarrow_M$ *tree-sigma M*
proof (*rule measurable-sigma-sets*)
show *sets* (*tree-sigma M*) = *sigma-sets* (*trees* (*space M*)) (*trees-cyl* ‘ *trees* (*sets M*))
by (*simp add*: *sets-tree-sigma-eq*)
show *trees-cyl* ‘ *trees* (*sets M*) ⊆ *Pow* (*trees* (*space M*))
by (*rule trees-cyl-sets-in-space*)
show (λ(*l*, *x*, *r*). ⟨*l*, *x*, *r*⟩) ∈ *space* (*tree-sigma M* $\bigotimes_M$ *M* $\bigotimes_M$ *tree-sigma M*) → *trees* (*space M*)
by (*auto simp*: *space-pair-measure space-tree-sigma*)
fix *A* **assume** *t*: *A* ∈ *trees-cyl* ‘ *trees* (*sets M*)
then obtain *t* **where** *t*: *t* ∈ *trees* (*sets M*) *A* = *trees-cyl t* **by** *auto*
show (λ(*l*, *x*, *r*). ⟨*l*, *x*, *r*⟩) $-`$ *A* ∩
space (*tree-sigma M* $\bigotimes_M$ *M* $\bigotimes_M$ *tree-sigma M*)
∈ *sets* (*tree-sigma M* $\bigotimes_M$ *M* $\bigotimes_M$ *tree-sigma M*)
proof (*cases t*)
case *Leaf*
have (λ(*l*, *x*, *r*). ⟨*l*, *x*, *r*⟩) $-`$ {*Leaf* :: *′a tree*} = {} **by** *auto*
with *Leaf* **show** *?thesis* **using** *t* **by** *simp*
next
case (*Node l B r*)
hence (λ(*l*, *x*, *r*). ⟨*l*, *x*, *r*⟩) $-`$ *A* ∩ *space* (*tree-sigma M* $\bigotimes_M$ *M* $\bigotimes_M$ *tree-sigma M*) =
trees-cyl l × *B* × *trees-cyl r*
using *t* **and** *Node* **and** *trees-cyl-sub-trees*[*of* - *sets M space M*]

```
    by (auto simp: space-pair-measure space-tree-sigma
          dest: sets.sets-into-space[of - M])
  thus ?thesis using t and Node
    by (auto intro!: pair-measureI simp: sets-tree-sigma-eq)
 qed
qed

lemma measurable-Node' [measurable (raw)]:
 assumes [measurable]: l ∈ B →M tree-sigma A
 assumes [measurable]: x ∈ B →M A
 assumes [measurable]: r ∈ B →M tree-sigma A
 shows  (λy. Node (l y) (x y) (r y)) ∈ B →M tree-sigma A
proof −
 have (λy. Node (l y) (x y) (r y)) = (λ(a,b,c). Node a b c) ∘ (λy. (l y, x y, r y))
   by (simp add: o-def)
 also have ... ∈ B →M tree-sigma A
   by (intro measurable-comp[OF - measurable-Node]) simp-all
 finally show ?thesis .
qed

lemma measurable-rec-tree[measurable (raw)]:
 assumes t: t ∈ B →M tree-sigma M
 assumes l: l ∈ B →M A
 assumes n: (λ(x, l, v, r, al, ar). n x l v r al ar) ∈
   (B ⨂M tree-sigma M ⨂M M ⨂M tree-sigma M ⨂M A ⨂M A) →M A (is
?N ∈ ?M →M A)
 shows (λx. rec-tree (l x) (n x) (t x)) ∈ B →M A
proof (rule measurable-piecewise-restrict)
 let ?C = λt. λs::unit tree. t −‘ trees-cyl (map-tree (λ-. space M) s)
 show countable (range (?C t)) by (intro countable-image countableI-type)
 show space B ⊆ (⋃s. ?C t s)
 proof (safe; clarsimp)
   fix x assume x: x ∈ space B have t x ∈ trees (space M)
     using t[THEN measurable-space, OF x] by (simp add: space-tree-sigma)
   then show ∃xa::unit tree. t x ∈ trees-cyl (map-tree (λ-. space M) xa)
     by (intro exI[of - map-tree (λ-. ()) (t x)])
       (simp add: tree.map-comp comp-def trees-cyl-map-treeI)
 qed
 fix Ω assume Ω ∈ range (?C t)
 then obtain s :: unit tree where Ω: Ω = ?C t s by auto
 then show Ω ∩ space B ∈ sets B
   by (safe intro!: measurable-sets[OF t] trees-cyl-map-in-sets)
 show (λx. rec-tree (l x) (n x) (t x)) ∈ restrict-space B Ω →M A
   unfolding Ω using t
 proof (induction s arbitrary: t)
   case Leaf
   show ?case
   proof (rule measurable-cong[THEN iffD2])
     fix ω assume ω ∈ space (restrict-space B (?C t Leaf))
```

```
    then show rec-tree (l ω) (n ω) (t ω) = l ω
      by (auto simp: space-restrict-space)
  next
    show l ∈ restrict-space B (?C t Leaf) →M A
      using l by (rule measurable-restrict-space1)
  qed
next
  case (Node ls u rs)
  let ?F = λω. ?N (ω, left (t ω), value (t ω), right (t ω),
      rec-tree (l ω) (n ω) (left (t ω)), rec-tree (l ω) (n ω) (right (t ω)))
  show ?case
  proof (rule measurable-cong[THEN iffD2])
    fix ω assume ω ∈ space (restrict-space B (?C t (Node ls u rs)))
    then show rec-tree (l ω) (n ω) (t ω) = ?F ω
      by (auto simp: space-restrict-space)
  next
    show ?F ∈ (restrict-space B (?C t (Node ls u rs))) →M A
      apply (intro measurable-compose[OF - n] measurable-Pair[rotated])
      subgoal
        apply (rule measurable-restrict-mono[OF Node(2)])
        apply (rule measurable-compose[OF Node(3) measurable-right])
        by auto
      subgoal
        apply (rule measurable-restrict-mono[OF Node(1)])
        apply (rule measurable-compose[OF Node(3) measurable-left])
        by auto
      subgoal
        by (rule measurable-restrict-space1)
           (rule measurable-compose[OF Node(3) measurable-right])
      subgoal
        apply (rule measurable-compose[OF - measurable-value'])
        apply (rule measurable-restrict-space3[OF Node(3)])
        by auto
      subgoal
        by (rule measurable-restrict-space1)
           (rule measurable-compose[OF Node(3) measurable-left])
      by (rule measurable-restrict-space1) auto
  qed
qed
qed

lemma measurable-case-tree [measurable (raw)]:
  assumes t ∈ B →M tree-sigma M
  assumes l ∈ B →M A
  assumes (λ(x, l, v, r). n x l v r)
          ∈ B ⨂M tree-sigma M ⨂M M ⨂M tree-sigma M →M A
  shows   (λx. case-tree (l x) (n x) (t x)) ∈ B →M (A :: 'a measure)
proof −
  define n' where n' = (λx l v r (-::'a) (-::'a). n x l v r)
```

have (λx. *case-tree* (*l x*) (*n x*) (*t x*)) = (λx. *rec-tree* (*l x*) (*n′ x*) (*t x*))
(**is** - = (λx. *rec-tree* - (*?n′ x*) -)) **by** (*rule ext*) (*auto split*: *tree.splits simp*: *n′-def*)
also have ... $\in B \to_M A$
proof (*rule measurable-rec-tree*)
have ($\lambda(x, l, v, r, al, ar)$. *n′ x l v r al ar*) =
($\lambda(x,l,v,r)$. *n x l v r*) $\circ$ ($\lambda(x,l,v,r,al,ar)$. (*x*,*l*,*v*,*r*))
by (*simp add*: *n′-def o-def case-prod-unfold*)
also have ... $\in B \bigotimes_M$ *tree-sigma M* $\bigotimes_M$ *M* $\bigotimes_M$ *tree-sigma M* $\bigotimes_M$ *A* $\bigotimes_M$ *A* $\to_M$ *A*
using *assms*(*3*) **by** *measurable*
finally show ($\lambda(x, l, v, r, al, ar)$. *n′ x l v r al ar*) $\in$ ... **.**
qed (*insert assms*, *simp-all*)
finally show *?thesis* **.**
qed

hide-const (**open**) *left*
hide-const (**open**) *right*

end

28 Conditional Expectation

theory *Conditional-Expectation*
imports *Probability-Measure*
begin

28.1 Restricting a measure to a sub-sigma-algebra

definition *subalgebra*::*'a measure* ⇒ *'a measure* ⇒ *bool* **where**
subalgebra M F = ((*space F* = *space M*) ∧ (*sets F* ⊆ *sets M*))

lemma *sub-measure-space*:
assumes *i*: *subalgebra M F*
shows *measure-space* (*space M*) (*sets F*) (*emeasure M*)
proof −
have *sigma-algebra* (*space M*) (*sets F*)
by (*metis i measure-space measure-space-def subalgebra-def*)
moreover have *positive* (*sets F*) (*emeasure M*)
using *Sigma-Algebra.positive-def* **by** *auto*
moreover have *countably-additive* (*sets F*) (*emeasure M*)
by (*meson countably-additive-def emeasure-countably-additive i order-trans subalgebra-def subsetCE*)
ultimately show *?thesis* **unfolding** *measure-space-def* **by** *simp*
qed

definition *restr-to-subalg*::*'a measure* ⇒ *'a measure* ⇒ *'a measure* **where**
restr-to-subalg M F = *measure-of* (*space M*) (*sets F*) (*emeasure M*)

lemma *space-restr-to-subalg*:
space (*restr-to-subalg M F*) = *space M*
unfolding *restr-to-subalg-def* **by** (*simp add*: *space-measure-of-conv*)

lemma *sets-restr-to-subalg* [*measurable-cong*]:
assumes *subalgebra M F*
shows *sets* (*restr-to-subalg M F*) = *sets F*
unfolding *restr-to-subalg-def* **by** (*metis sets.sets-measure-of-eq assms subalgebra-def*)

lemma *emeasure-restr-to-subalg*:
assumes *subalgebra M F*
A ∈ *sets F*
shows *emeasure* (*restr-to-subalg M F*) *A* = *emeasure M A*
unfolding *restr-to-subalg-def*
by (*metis assms subalgebra-def emeasure-measure-of-conv sub-measure-space sets.sigma-sets-eq*)

lemma *null-sets-restr-to-subalg*:
assumes *subalgebra M F*
shows *null-sets* (*restr-to-subalg M F*) = *null-sets M* ∩ *sets F*
proof
have *x* ∈ *null-sets M* ∩ *sets F* **if** *x* ∈ *null-sets* (*restr-to-subalg M F*) **for** *x*
by (*metis that Int-iff assms emeasure-restr-to-subalg null-setsD1 null-setsD2 null-setsI*
sets-restr-to-subalg subalgebra-def subsetD)
then show *null-sets* (*restr-to-subalg M F*) ⊆ *null-sets M* ∩ *sets F* **by** *auto*
next
have *x* ∈ *null-sets* (*restr-to-subalg M F*) **if** *x* ∈ *null-sets M* ∩ *sets F* **for** *x*
by (*metis that Int-iff assms null-setsD1 null-setsI sets-restr-to-subalg emeasure-restr-to-subalg*[*OF assms*])
then show *null-sets M* ∩ *sets F* ⊆ *null-sets* (*restr-to-subalg M F*) **by** *auto*
qed

lemma *AE-restr-to-subalg*:
assumes *subalgebra M F*
AE x in (*restr-to-subalg M F*). *P x*
shows *AE x in M*. *P x*
proof −
obtain *A* **where** *A*: ⋀*x*. *x* ∈ *space* (*restr-to-subalg M F*) − *A* ⟹ *P x A* ∈ *null-sets* (*restr-to-subalg M F*)
using *AE-E3*[*OF assms*(*2*)] **by** *auto*
then have *A* ∈ *null-sets M* **using** *null-sets-restr-to-subalg*[*OF assms*(*1*)] **by** *auto*
moreover have ⋀*x*. *x* ∈ *space M* − *A* ⟹ *P x*
using *space-restr-to-subalg A*(*1*) **by** *fastforce*
ultimately show *?thesis*
unfolding *eventually-ae-filter* **by** *auto*
qed

lemma *AE-restr-to-subalg2*:
assumes *subalgebra M F*

AE x in M. P x **and** [*measurable*]: *P ∈ measurable F* (*count-space UNIV*)
shows *AE x in* (*restr-to-subalg M F*). *P x*
proof −
define *U* **where** *U* = {*x ∈ space M*. ¬(*P x*)}
then have ∗: *U* = {*x ∈ space F*. ¬(*P x*)} **using** *assms*(*1*) **by** (*simp add*: *subalgebra-def*)
then have *U ∈ sets F* **by** *simp*
then have *U ∈ sets M* **using** *assms*(*1*) **by** (*meson subalgebra-def subsetD*)
then have *U ∈ null-sets M* **unfolding** *U-def* **using** *assms*(*2*) **using** *AE-iff-measurable* **by** *blast*
then have *U ∈ null-sets* (*restr-to-subalg M F*) **using** *null-sets-restr-to-subalg*[*OF assms*(*1*)] ‹*U ∈ sets F*› **by** *auto*
then show *?thesis* **using** ∗ **by** (*metis* (*no-types*, *lifting*) *Collect-mono U-def eventually-ae-filter space-restr-to-subalg*)
qed

lemma *prob-space-restr-to-subalg*:
assumes *subalgebra M F*
prob-space M
shows *prob-space* (*restr-to-subalg M F*)
by (*metis* (*no-types*, *lifting*) *assms*(*1*) *assms*(*2*) *emeasure-restr-to-subalg prob-space.emeasure-space-1 prob-spaceI*
sets.top space-restr-to-subalg subalgebra-def)

lemma *finite-measure-restr-to-subalg*:
assumes *subalgebra M F*
finite-measure M
shows *finite-measure* (*restr-to-subalg M F*)
by (*metis* (*no-types*, *lifting*) *assms emeasure-restr-to-subalg finite-measure.finite-emeasure-space finite-measureI sets.top space-restr-to-subalg subalgebra-def infinity-ennreal-def*)

lemma *measurable-in-subalg*:
assumes *subalgebra M F*
f ∈ measurable F N
shows *f ∈ measurable* (*restr-to-subalg M F*) *N*
by (*metis measurable-cong-sets assms*(*2*) *sets-restr-to-subalg*[*OF assms*(*1*)])

lemma *measurable-in-subalg′*:
assumes *subalgebra M F*
f ∈ measurable (*restr-to-subalg M F*) *N*
shows *f ∈ measurable F N*
by (*metis measurable-cong-sets assms*(*2*) *sets-restr-to-subalg*[*OF assms*(*1*)])

lemma *measurable-from-subalg*:
assumes *subalgebra M F*
f ∈ measurable F N
shows *f ∈ measurable M N*
using *assms* **unfolding** *measurable-def subalgebra-def* **by** *auto*

The following is the direct transposition of `nn_integral_subalgebra` (from

`Nonnegative_Lebesgue_Integration`) in the current notations, with the removal of the useless assumption $f \geq 0$.

lemma *nn-integral-subalgebra2*:
assumes *subalgebra M F* **and** [*measurable*]: $f \in$ *borel-measurable F*
shows $(\int^+ x.\ f\,x\ \partial(\textit{restr-to-subalg}\ M\ F)) = (\int^+ x.\ f\,x\ \partial M)$
proof (*rule nn-integral-subalgebra*)
show $f \in$ *borel-measurable* (*restr-to-subalg M F*)
by (*rule measurable-in-subalg*[*OF assms*(*1*)]) *simp*
show *sets* (*restr-to-subalg M F*) $\subseteq$ *sets M* **by** (*metis sets-restr-to-subalg*[*OF assms*(*1*)] *assms*(*1*) *subalgebra-def*)
fix *A* **assume** $A \in$ *sets* (*restr-to-subalg M F*)
then show *emeasure* (*restr-to-subalg M F*) *A* = *emeasure M A*
by (*metis sets-restr-to-subalg*[*OF assms*(*1*)] *emeasure-restr-to-subalg*[*OF assms*(*1*)])
qed (*auto simp add*: *assms space-restr-to-subalg sets-restr-to-subalg*[*OF assms*(*1*)])

The following is the direct transposition of `integral_subalgebra` (from `Bochner_Integration`) in the current notations.

lemma *integral-subalgebra2*:
fixes *f* :: *'a* ⇒ *'b*::{*banach*, *second-countable-topology*}
assumes *subalgebra M F* **and**
[*measurable*]: $f \in$ *borel-measurable F*
shows $(\int x.\ f\,x\ \partial(\textit{restr-to-subalg}\ M\ F)) = (\int x.\ f\,x\ \partial M)$
by (*rule integral-subalgebra*,
metis measurable-in-subalg[*OF assms*(*1*)] *assms*(*2*),
auto simp add: *assms space-restr-to-subalg sets-restr-to-subalg emeasure-restr-to-subalg*,
meson assms(*1*) *subalgebra-def subset-eq*)

lemma *integrable-from-subalg*:
fixes *f* :: *'a* ⇒ *'b*::{*banach*, *second-countable-topology*}
assumes *subalgebra M F*
integrable (*restr-to-subalg M F*) *f*
shows *integrable M f*
proof (*rule integrableI-bounded*)
have [*measurable*]: $f \in$ *borel-measurable F* **using** *assms* **by** *auto*
then show $f \in$ *borel-measurable M* **using** *assms*(*1*) *measurable-from-subalg* **by** *blast*

have $(\int^+ x.\ \textit{ennreal}\ (\textit{norm}\ (f\,x))\ \partial M) = (\int^+ x.\ \textit{ennreal}\ (\textit{norm}\ (f\,x))\ \partial(\textit{restr-to-subalg}\ M\ F))$
by (*rule nn-integral-subalgebra2*[*symmetric*], *auto simp add*: *assms*)
also have ... $< \infty$ **using** *integrable-iff-bounded assms* **by** *auto*
finally show $(\int^+ x.\ \textit{ennreal}\ (\textit{norm}\ (f\,x))\ \partial M) < \infty$ **by** *simp*
qed

lemma *integrable-in-subalg*:
fixes *f* :: *'a* ⇒ *'b*::{*banach*, *second-countable-topology*}
assumes [*measurable*]: *subalgebra M F*
$f \in$ *borel-measurable F*
integrable M f

shows *integrable* (*restr-to-subalg M F*) *f*
proof (*rule integrableI-bounded*)
show *f* ∈ *borel-measurable* (*restr-to-subalg M F*) **using** *assms*(*2*) *assms*(*1*) **by** *auto*
have (∫⁺ *x*. *ennreal* (*norm* (*f x*)) ∂(*restr-to-subalg M F*)) = (∫⁺ *x*. *ennreal* (*norm* (*f x*)) ∂*M*)
by (*rule nn-integral-subalgebra2*, *auto simp add*: *assms*)
also have ... < ∞ **using** *integrable-iff-bounded assms* **by** *auto*
finally show (∫⁺ *x*. *ennreal* (*norm* (*f x*)) ∂(*restr-to-subalg M F*)) < ∞ **by** *simp*
qed

28.2 Nonnegative conditional expectation

The conditional expectation of a function f, on a measure space M, with respect to a sub sigma algebra F, should be a function g which is F-measurable whose integral on any F-set coincides with the integral of f. Such a function is uniquely defined almost everywhere. The most direct construction is to use the measure fdM, restrict it to the sigma-algebra F, and apply the Radon-Nikodym theorem to write it as $gdM_{|F}$ for some F-measurable function g. Another classical construction for L^2 functions is done by orthogonal projection on F-measurable functions, and then extending by density to L^1. The Radon-Nikodym point of view avoids the L^2 machinery, and works for all positive functions.

In this paragraph, we develop the definition and basic properties for nonnegative functions, as the basics of the general case. As in the definition of integrals, the nonnegative case is done with ennreal-valued functions, without any integrability assumption.

definition *nn-cond-exp* :: *'a measure* ⇒ *'a measure* ⇒ (*'a* ⇒ *ennreal*) ⇒ (*'a* ⇒ *ennreal*)
where
nn-cond-exp M F f =
(*if f* ∈ *borel-measurable M* ∧ *subalgebra M F*
then RN-deriv (*restr-to-subalg M F*) (*restr-to-subalg* (*density M f*) *F*)
else (λ-. *0*))

lemma
shows *borel-measurable-nn-cond-exp* [*measurable*]: *nn-cond-exp M F f* ∈ *borel-measurable F*
and *borel-measurable-nn-cond-exp2* [*measurable*]: *nn-cond-exp M F f* ∈ *borel-measurable M*
by (*simp-all add*: *nn-cond-exp-def*)
(*metis borel-measurable-RN-deriv borel-measurable-subalgebra sets-restr-to-subalg space-restr-to-subalg subalgebra-def*)

The good setting for conditional expectations is the situation where the subalgebra F gives rise to a sigma-finite measure space. To see what goes wrong if it is not sigma-finite, think of $\mathbb{R}$ with the trivial sigma-algebra

$\{\emptyset, \mathbb{R}\}$. In this case, conditional expectations have to be constant functions, so they have integral 0 or ∞. This means that a positive integrable function can have no meaningful conditional expectation.

locale *sigma-finite-subalgebra* =
  fixes *M F*::*'a measure*
  assumes *subalg*: *subalgebra M F*
    and *sigma-fin-subalg*: *sigma-finite-measure* (*restr-to-subalg M F*)

lemma *sigma-finite-subalgebra-is-sigma-finite*:
  assumes *sigma-finite-subalgebra M F*
  shows *sigma-finite-measure M*
proof
  have *subalg*: *subalgebra M F*
   and *sigma-fin-subalg*: *sigma-finite-measure* (*restr-to-subalg M F*)
    using *assms* **unfolding** *sigma-finite-subalgebra-def* **by** *auto*
  obtain *A* **where** *Ap*: *countable A* ∧ *A* ⊆ *sets* (*restr-to-subalg M F*) ∧ ⋃ *A* = *space* (*restr-to-subalg M F*) ∧ (∀ *a*∈*A*. *emeasure* (*restr-to-subalg M F*) *a* ≠ ∞)
   using *sigma-finite-measure.sigma-finite-countable*[*OF sigma-fin-subalg*] **by** *fastforce*
  have *A* ⊆ *sets F* **using** *Ap sets-restr-to-subalg*[*OF subalg*] **by** *fastforce*
  then have *A* ⊆ *sets M* **using** *subalg subalgebra-def* **by** *force*
  moreover have ⋃ *A* = *space M* **using** *Ap space-restr-to-subalg* **by** *simp*
 moreover have ∀ *a*∈*A*. *emeasure M a* ≠ ∞ **by** (*metis subsetD emeasure-restr-to-subalg*[*OF subalg*] ‹*A* ⊆ *sets F*› *Ap*)
  ultimately show ∃ *A*. *countable A* ∧ *A* ⊆ *sets M* ∧ ⋃ *A* = *space M* ∧ (∀ *a*∈*A*. *emeasure M a* ≠ ∞) **using** *Ap* **by** *auto*
qed

sublocale *sigma-finite-subalgebra* ⊆ *sigma-finite-measure*
using *sigma-finite-subalgebra-is-sigma-finite sigma-finite-subalgebra-axioms* **by** *blast*

Conditional expectations are very often used in probability spaces. This is a special case of the previous one, as we prove now.

locale *finite-measure-subalgebra* = *finite-measure* +
  fixes *F*::*'a measure*
  assumes *subalg*: *subalgebra M F*

lemma *finite-measure-subalgebra-is-sigma-finite*:
  assumes *finite-measure-subalgebra M F*
  shows *sigma-finite-subalgebra M F*
proof −
  interpret *finite-measure-subalgebra M F* **using** *assms* **by** *simp*
  have *finite-measure* (*restr-to-subalg M F*)
   using *finite-measure-restr-to-subalg subalg finite-emeasure-space finite-measureI*
unfolding *infinity-ennreal-def* **by** *blast*
  then have *sigma-finite-measure* (*restr-to-subalg M F*)
   unfolding *finite-measure-def* **by** *simp*
  then show *sigma-finite-subalgebra M F* **unfolding** *sigma-finite-subalgebra-def*
using *subalg* **by** *simp*

qed

sublocale *finite-measure-subalgebra* ⊆ *sigma-finite-subalgebra*
proof −
have *finite-measure* (*restr-to-subalg M F*)
using *finite-measure-restr-to-subalg subalg finite-emeasure-space finite-measureI*
unfolding *infinity-ennreal-def* **by** *blast*
then have *sigma-finite-measure* (*restr-to-subalg M F*)
unfolding *finite-measure-def* **by** *simp*
then show *sigma-finite-subalgebra M F* **unfolding** *sigma-finite-subalgebra-def*
using *subalg* **by** *simp*
qed

context *sigma-finite-subalgebra*
begin

The next lemma is arguably the most fundamental property of conditional expectation: when computing an expectation against an F-measurable function, it is equivalent to work with a function or with its F-conditional expectation.

This property (even for bounded test functions) characterizes conditional expectations, as the second lemma below shows. From this point on, we will only work with it, and forget completely about the definition using Radon-Nikodym derivatives.

lemma *nn-cond-exp-intg*:
assumes [*measurable*]: *f* ∈ *borel-measurable F g* ∈ *borel-measurable M*
shows (∫⁺ *x*. *f x* ∗ *nn-cond-exp M F g x* ∂*M*) = (∫⁺ *x*. *f x* ∗ *g x* ∂*M*)
proof −
have [*measurable*]: *f* ∈ *borel-measurable M*
by (*meson assms subalg borel-measurable-subalgebra subalgebra-def*)
have *ac*: *absolutely-continuous* (*restr-to-subalg M F*) (*restr-to-subalg* (*density M g*) *F*)
unfolding *absolutely-continuous-def*
proof −
have *null-sets* (*restr-to-subalg M F*) = *null-sets M* ∩ *sets F* **by** (*rule null-sets-restr-to-subalg*[*OF subalg*])
moreover have *null-sets M* ⊆ *null-sets* (*density M g*)
by (*rule absolutely-continuousI-density*[*unfolded absolutely-continuous-def*]) *auto*
ultimately have *null-sets* (*restr-to-subalg M F*) ⊆ *null-sets* (*density M g*) ∩ *sets F* **by** *auto*
moreover have *null-sets* (*density M g*) ∩ *sets F* = *null-sets* (*restr-to-subalg* (*density M g*) *F*)
by (*rule null-sets-restr-to-subalg*[*symmetric*]) (*metis subalg sets-density space-density subalgebra-def*)
ultimately show *null-sets* (*restr-to-subalg M F*) ⊆ *null-sets* (*restr-to-subalg* (*density M g*) *F*) **by** *auto*
qed

have $(\int^+ x.\ f\ x * \textit{nn-cond-exp}\ M\ F\ g\ x\ \partial M) = (\int^+ x.\ f\ x * \textit{nn-cond-exp}\ M\ F\ g\ x\ \partial(\textit{restr-to-subalg}\ M\ F))$
by (*rule nn-integral-subalgebra2*[*symmetric*]) (*simp-all add*: *assms subalg*)
also have ... $= (\int^+ x.\ f\ x * \textit{RN-deriv}\ (\textit{restr-to-subalg}\ M\ F)\ (\textit{restr-to-subalg}\ (\textit{density}\ M\ g)\ F)\ x\ \partial(\textit{restr-to-subalg}\ M\ F))$
unfolding *nn-cond-exp-def* **using** *assms subalg* **by** *simp*
also have ... $= (\int^+ x.\ \textit{RN-deriv}\ (\textit{restr-to-subalg}\ M\ F)\ (\textit{restr-to-subalg}\ (\textit{density}\ M\ g)\ F)\ x * f\ x\ \partial(\textit{restr-to-subalg}\ M\ F))$
by (*simp add*: *mult.commute*)
also have ... $= (\int^+ x.\ f\ x\ \partial(\textit{restr-to-subalg}\ (\textit{density}\ M\ g)\ F))$
proof (*rule sigma-finite-measure.RN-deriv-nn-integral*[*symmetric*])
show *sets* $(\textit{restr-to-subalg}\ (\textit{density}\ M\ g)\ F) = \textit{sets}\ (\textit{restr-to-subalg}\ M\ F)$
by (*metis subalg restr-to-subalg-def sets.sets-measure-of-eq space-density subalgebra-def*)
qed (*auto simp add*: *assms measurable-restrict ac measurable-in-subalg subalg sigma-fin-subalg*)
also have ... $= (\int^+ x.\ f\ x\ \partial(\textit{density}\ M\ g))$
by (*metis nn-integral-subalgebra2 subalg assms*(*1*) *sets-density space-density subalgebra-def*)
also have ... $= (\int^+ x.\ g\ x * f\ x\ \partial M)$
by (*rule nn-integral-density*) (*simp-all add*: *assms*)
also have ... $= (\int^+ x.\ f\ x * g\ x\ \partial M)$
by (*simp add*: *mult.commute*)
finally show *?thesis* **by** *simp*
qed

lemma *nn-cond-exp-charact*:
assumes $\bigwedge A.\ A \in \textit{sets}\ F \Longrightarrow (\int^+ x \in A.\ f\ x\ \partial M) = (\int^+ x \in A.\ g\ x\ \partial M)$ **and**
[*measurable*]: $f \in \textit{borel-measurable}\ M\ g \in \textit{borel-measurable}\ F$
shows *AE x in M*. $g\ x = \textit{nn-cond-exp}\ M\ F\ f\ x$
proof −
let *?MF* = *restr-to-subalg M F*
{
fix *A* **assume** $A \in \textit{sets}$ *?MF*
then have [*measurable*]: $A \in \textit{sets}\ F$ **using** *sets-restr-to-subalg*[*OF subalg*] **by** *simp*
have $(\int^+ x \in A.\ g\ x\ \partial$ *?MF*$) = (\int^+ x \in A.\ g\ x\ \partial M)$
by (*simp add*: *nn-integral-subalgebra2 subalg*)
also have ... $= (\int^+ x \in A.\ f\ x\ \partial M)$ **using** *assms*(*1*) **by** *simp*
also have ... $= (\int^+ x.\ \textit{indicator}\ A\ x * f\ x\ \partial M)$ **by** (*simp add*: *mult.commute*)
also have ... $= (\int^+ x.\ \textit{indicator}\ A\ x * \textit{nn-cond-exp}\ M\ F\ f\ x\ \partial M)$
by (*rule nn-cond-exp-intg*[*symmetric*]) (*auto simp add*: *assms*)
also have ... $= (\int^+ x \in A.\ \textit{nn-cond-exp}\ M\ F\ f\ x\ \partial M)$ **by** (*simp add*: *mult.commute*)
also have ... $= (\int^+ x \in A.\ \textit{nn-cond-exp}\ M\ F\ f\ x\ \partial$ *?MF*$)$
by (*simp add*: *nn-integral-subalgebra2 subalg*)
finally have $(\int^+ x \in A.\ g\ x\ \partial$ *?MF*$) = (\int^+ x \in A.\ \textit{nn-cond-exp}\ M\ F\ f\ x\ \partial$ *?MF*$)$ **by** *simp*

} note $* = this$
have *AE x in ?MF. g x = nn-cond-exp M F f x*
by (*rule sigma-finite-measure.density-unique2*)
(*auto simp add*: *assms subalg sigma-fin-subalg AE-restr-to-subalg2* $*$)
then show *?thesis* **using** *AE-restr-to-subalg*[*OF subalg*] **by** *simp*
qed

lemma *nn-cond-exp-F-meas*:
assumes $f \in$ *borel-measurable F*
shows *AE x in M. f x = nn-cond-exp M F f x*
by (*rule nn-cond-exp-charact*) (*auto simp add*: *assms measurable-from-subalg*[*OF subalg*])

lemma *nn-cond-exp-prod*:
assumes [*measurable*]: $f \in$ *borel-measurable F* $g \in$ *borel-measurable M*
shows *AE x in M. f x * nn-cond-exp M F g x = nn-cond-exp M F* ($\lambda x.$ *f x * g x*) *x*
proof (*rule nn-cond-exp-charact*)
have [*measurable*]: $f \in$ *borel-measurable M* **by** (*rule measurable-from-subalg*[*OF subalg assms*(*1*)])
show ($\lambda x.$ *f x * g x*) $\in$ *borel-measurable M* **by** *measurable*

fix *A* **assume** $A \in$ *sets F*
then have [*measurable*]: ($\lambda x.$ *f x * indicator A x*) $\in$ *borel-measurable F* **by** *measurable*
have $\int^+ x \in A.$ (*f x * g x*) $\partial M = \int^+ x.$ (*f x * indicator A x*) * *g x* ∂M
by (*simp add*: *mult.commute mult.left-commute*)
also have ... $= \int^+ x.$ (*f x * indicator A x*) * *nn-cond-exp M F g x* ∂M
by (*rule nn-cond-exp-intg*[*symmetric*]) (*auto simp add*: *assms*)
also have ... $= \int^+ x \in A.$ (*f x * nn-cond-exp M F g x*)∂M
by (*simp add*: *mult.commute mult.left-commute*)
finally show $\int^+ x \in A.$ (*f x * g x*) $\partial M = \int^+ x \in A.$ (*f x * nn-cond-exp M F g x*)∂M
by *simp*
qed (*auto simp add*: *assms*)

lemma *nn-cond-exp-sum*:
assumes [*measurable*]: $f \in$ *borel-measurable M* $g \in$ *borel-measurable M*
shows *AE x in M. nn-cond-exp M F f x + nn-cond-exp M F g x = nn-cond-exp M F* ($\lambda x.$ *f x + g x*) *x*
proof (*rule nn-cond-exp-charact*)
fix *A* **assume** [*measurable*]: $A \in$ *sets F*
then have $A \in$ *sets M* **by** (*meson subalg subalgebra-def subsetD*)
have $\int^+ x \in A.$ (*nn-cond-exp M F f x + nn-cond-exp M F g x*)∂M = ($\int^+ x \in A.$ *nn-cond-exp M F f x* ∂M) + ($\int^+ x \in A.$ *nn-cond-exp M F g x* ∂M)
by (*rule nn-set-integral-add*) (*auto simp add*: *assms* ‹$A \in$ *sets M*›)
also have ... = ($\int^+ x.$ *indicator A x * nn-cond-exp M F f x* ∂M) + ($\int^+ x.$ *indicator A x * nn-cond-exp M F g x* ∂M)
by (*metis* (*no-types*, *lifting*) *mult.commute nn-integral-cong*)
also have ... = ($\int^+ x.$ *indicator A x * f x* ∂M) + ($\int^+ x.$ *indicator A x * g x* ∂M)

by (*simp add*: *nn-cond-exp-intg*)
also have ... = $(\int^+ x{\in}A.\ f\ x\ \partial M) + (\int^+ x{\in}A.\ g\ x\ \partial M)$
by (*metis* (*no-types*, *lifting*) *mult.commute nn-integral-cong*)
also have ... = $\int^+ x{\in}A.\ (f\ x + g\ x)\partial M$
by (*rule nn-set-integral-add*[*symmetric*]) (*auto simp add*: *assms* ‹$A \in$ *sets* M›)
finally show $\int^+ x{\in}A.\ (f\ x + g\ x)\partial M = \int^+ x{\in}A.\ (nn\text{-}cond\text{-}exp\ M\ F\ f\ x + nn\text{-}cond\text{-}exp\ M\ F\ g\ x)\partial M$
by *simp*
qed (*auto simp add*: *assms*)

lemma *nn-cond-exp-cong*:
assumes *AE x in M*. $f\ x = g\ x$
and [*measurable*]: $f \in$ *borel-measurable M* $g \in$ *borel-measurable M*
shows *AE x in M*. *nn-cond-exp M F f x* = *nn-cond-exp M F g x*
proof (*rule nn-cond-exp-charact*)
fix *A* **assume** [*measurable*]: $A \in$ *sets F*
have $\int^+ x{\in}A.\ nn\text{-}cond\text{-}exp\ M\ F\ f\ x\ \partial M = \int^+ x.\ indicator\ A\ x * nn\text{-}cond\text{-}exp\ M\ F\ f\ x\ \partial M$
by (*simp add*: *mult.commute*)
also have ... = $\int^+ x.\ indicator\ A\ x * f\ x\ \partial M$ **by** (*simp add*: *nn-cond-exp-intg assms*)
also have ... = $\int^+ x{\in}A.\ f\ x\ \partial M$ **by** (*simp add*: *mult.commute*)
also have ... = $\int^+ x{\in}A.\ g\ x\ \partial M$ **by** (*rule nn-set-integral-cong*[*OF assms*(*1*)])
finally show $\int^+ x{\in}A.\ g\ x\ \partial M = \int^+ x{\in}A.\ nn\text{-}cond\text{-}exp\ M\ F\ f\ x\ \partial M$ **by** *simp*
qed (*auto simp add*: *assms*)

lemma *nn-cond-exp-mono*:
assumes *AE x in M*. $f\ x \le g\ x$
and [*measurable*]: $f \in$ *borel-measurable M* $g \in$ *borel-measurable M*
shows *AE x in M*. *nn-cond-exp M F f x* $\le$ *nn-cond-exp M F g x*
proof −
define h **where** $h = (\lambda x.\ g\ x - f\ x)$
have [*measurable*]: $h \in$ *borel-measurable M* **unfolding** *h-def* **by** *simp*
have $*$: *AE x in M*. $g\ x = f\ x + h\ x$ **unfolding** *h-def* **using** *assms*(*1*) **by** (*auto simp*: *ennreal-ineq-diff-add*)
have *AE x in M*. *nn-cond-exp M F g x* = *nn-cond-exp M F* $(\lambda x.\ f\ x + h\ x)\ x$
by (*rule nn-cond-exp-cong*) (*auto simp add*: $*$ *assms*)
moreover have *AE x in M*. *nn-cond-exp M F f x* + *nn-cond-exp M F h x* = *nn-cond-exp M F* $(\lambda x.\ f\ x + h\ x)\ x$
by (*rule nn-cond-exp-sum*) (*auto simp add*: *assms*)
ultimately have *AE x in M*. *nn-cond-exp M F g x* = *nn-cond-exp M F f x* + *nn-cond-exp M F h x* **by** *auto*
then show *?thesis* **by** *force*
qed

lemma *nested-subalg-is-sigma-finite*:
assumes *subalgebra M G subalgebra G F*
shows *sigma-finite-subalgebra M G*
unfolding *sigma-finite-subalgebra-def*

proof (*auto simp add*: *assms*)
have $\exists A.$ *countable* $A \wedge A \subseteq$ *sets* (*restr-to-subalg M F*) $\wedge \bigcup A =$ *space* (*restr-to-subalg M F*) $\wedge$ ($\forall a{\in}A.$ *emeasure* (*restr-to-subalg M F*) $a \neq \infty$)
using *sigma-fin-subalg sigma-finite-measure-def* **by** *auto*
then obtain *A* **where** *A*:*countable* $A \wedge A \subseteq$ *sets* (*restr-to-subalg M F*) $\wedge \bigcup A$ = *space* (*restr-to-subalg M F*) $\wedge$ ($\forall a{\in}A.$ *emeasure* (*restr-to-subalg M F*) $a \neq \infty$)
by *auto*
have *sets F* $\subseteq$ *sets M*
by (*meson assms order-trans subalgebra-def*)
then have *countable* $A \wedge A \subseteq$ *sets* (*restr-to-subalg M G*) $\wedge \bigcup A =$ *space* (*restr-to-subalg M F*) $\wedge$ ($\forall a{\in}A.$ *emeasure* (*restr-to-subalg M G*) $a \neq \infty$)
by (*metis* (*no-types*) *A assms basic-trans-rules*(*31*) *emeasure-restr-to-subalg order-trans sets-restr-to-subalg subalgebra-def*)
then show *sigma-finite-measure* (*restr-to-subalg M G*)
by (*metis sigma-finite-measure.intro space-restr-to-subalg*)
qed

lemma *nn-cond-exp-nested-subalg*:
assumes *subalgebra M G subalgebra G F*
and [*measurable*]: $f \in$ *borel-measurable M*
shows *AE x in M. nn-cond-exp M F f x = nn-cond-exp M F* (*nn-cond-exp M G f*) *x*
proof (*rule nn-cond-exp-charact*, *auto*)
interpret *G*: *sigma-finite-subalgebra M G* **by** (*rule nested-subalg-is-sigma-finite*[*OF assms*(*1*) *assms*(*2*)])
fix *A* **assume** [*measurable*]: $A \in$ *sets F*
then have [*measurable*]: $A \in$ *sets G* **using** *assms*(*2*) **by** (*meson subsetD subalgebra-def*)

have *set-nn-integral M A* (*nn-cond-exp M G f*) = $(\int^+ x.$ *indicator A x* $*$ *nn-cond-exp M G f x*$\partial M)$
by (*metis* (*no-types*) *mult.commute*)
also have ... = $(\int^+ x.$ *indicator A x* $*$ *f x* $\partial M)$ **by** (*rule G.nn-cond-exp-intg*, *auto simp add*: *assms*)
also have ... = $(\int^+ x.$ *indicator A x* $*$ *nn-cond-exp M F f x* $\partial M)$ **by** (*rule nn-cond-exp-intg*[*symmetric*], *auto simp add*: *assms*)
also have ... = *set-nn-integral M A* (*nn-cond-exp M F f*) **by** (*metis* (*no-types*) *mult.commute*)
finally show *set-nn-integral M A* (*nn-cond-exp M G f*) = *set-nn-integral M A* (*nn-cond-exp M F f*)**.**
qed

end

28.3 Real conditional expectation

Once conditional expectations of positive functions are defined, the definition for real-valued functions follows readily, by taking the difference of positive and negative parts. One could also define a conditional expecta-

tion of vector-space valued functions, as in `Bochner_Integral`, but since the real-valued case is the most important, and quicker to formalize, I concentrate on it. (It is also essential for the case of the most general Pettis integral.)

definition *real-cond-exp* :: *'a measure* ⇒ *'a measure* ⇒ (*'a* ⇒ *real*) ⇒ (*'a* ⇒ *real*)
where
real-cond-exp M F f =
(λ*x*. *enn2real*(*nn-cond-exp M F* (λ*x*. *ennreal* (*f x*)) *x*) − *enn2real*(*nn-cond-exp M F* (λ*x*. *ennreal* (−*f x*)) *x*))

lemma
shows *borel-measurable-cond-exp* [*measurable*]: *real-cond-exp M F f* ∈ *borel-measurable F*
and *borel-measurable-cond-exp2* [*measurable*]: *real-cond-exp M F f* ∈ *borel-measurable M*
unfolding *real-cond-exp-def* **by** *auto*

context *sigma-finite-subalgebra*
begin

lemma *real-cond-exp-abs*:
assumes [*measurable*]: *f* ∈ *borel-measurable M*
shows *AE x in M*. *abs*(*real-cond-exp M F f x*) ≤ *nn-cond-exp M F* (λ*x*. *ennreal* (*abs*(*f x*))) *x*
proof −
define *fp* **where** *fp* = (λ*x*. *ennreal* (*f x*))
define *fm* **where** *fm* = (λ*x*. *ennreal* (− *f x*))
have [*measurable*]: *fp* ∈ *borel-measurable M fm* ∈ *borel-measurable M* **unfolding** *fp-def fm-def* **by** *auto*
have *eq*: ⋀*x*. *ennreal* |*f x*| = *fp x* + *fm x* **unfolding** *fp-def fm-def* **by** (*simp add*: *abs-real-def ennreal-neg*)

{
fix *x* **assume** *H*: *nn-cond-exp M F fp x* + *nn-cond-exp M F fm x* = *nn-cond-exp M F* (λ*x*. *fp x* + *fm x*) *x*
have |*real-cond-exp M F f x*| ≤ |*enn2real*(*nn-cond-exp M F fp x*)| + |*enn2real*(*nn-cond-exp M F fm x*)|
unfolding *real-cond-exp-def fp-def fm-def* **by** (*auto intro*: *abs-triangle-ineq4 simp del*: *enn2real-nonneg*)
from *ennreal-leI*[*OF this*]
have *abs*(*real-cond-exp M F f x*) ≤ *nn-cond-exp M F fp x* + *nn-cond-exp M F fm x*
by *simp* (*metis add.commute ennreal-enn2real le-iff-add not-le top-unique*)
also have ... = *nn-cond-exp M F* (λ*x*. *fp x* + *fm x*) *x* **using** *H* **by** *simp*
finally have *abs*(*real-cond-exp M F f x*) ≤ *nn-cond-exp M F* (λ*x*. *fp x* + *fm x*) *x* **by** *simp*
}
moreover have *AE x in M*. *nn-cond-exp M F fp x* + *nn-cond-exp M F fm x* = *nn-cond-exp M F* (λ*x*. *fp x* + *fm x*) *x*

```
    by (rule nn-cond-exp-sum) (auto simp add: fp-def fm-def)
  ultimately have AE x in M. abs(real-cond-exp M F f x) ≤ nn-cond-exp M F
(λx. fp x + fm x) x
    by auto
  then show ?thesis using eq by simp
qed
```

The next lemma shows that the conditional expectation is an F-measurable function whose average against an F-measurable function f coincides with the average of the original function against f. It is obtained as a consequence of the same property for the positive conditional expectation, taking the difference of the positive and the negative part. The proof is given first assuming $f \geq 0$ for simplicity, and then extended to the general case in the subsequent lemma. The idea of the proof is essentially trivial, but the implementation is slightly tedious as one should check all the integrability properties of the different parts, and go back and forth between positive integral and signed integrals, and between real-valued functions and ennreal-valued functions.

Once this lemma is available, we will use it to characterize the conditional expectation, and never come back to the original technical definition, as we did in the case of the nonnegative conditional expectation.

```
lemma real-cond-exp-intg-fpos:
  assumes integrable M (λx. f x * g x) and f-pos[simp]: ⋀x. f x ≥ 0 and
          [measurable]: f ∈ borel-measurable F g ∈ borel-measurable M
  shows integrable M (λx. f x * real-cond-exp M F g x)
        (∫ x. f x * real-cond-exp M F g x ∂M) = (∫ x. f x * g x ∂M)
proof −
  have [measurable]: f ∈ borel-measurable M by (rule measurable-from-subalg[OF
subalg assms(3)])
  define gp where gp = (λx. ennreal (g x))
  define gm where gm = (λx. ennreal (− g x))
  have [measurable]: gp ∈ borel-measurable M gm ∈ borel-measurable M unfolding
gp-def gm-def by auto
  define h where h = (λx. ennreal(abs(g x)))
  have hgpgm: ⋀x. h x = gp x + gm x unfolding gp-def gm-def h-def by (simp
add: abs-real-def ennreal-neg)
  have [measurable]: h ∈ borel-measurable M unfolding h-def by simp
  have pos[simp]: ⋀x. h x ≥ 0 ⋀x. gp x ≥ 0 ⋀x. gm x ≥ 0 unfolding h-def gp-def
gm-def by simp-all
  have gp-real: ⋀x. enn2real(gp x) = max (g x) 0
    unfolding gp-def by (simp add: max-def ennreal-neg)
  have gm-real: ⋀x. enn2real(gm x) = max (−g x) 0
    unfolding gm-def by (simp add: max-def ennreal-neg)

  have (∫+ x. norm(f x * max (g x) 0) ∂M) ≤ (∫+ x. norm(f x * g x) ∂M)
    by (simp add: nn-integral-mono)
  also have ... < ∞ using assms(1) by (simp add: integrable-iff-bounded)
```

finally have $(\int^+ x.\ norm(f\ x * max\ (g\ x)\ 0)\ \partial M) < \infty$ **by** *simp*
then have *int1*: *integrable M* $(\lambda x.\ f\ x * max\ (g\ x)\ 0)$ **by** (*simp add*: *integrableI-bounded*)

have $(\int^+ x.\ norm(f\ x * max\ (-g\ x)\ 0)\ \partial M) \leq (\int^+ x.\ norm(f\ x * g\ x)\ \partial M)$
by (*simp add*: *nn-integral-mono*)
also have ... $< \infty$ **using** *assms*(*1*) **by** (*simp add*: *integrable-iff-bounded*)
finally have $(\int^+ x.\ norm(f\ x * max\ (-g\ x)\ 0)\ \partial M) < \infty$ **by** *simp*
then have *int2*: *integrable M* $(\lambda x.\ f\ x * max\ (-g\ x)\ 0)$ **by** (*simp add*: *integrableI-bounded*)

have $(\int^+ x.\ f\ x * nn\text{-}cond\text{-}exp\ M\ F\ h\ x\ \partial M) = (\int^+ x.\ f\ x * h\ x\ \partial M)$
by (*rule nn-cond-exp-intg*) *auto*
also have $\ldots = \int^+ x.\ ennreal\ (f\ x * max\ (g\ x)\ 0 + f\ x * max\ (-\ g\ x)\ 0)\ \partial M$
unfolding *h-def*
by (*intro nn-integral-cong*)(*auto simp*: *ennreal-mult*[*symmetric*] *abs-mult split*: *split-max*)
also have ... $< \infty$
using *Bochner-Integration.integrable-add*[*OF int1 int2*, *THEN integrableD*(*2*)] **by** (*auto simp add*: *less-top*)
finally have $*$: $(\int^+ x.\ f\ x * nn\text{-}cond\text{-}exp\ M\ F\ h\ x\ \partial M) < \infty$ **by** *simp*

have $(\int^+ x.\ norm(f\ x * real\text{-}cond\text{-}exp\ M\ F\ g\ x)\ \partial M) = (\int^+ x.\ f\ x * abs(real\text{-}cond\text{-}exp\ M\ F\ g\ x)\ \partial M)$
by (*simp add*: *abs-mult*)
also have ... $\leq (\int^+ x.\ f\ x * nn\text{-}cond\text{-}exp\ M\ F\ h\ x\ \partial M)$
proof (*rule nn-integral-mono-AE*)
{
fix *x* **assume** $*$: $abs(real\text{-}cond\text{-}exp\ M\ F\ g\ x) \leq nn\text{-}cond\text{-}exp\ M\ F\ h\ x$
have $ennreal\ (f\ x * |real\text{-}cond\text{-}exp\ M\ F\ g\ x|) = f\ x * ennreal(|real\text{-}cond\text{-}exp\ M\ F\ g\ x|)$
by (*simp add*: *ennreal-mult*)
also have ... $\leq f\ x * nn\text{-}cond\text{-}exp\ M\ F\ h\ x$
using $*$ **by** (*auto intro*!: *mult-left-mono*)
finally have $ennreal\ (f\ x * |real\text{-}cond\text{-}exp\ M\ F\ g\ x|) \leq f\ x * nn\text{-}cond\text{-}exp\ M\ F\ h\ x$
by *simp*
}
then show $AE\ x\ in\ M.\ ennreal\ (f\ x * |real\text{-}cond\text{-}exp\ M\ F\ g\ x|) \leq f\ x * nn\text{-}cond\text{-}exp\ M\ F\ h\ x$
using *real-cond-exp-abs*[*OF assms*(*4*)] *h-def* **by** *auto*
qed
finally have $**$: $(\int^+ x.\ norm(f\ x * real\text{-}cond\text{-}exp\ M\ F\ g\ x)\ \partial M) < \infty$ **using** $*$ **by** *auto*
show *integrable M* $(\lambda x.\ f\ x * real\text{-}cond\text{-}exp\ M\ F\ g\ x)$
using $**$ **by** (*intro integrableI-bounded*) *auto*

have $(\int^+ x.\ f\ x * nn\text{-}cond\text{-}exp\ M\ F\ gp\ x\ \partial M) \leq (\int^+ x.\ f\ x * nn\text{-}cond\text{-}exp\ M\ F\ h$

x ∂M)
proof (*rule nn-integral-mono-AE*)
have *AE x in M. nn-cond-exp M F gp x ≤ nn-cond-exp M F h x*
by (*rule nn-cond-exp-mono*) (*auto simp add*: *hgpgm*)
then show *AE x in M. f x * nn-cond-exp M F gp x ≤ f x * nn-cond-exp M F h x*
by (*auto simp*: *mult-left-mono*)
qed
then have *a*: (∫⁺*x. f x * nn-cond-exp M F gp x ∂M*) < ∞
using * **by** *auto*
have *ennreal*(*norm*(*f x * enn2real*(*nn-cond-exp M F gp x*))) ≤ *f x * nn-cond-exp M F gp x* **for** *x*
by (*auto simp add*: *ennreal-mult intro*!: *mult-left-mono*)
(*metis enn2real-ennreal enn2real-nonneg le-cases le-ennreal-iff*)
then have (∫⁺*x. norm*(*f x * enn2real*(*nn-cond-exp M F gp x*)) *∂M*) ≤ (∫⁺*x. f x * nn-cond-exp M F gp x ∂M*)
by (*simp add*: *nn-integral-mono*)
then have (∫⁺*x. norm*(*f x * enn2real*(*nn-cond-exp M F gp x*)) *∂M*) < ∞ **using** *a* **by** *auto*
then have *gp-int*: *integrable M* (λ*x. f x * enn2real*(*nn-cond-exp M F gp x*)) **by** (*simp add*: *integrableI-bounded*)
have *gp-fin*: *AE x in M. f x * nn-cond-exp M F gp x* ≠ ∞
apply (*rule nn-integral-PInf-AE*) **using** *a* **by** *auto*

have (∫ *x. f x * enn2real*(*nn-cond-exp M F gp x*) *∂M*) = *enn2real* (∫⁺ *x. f x * enn2real*(*nn-cond-exp M F gp x*) *∂M*)
by (*rule integral-eq-nn-integral*) *auto*
also have ... = *enn2real*(∫⁺ *x. ennreal*(*f x * enn2real*(*gp x*)) *∂M*)
proof −
{
fix *x* **assume** *f x * nn-cond-exp M F gp x* ≠ ∞
then have *ennreal* (*f x * enn2real* (*nn-cond-exp M F gp x*)) = *ennreal* (*f x*) * *nn-cond-exp M F gp x*
by (*auto simp add*: *ennreal-mult ennreal-mult-eq-top-iff less-top intro*!: *ennreal-mult-left-cong*)
}
then have *AE x in M. ennreal* (*f x * enn2real* (*nn-cond-exp M F gp x*)) = *ennreal* (*f x*) * *nn-cond-exp M F gp x*
using *gp-fin* **by** *auto*
then have (∫⁺ *x. f x * enn2real*(*nn-cond-exp M F gp x*) *∂M*) = (∫⁺ *x. f x * nn-cond-exp M F gp x ∂M*)
by (*rule nn-integral-cong-AE*)
also have ... = (∫⁺ *x. f x * gp x ∂M*)
by (*rule nn-cond-exp-intg*) (*auto simp add*: *gp-def*)
also have ... = (∫⁺ *x. ennreal*(*f x * enn2real*(*gp x*)) *∂M*)
by (*rule nn-integral-cong-AE*) (*auto simp*: *ennreal-mult gp-def*)
finally have (∫⁺ *x. f x * enn2real*(*nn-cond-exp M F gp x*) *∂M*) = (∫⁺ *x. ennreal*(*f x * enn2real*(*gp x*)) *∂M*) **by** *simp*
then show *?thesis* **by** *simp*

qed
also have ... = $(\int\ x.\ f\ x * ennreal(gp\ x)\ \partial M)$
by (*rule integral-eq-nn-integral*[*symmetric*]) (*auto simp add*: *gp-def*)
finally have *gp-expr*: $(\int\ x.\ f\ x * ennreal(nn\text{-}cond\text{-}exp\ M\ F\ gp\ x)\ \partial M) = (\int\ x.\ f\ x * ennreal(gp\ x)\ \partial M)$ **by** *simp*

have $(\int^+ x.\ f\ x * nn\text{-}cond\text{-}exp\ M\ F\ gm\ x\ \partial M) \leq (\int^+ x.\ f\ x * nn\text{-}cond\text{-}exp\ M\ F\ h\ x\ \partial M)$
proof (*rule nn-integral-mono-AE*)
have *AE x in M*. *nn-cond-exp M F gm x* $\leq$ *nn-cond-exp M F h x*
by (*rule nn-cond-exp-mono*) (*auto simp add*: *hgpgm*)
then show *AE x in M*. $f\ x * nn\text{-}cond\text{-}exp\ M\ F\ gm\ x \leq f\ x * nn\text{-}cond\text{-}exp\ M\ F\ h\ x$
by (*auto simp*: *mult-left-mono*)
qed
then have *a*: $(\int^+ x.\ f\ x * nn\text{-}cond\text{-}exp\ M\ F\ gm\ x\ \partial M) < \infty$
using $*$ **by** *auto*
have $\bigwedge x.\ ennreal(norm(f\ x * ennreal(nn\text{-}cond\text{-}exp\ M\ F\ gm\ x))) \leq f\ x * nn\text{-}cond\text{-}exp\ M\ F\ gm\ x$
by (*auto simp add*: *ennreal-mult intro*!: *mult-left-mono*)
(*metis enn2real-ennreal enn2real-nonneg le-cases le-ennreal-iff*)
then have $(\int^+ x.\ norm(f\ x * ennreal(nn\text{-}cond\text{-}exp\ M\ F\ gm\ x))\ \partial M) \leq (\int^+ x.\ f\ x * nn\text{-}cond\text{-}exp\ M\ F\ gm\ x\ \partial M)$
by (*simp add*: *nn-integral-mono*)
then have $(\int^+ x.\ norm(f\ x * ennreal(nn\text{-}cond\text{-}exp\ M\ F\ gm\ x))\ \partial M) < \infty$ **using** *a* **by** *auto*
then have *gm-int*: *integrable M* $(\lambda x.\ f\ x * ennreal(nn\text{-}cond\text{-}exp\ M\ F\ gm\ x))$ **by** (*simp add*: *integrableI-bounded*)
have *gm-fin*: *AE x in M*. $f\ x * nn\text{-}cond\text{-}exp\ M\ F\ gm\ x \neq \infty$
apply (*rule nn-integral-PInf-AE*) **using** *a* **by** *auto*

have $(\int\ x.\ f\ x * ennreal(nn\text{-}cond\text{-}exp\ M\ F\ gm\ x)\ \partial M) = ennreal\ (\int^+\ x.\ f\ x * ennreal(nn\text{-}cond\text{-}exp\ M\ F\ gm\ x)\ \partial M)$
by (*rule integral-eq-nn-integral*) *auto*
also have ... = $ennreal(\int^+\ x.\ ennreal(f\ x * ennreal(gm\ x))\ \partial M)$
proof −
{
fix *x* **assume** $f\ x * nn\text{-}cond\text{-}exp\ M\ F\ gm\ x \neq \infty$
then have $ennreal\ (f\ x * ennreal\ (nn\text{-}cond\text{-}exp\ M\ F\ gm\ x)) = ennreal\ (f\ x) * nn\text{-}cond\text{-}exp\ M\ F\ gm\ x$
by (*auto simp add*: *ennreal-mult ennreal-mult-eq-top-iff less-top intro*!: *ennreal-mult-left-cong*)
}
then have *AE x in M*. $ennreal\ (f\ x * ennreal\ (nn\text{-}cond\text{-}exp\ M\ F\ gm\ x)) = ennreal\ (f\ x) * nn\text{-}cond\text{-}exp\ M\ F\ gm\ x$
using *gm-fin* **by** *auto*
then have $(\int^+\ x.\ f\ x * ennreal(nn\text{-}cond\text{-}exp\ M\ F\ gm\ x)\ \partial M) = (\int^+\ x.\ f\ x * nn\text{-}cond\text{-}exp\ M\ F\ gm\ x\ \partial M)$

by (*rule nn-integral-cong-AE*)
also have ... = $(\int^+ x.\ f\ x * gm\ x\ \partial M)$
by (*rule nn-cond-exp-intg*) (*auto simp add*: *gm-def*)
also have ... = $(\int^+ x.\ ennreal(f\ x * enn2real(gm\ x))\ \partial M)$
by (*rule nn-integral-cong-AE*) (*auto simp*: *ennreal-mult gm-def*)
finally have $(\int^+ x.\ f\ x * enn2real(nn\text{-}cond\text{-}exp\ M\ F\ gm\ x)\ \partial M) = (\int^+ x.\ ennreal(f\ x * enn2real(gm\ x))\ \partial M)$ **by** *simp*
then show *?thesis* **by** *simp*
qed
also have ... = $(\int x.\ f\ x * enn2real(gm\ x)\ \partial M)$
by (*rule integral-eq-nn-integral*[*symmetric*]) (*auto simp add*: *gm-def*)
finally have *gm-expr*: $(\int x.\ f\ x * enn2real(nn\text{-}cond\text{-}exp\ M\ F\ gm\ x)\ \partial M) = (\int x.\ f\ x * enn2real(gm\ x)\ \partial M)$ **by** *simp*

have $(\int x.\ f\ x * real\text{-}cond\text{-}exp\ M\ F\ g\ x\ \partial M) = (\int x.\ f\ x * enn2real(nn\text{-}cond\text{-}exp\ M\ F\ gp\ x) - f\ x * enn2real(nn\text{-}cond\text{-}exp\ M\ F\ gm\ x)\ \partial M)$
unfolding *real-cond-exp-def gp-def gm-def* **by** (*simp add*: *right-diff-distrib*)
also have ... = $(\int x.\ f\ x * enn2real(nn\text{-}cond\text{-}exp\ M\ F\ gp\ x)\ \partial M) - (\int x.\ f\ x * enn2real(nn\text{-}cond\text{-}exp\ M\ F\ gm\ x)\ \partial M)$
by (*rule Bochner-Integration.integral-diff*) (*simp-all add*: *gp-int gm-int*)
also have ... = $(\int x.\ f\ x * enn2real(gp\ x)\ \partial M) - (\int x.\ f\ x * enn2real(gm\ x)\ \partial M)$
using *gp-expr gm-expr* **by** *simp*
also have ... = $(\int x.\ f\ x * max\ (g\ x)\ 0\ \partial M) - (\int x.\ f\ x * max\ (-g\ x)\ 0\ \partial M)$
using *gp-real gm-real* **by** *simp*
also have ... = $(\int x.\ f\ x * max\ (g\ x)\ 0 - f\ x * max\ (-g\ x)\ 0\ \partial M)$
by (*rule Bochner-Integration.integral-diff*[*symmetric*]) (*simp-all add*: *int1 int2*)
also have ... = $(\int x.\ f\ x * g\ x\ \partial M)$
by (*metis* (*mono-tags*, *opaque-lifting*) *diff-0 diff-zero eq-iff max.cobounded2 max-def minus-minus neg-le-0-iff-le right-diff-distrib*)
finally show $(\int x.\ f\ x * real\text{-}cond\text{-}exp\ M\ F\ g\ x\ \partial M) = (\int x.\ f\ x * g\ x\ \partial M)$
by *simp*
qed

lemma *real-cond-exp-intg*:
assumes *integrable M* $(\lambda x.\ f\ x * g\ x)$ **and**
[*measurable*]: *f* ∈ *borel-measurable F g* ∈ *borel-measurable M*
shows *integrable M* $(\lambda x.\ f\ x * real\text{-}cond\text{-}exp\ M\ F\ g\ x)$
$(\int x.\ f\ x * real\text{-}cond\text{-}exp\ M\ F\ g\ x\ \partial M) = (\int x.\ f\ x * g\ x\ \partial M)$
proof −
have [*measurable*]: *f* ∈ *borel-measurable M* **by** (*rule measurable-from-subalg*[*OF subalg assms*(*2*)])
define *fp* **where** $fp = (\lambda x.\ max\ (f\ x)\ 0)$
define *fm* **where** $fm = (\lambda x.\ max\ (-f\ x)\ 0)$
have [*measurable*]: *fp* ∈ *borel-measurable M fm* ∈ *borel-measurable M*
unfolding *fp-def fm-def* **by** *simp-all*
have [*measurable*]: *fp* ∈ *borel-measurable F fm* ∈ *borel-measurable F*
unfolding *fp-def fm-def* **by** *simp-all*

have $(\int^+ x.\ norm(fp\ x * g\ x)\ \partial M) \leq (\int^+ x.\ norm(f\ x * g\ x)\ \partial M)$
by (*simp add*: *fp-def nn-integral-mono*)
also have $... < \infty$ **using** *assms*(*1*) **by** (*simp add*: *integrable-iff-bounded*)
finally have $(\int^+ x.\ norm(fp\ x * g\ x)\ \partial M) < \infty$ **by** *simp*
then have *intp*: *integrable M* $(\lambda x.\ fp\ x * g\ x)$ **by** (*simp add*: *integrableI-bounded*)
moreover have $\bigwedge x.\ fp\ x \geq 0$ **unfolding** *fp-def* **by** *simp*
ultimately have *Rp*: *integrable M* $(\lambda x.\ fp\ x *$ *real-cond-exp M F g x*$)$
$(\int\ x.\ fp\ x *$ *real-cond-exp M F g x* $\partial M) = (\int\ x.\ fp\ x * g\ x\ \partial M)$
using *real-cond-exp-intg-fpos* **by** *auto*

have $(\int^+ x.\ norm(fm\ x * g\ x)\ \partial M) \leq (\int^+ x.\ norm(f\ x * g\ x)\ \partial M)$
by (*simp add*: *fm-def nn-integral-mono*)
also have $... < \infty$ **using** *assms*(*1*) **by** (*simp add*: *integrable-iff-bounded*)
finally have $(\int^+ x.\ norm(fm\ x * g\ x)\ \partial M) < \infty$ **by** *simp*
then have *intm*: *integrable M* $(\lambda x.\ fm\ x * g\ x)$ **by** (*simp add*: *integrableI-bounded*)
moreover have $\bigwedge x.\ fm\ x \geq 0$ **unfolding** *fm-def* **by** *simp*
ultimately have *Rm*: *integrable M* $(\lambda x.\ fm\ x *$ *real-cond-exp M F g x*$)$
$(\int\ x.\ fm\ x *$ *real-cond-exp M F g x* $\partial M) = (\int\ x.\ fm\ x * g\ x\ \partial M)$
using *real-cond-exp-intg-fpos* **by** *auto*

have *integrable M* $(\lambda x.\ fp\ x *$ *real-cond-exp M F g x* $-$ *fm x* $*$ *real-cond-exp M F g x*$)$
using *Rp*(*1*) *Rm*(*1*) *integrable-diff* **by** *simp*
moreover have $*$: $\bigwedge x.\ f\ x *$ *real-cond-exp M F g x* $=$ *fp x* $*$ *real-cond-exp M F g x* $-$ *fm x* $*$ *real-cond-exp M F g x*
unfolding *fp-def fm-def* **by** (*simp add*: *max-def*)
ultimately show *integrable M* $(\lambda x.\ f\ x *$ *real-cond-exp M F g x*$)$
by *simp*

have $(\int\ x.\ f\ x *$ *real-cond-exp M F g x* $\partial M) = (\int\ x.\ fp\ x *$ *real-cond-exp M F g x* $-$ *fm x* $*$ *real-cond-exp M F g x* $\partial M)$
using $*$ **by** *simp*
also have $... = (\int\ x.\ fp\ x *$ *real-cond-exp M F g x* $\partial M) - (\int\ x.\ fm\ x *$ *real-cond-exp M F g x* $\partial M)$
using *Rp*(*1*) *Rm*(*1*) **by** *simp*
also have $... = (\int\ x.\ fp\ x * g\ x\ \partial M) - (\int\ x.\ fm\ x * g\ x\ \partial M)$
using *Rp*(*2*) *Rm*(*2*) **by** *simp*
also have $... = (\int\ x.\ fp\ x * g\ x - fm\ x * g\ x\ \partial M)$
using *intm intp* **by** *simp*
also have $... = (\int\ x.\ f\ x * g\ x\ \partial M)$
unfolding *fp-def fm-def* **by** (*metis* (*no-types*, *opaque-lifting*) *diff-0 diff-zero max.commute*
max-def minus-minus mult.commute neg-le-iff-le right-diff-distrib)
finally show $(\int\ x.\ f\ x *$ *real-cond-exp M F g x* $\partial M) = (\int\ x.\ f\ x * g\ x\ \partial M)$ **by** *simp*
qed

lemma *real-cond-exp-intA*:

assumes [*measurable*]: *integrable M f A ∈ sets F*
shows $(\int x \in A.\ f\ x\ \partial M) = (\int x \in A.$ *real-cond-exp M F f x* $\partial M)$
proof −
have *A ∈ sets M* **by** (*meson assms(2) subalg subalgebra-def subsetD*)
have *integrable M* $(\lambda x.$ *indicator A x ∗ f x*) **using** *integrable-mult-indicator*[*OF* ‹*A ∈ sets M*› *assms(1)*] **by** *auto*
then show *?thesis* **using** *real-cond-exp-intg(2)*[**where** *?f = indicator A* **and** *?g = f*, *symmetric*]
unfolding *set-lebesgue-integral-def* **by** *auto*
qed

lemma *real-cond-exp-int* [*intro*]:
assumes *integrable M f*
shows *integrable M* (*real-cond-exp M F f*) $(\int x.$ *real-cond-exp M F f x* $\partial M) = (\int x.\ f\ x\ \partial M)$
using *real-cond-exp-intg*[**where** *?f* = $\lambda x.$ *1* **and** *?g = f*] *assms* **by** *auto*

lemma *real-cond-exp-charact*:
assumes $\bigwedge A.\ A \in$ *sets F* $\Longrightarrow (\int x \in A.\ f\ x\ \partial M) = (\int x \in A.\ g\ x\ \partial M)$
and [*measurable*]: *integrable M f integrable M g*
g ∈ borel-measurable F
shows *AE x in M. real-cond-exp M F f x = g x*
proof −
let *?MF = restr-to-subalg M F*
have *AE x in ?MF. real-cond-exp M F f x = g x*
proof (*rule AE-symmetric*[*OF density-unique-real*])
fix *A* **assume** *A ∈ sets ?MF*
then have [*measurable*]: *A ∈ sets F* **using** *sets-restr-to-subalg*[*OF subalg*] **by** *simp*
then have *a* [*measurable*]: *A ∈ sets M* **by** (*meson subalg subalgebra-def subsetD*)
have $(\int x \in A.\ g\ x\ \partial$ *?MF*$) = (\int x \in A.\ g\ x\ \partial M)$
unfolding *set-lebesgue-integral-def* **by** (*simp add*: *integral-subalgebra2 subalg*)
also have ... $= (\int x \in A.\ f\ x\ \partial M)$ **using** *assms(1)* **by** *simp*
also have ... $= (\int x.$ *indicator A x ∗ f x* $\partial M)$ **by** (*simp add*: *mult.commute set-lebesgue-integral-def*)
also have ... $= (\int x.$ *indicator A x ∗ real-cond-exp M F f x* $\partial M)$
apply (*rule real-cond-exp-intg(2)*[*symmetric*]) **using** *integrable-mult-indicator*[*OF a assms(2)*] **by** (*auto simp add*: *assms*)
also have ... $= (\int x \in A.$ *real-cond-exp M F f x* $\partial M)$ **by** (*simp add*: *mult.commute set-lebesgue-integral-def*)
also have ... $= (\int x \in A.$ *real-cond-exp M F f x* ∂ *?MF*$)$
by (*simp add*: *integral-subalgebra2 subalg set-lebesgue-integral-def*)
finally show $(\int x \in A.\ g\ x\ \partial$ *?MF*$) = (\int x \in A.$ *real-cond-exp M F f x* ∂ *?MF*$)$ **by** *simp*
next
have *integrable M* (*real-cond-exp M F f*) **by** (*rule real-cond-exp-int(1)*[*OF assms(2)*])
then show *integrable ?MF* (*real-cond-exp M F f*) **by** (*metis borel-measurable-cond-exp integrable-in-subalg*[*OF subalg*])

show *integrable* (*restr-to-subalg M F*) *g* **by** (*simp add*: *assms*(*3*) *integrable-in-subalg*[*OF subalg*])
qed
then show *?thesis* **using** *AE-restr-to-subalg*[*OF subalg*] **by** *auto*
qed

lemma *real-cond-exp-F-meas* [*intro*, *simp*]:
assumes *integrable M f*
f ∈ *borel-measurable F*
shows *AE x in M*. *real-cond-exp M F f x* = *f x*
by (*rule real-cond-exp-charact*, *auto simp add*: *assms measurable-from-subalg*[*OF subalg*])

lemma *real-cond-exp-mult*:
assumes [*measurable*]:*f* ∈ *borel-measurable F g* ∈ *borel-measurable M integrable M* (λ*x*. *f x* ∗ *g x*)
shows *AE x in M*. *real-cond-exp M F* (λ*x*. *f x* ∗ *g x*) *x* = *f x* ∗ *real-cond-exp M F g x*
proof (*rule real-cond-exp-charact*)
fix *A* **assume** *A* ∈ *sets F*
then have [*measurable*]: (λ*x*. *f x* ∗ *indicator A x*) ∈ *borel-measurable F* **by** *measurable*
have [*measurable*]: *A* ∈ *sets M* **using** *subalg* **by** (*meson* ‹*A* ∈ *sets F*› *subalgebra-def subsetD*)
have ∫ *x*∈*A*. (*f x* ∗ *g x*) ∂*M* = ∫ *x*. (*f x* ∗ *indicator A x*) ∗ *g x* ∂*M*
by (*simp add*: *mult.commute mult.left-commute set-lebesgue-integral-def*)
also have ... = ∫ *x*. (*f x* ∗ *indicator A x*) ∗ *real-cond-exp M F g x* ∂*M*
apply (*rule real-cond-exp-intg*(*2*)[*symmetric*], *auto simp add*: *assms*)
using *integrable-mult-indicator*[*OF* ‹*A* ∈ *sets M*› *assms*(*3*)] **by** (*simp add*: *mult.commute mult.left-commute*)
also have ... = ∫ *x*∈*A*. (*f x* ∗ *real-cond-exp M F g x*)∂*M*
by (*simp add*: *mult.commute mult.left-commute set-lebesgue-integral-def*)
finally show ∫ *x*∈*A*. (*f x* ∗ *g x*) ∂*M* = ∫ *x*∈*A*. (*f x* ∗ *real-cond-exp M F g x*)∂*M*
by *simp*
qed (*auto simp add*: *real-cond-exp-intg*(*1*) *assms*)

lemma *real-cond-exp-add* [*intro*]:
assumes [*measurable*]: *integrable M f integrable M g*
shows *AE x in M*. *real-cond-exp M F* (λ*x*. *f x* + *g x*) *x* = *real-cond-exp M F f x* + *real-cond-exp M F g x*
proof (*rule real-cond-exp-charact*)
have *integrable M* (*real-cond-exp M F f*) *integrable M* (*real-cond-exp M F g*)
using *real-cond-exp-int*(*1*) *assms* **by** *auto*
then show *integrable M* (λ*x*. *real-cond-exp M F f x* + *real-cond-exp M F g x*)
by *auto*

fix *A* **assume** [*measurable*]: *A* ∈ *sets F*
then have *A* ∈ *sets M* **by** (*meson subalg subalgebra-def subsetD*)
have *intAf*: *integrable M* (λ*x*. *indicator A x* ∗ *f x*)

using *integrable-mult-indicator*[*OF* ‹$A \in$ *sets* M› *assms*(*1*)] **by** *auto*
have *intAg*: *integrable* M (λx. *indicator* A $x * g$ x)
using *integrable-mult-indicator*[*OF* ‹$A \in$ *sets* M› *assms*(*2*)] **by** *auto*

have $\int x{\in}A$. (*real-cond-exp* M F f x + *real-cond-exp* M F g x)∂M = ($\int x{\in}A$. *real-cond-exp* M F f x ∂M) + ($\int x{\in}A$. *real-cond-exp* M F g x ∂M)
apply (*rule set-integral-add*, *auto simp add*: *assms set-integrable-def*)
using *integrable-mult-indicator*[*OF* ‹$A \in$ *sets* M› *real-cond-exp-int*(*1*)[*OF assms*(*1*)]] *integrable-mult-indicator*[*OF* ‹$A \in$ *sets* M› *real-cond-exp-int*(*1*)[*OF assms*(*2*)]] **by** *simp-all*
also have ... = ($\int x$. *indicator* A x * *real-cond-exp* M F f x ∂M) + ($\int x$. *indicator* A x * *real-cond-exp* M F g x ∂M)
unfolding *set-lebesgue-integral-def* **by** *auto*
also have ... = ($\int x$. *indicator* A $x * f$ x ∂M) + ($\int x$. *indicator* A $x * g$ x ∂M)
using *real-cond-exp-intg*(*2*) *assms* ‹$A \in$ *sets* F› *intAf intAg* **by** *auto*
also have ... = ($\int x{\in}A$. f x ∂M) + ($\int x{\in}A$. g x ∂M)
unfolding *set-lebesgue-integral-def* **by** *auto*
also have ... = $\int x{\in}A$. (f x + g x)∂M
by (*rule set-integral-add*(*2*)[*symmetric*]) (*auto simp add*: *assms set-integrable-def* ‹$A \in$ *sets* M› *intAf intAg*)
finally show $\int x{\in}A$. (f x + g x)∂M = $\int x{\in}A$. (*real-cond-exp* M F f x + *real-cond-exp* M F g x)∂M
by *simp*
qed (*auto simp add*: *assms*)

lemma *real-cond-exp-cong*:
assumes *ae*: *AE* x *in* M. f x = g x **and** [*measurable*]: $f \in$ *borel-measurable* M $g \in$ *borel-measurable* M
shows *AE* x *in* M. *real-cond-exp* M F f x = *real-cond-exp* M F g x
proof −
have *AE* x *in* M. *nn-cond-exp* M F (λx. *ennreal* (f x)) x = *nn-cond-exp* M F (λx. *ennreal* (g x)) x
apply (*rule nn-cond-exp-cong*) **using** *assms* **by** *auto*
moreover have *AE* x *in* M. *nn-cond-exp* M F (λx. *ennreal* ($-f$ x)) x = *nn-cond-exp* M F (λx. *ennreal*($-g$ x)) x
apply (*rule nn-cond-exp-cong*) **using** *assms* **by** *auto*
ultimately show *AE* x *in* M. *real-cond-exp* M F f x = *real-cond-exp* M F g x
unfolding *real-cond-exp-def* **by** *auto*
qed

lemma *real-cond-exp-cmult* [*intro*, *simp*]:
fixes c::*real*
assumes *integrable* M f
shows *AE* x *in* M. *real-cond-exp* M F (λx. $c * f$ x) x = c * *real-cond-exp* M F f x
by (*rule real-cond-exp-mult*[**where** *?f* = λx. c **and** *?g* = f], *auto simp add*: *assms borel-measurable-integrable*)

lemma *real-cond-exp-cdiv* [*intro*, *simp*]:

fixes *c*::*real*
assumes *integrable M f*
shows *AE x in M. real-cond-exp M F* (λ*x. f x / c*) *x* = *real-cond-exp M F f x / c*
using *real-cond-exp-cmult*[*of - 1 / c*, *OF assms*] **by** (*auto simp add*: *field-split-simps*)

lemma *real-cond-exp-diff* [*intro*, *simp*]:
assumes [*measurable*]: *integrable M f integrable M g*
shows *AE x in M. real-cond-exp M F* (λ*x. f x − g x*) *x* = *real-cond-exp M F f x − real-cond-exp M F g x*
proof −
have *AE x in M. real-cond-exp M F* (λ*x. f x* + (− *g x*)) *x* = *real-cond-exp M F f x* + *real-cond-exp M F* (λ*x.* −*g x*) *x*
using *real-cond-exp-add*[**where** *?f* = *f* **and** *?g* = λ*x.* − *g x*] *assms* **by** *auto*
moreover have *AE x in M. real-cond-exp M F* (λ*x.* −*g x*) *x* = − *real-cond-exp M F g x*
using *real-cond-exp-cmult*[**where** *?f* = *g* **and** *?c* = −*1*] *assms*(*2*) **by** *auto*
ultimately show *?thesis* **by** *auto*
qed

lemma *real-cond-exp-pos* [*intro*]:
assumes *AE x in M. f x* ≥ *0* **and** [*measurable*]: *f* ∈ *borel-measurable M*
shows *AE x in M. real-cond-exp M F f x* ≥ *0*
proof −
define *g* **where** *g* = (λ*x. max* (*f x*) *0*)
have *AE x in M. f x* = *g x* **using** *assms g-def* **by** *auto*
then have ∗: *AE x in M. real-cond-exp M F f x* = *real-cond-exp M F g x* **using** *real-cond-exp-cong g-def* **by** *auto*

have ⋀*x. g x* ≥ *0* **unfolding** *g-def* **by** *simp*
then have (λ*x. ennreal*(−*g x*)) = (λ*x. 0*)
by (*simp add*: *ennreal-neg*)
moreover have *AE x in M. 0* = *nn-cond-exp M F* (λ*x. 0*) *x*
by (*rule nn-cond-exp-F-meas*, *auto*)
ultimately have *AE x in M. nn-cond-exp M F* (λ*x. ennreal*(−*g x*)) *x* = *0*
by *simp*
then have *AE x in M. real-cond-exp M F g x* = *enn2real*(*nn-cond-exp M F* (λ*x. ennreal* (*g x*)) *x*)
unfolding *real-cond-exp-def* **by** *auto*
then have *AE x in M. real-cond-exp M F g x* ≥ *0* **by** *auto*
then show *?thesis* **using** ∗ **by** *auto*
qed

lemma *real-cond-exp-mono*:
assumes *AE x in M. f x* ≤ *g x* **and** [*measurable*]: *integrable M f integrable M g*
shows *AE x in M. real-cond-exp M F f x* ≤ *real-cond-exp M F g x*
proof −
have *AE x in M. real-cond-exp M F g x* − *real-cond-exp M F f x* = *real-cond-exp M F* (λ*x. g x* − *f x*) *x*

by (*rule AE-symmetric*[*OF real-cond-exp-diff*], *auto simp add*: *assms*)
moreover have *AE x in M. real-cond-exp M F* (λ*x. g x − f x*) *x* ≥ *0*
by (*rule real-cond-exp-pos*, *auto simp add*: *assms*(*1*))
ultimately have *AE x in M. real-cond-exp M F g x − real-cond-exp M F f x* ≥ *0* **by** *auto*
then show *?thesis* **by** *auto*
qed

lemma (**in** −) *measurable-P-restriction* [*measurable* (*raw*)]:
assumes [*measurable*]: *Measurable.pred M P A* ∈ *sets M*
shows {*x* ∈ *A. P x*} ∈ *sets M*
proof −
have *A* ⊆ *space M* **using** *sets.sets-into-space*[*OF assms*(*2*)]**.**
then have {*x* ∈ *A. P x*} = *A* ∩ {*x* ∈ *space M. P x*} **by** *blast*
then show *?thesis* **by** *auto*
qed

lemma *real-cond-exp-gr-c*:
assumes [*measurable*]: *integrable M f*
and *AE*: *AE x in M. f x > c*
shows *AE x in M. real-cond-exp M F f x > c*
proof −
define *X* **where** *X* = {*x* ∈ *space M. real-cond-exp M F f x* ≤ *c*}
have [*measurable*]: *X* ∈ *sets F*
unfolding *X-def* **apply** *measurable* **by** (*metis sets.top subalg subalgebra-def*)
then have [*measurable*]: *X* ∈ *sets M* **using** *sets-restr-to-subalg subalg subalgebra-def* **by** *blast*
have *emeasure M X* = *0*
proof (*rule ccontr*)
assume ¬(*emeasure M X*) = *0*
have *emeasure* (*restr-to-subalg M F*) *X* = *emeasure M X*
by (*simp add*: *emeasure-restr-to-subalg subalg*)
then have *emeasure* (*restr-to-subalg M F*) *X* > *0*
using ‹¬(*emeasure M X*) = *0*› *gr-zeroI* **by** *auto*
then obtain *A* **where** *A* ∈ *sets* (*restr-to-subalg M F*) *A* ⊆ *X emeasure* (*restr-to-subalg M F*) *A* > *0 emeasure* (*restr-to-subalg M F*) *A* < ∞
using *sigma-fin-subalg* **by** (*metis emeasure-notin-sets ennreal-0 infinity-ennreal-def le-less-linear neq-top-trans*
not-gr-zero order-refl sigma-finite-measure.approx-PInf-emeasure-with-finite)
then have [*measurable*]: *A* ∈ *sets F* **using** *subalg sets-restr-to-subalg* **by** *blast*
then have [*measurable*]: *A* ∈ *sets M* **using** *sets-restr-to-subalg subalg subalgebra-def* **by** *blast*
have *Ic*: *set-integrable M A* (λ*x. c*)
unfolding *set-integrable-def*
using ‹*emeasure* (*restr-to-subalg M F*) *A* < ∞› *emeasure-restr-to-subalg subalg* **by** *fastforce*
have *If*: *set-integrable M A f*
unfolding *set-integrable-def*
by (*rule integrable-mult-indicator*, *auto simp add*: ‹*integrable M f*›)

```
    have AE x in M. indicator A x * c = indicator A x * f x
    proof (rule integral-ineq-eq-0-then-AE)
      have (∫ x∈A. c ∂M) = (∫ x∈A. f x ∂M)
      proof (rule antisym)
        show (∫ x∈A. c ∂M) ≤ (∫ x∈A. f x ∂M)
          apply (rule set-integral-mono-AE) using Ic If assms(2) by auto
        have (∫ x∈A. f x ∂M) = (∫ x∈A. real-cond-exp M F f x ∂M)
          by (rule real-cond-exp-intA, auto simp add: ‹integrable M f›)
        also have ... ≤ (∫ x∈A. c ∂M)
          apply (rule set-integral-mono)
          unfolding set-integrable-def
         apply (rule integrable-mult-indicator, simp, simp add: real-cond-exp-int(1)[OF
‹integrable M f›])
          using Ic X-def ‹A ⊆ X› by (auto simp: set-integrable-def)
        finally show (∫ x∈A. f x ∂M) ≤ (∫ x∈A. c ∂M) by simp
      qed
      then have measure M A * c = LINT x|M. indicat-real A x * f x
        by (auto simp: set-lebesgue-integral-def)
      then show LINT x|M. indicat-real A x * c = LINT x|M. indicat-real A x *
f x
        by auto
      show AE x in M. indicat-real A x * c ≤ indicat-real A x * f x
      using AE unfolding indicator-def by auto
    qed (use Ic If in ‹auto simp: set-integrable-def›)
    then have AE x∈A in M. c = f x by auto
    then have AE x∈A in M. False using assms(2) by auto
    have A ∈ null-sets M unfolding ae-filter-def by (meson AE-iff-null-sets ‹A
∈ sets M› ‹AE x∈A in M. False›)
    then show False using ‹emeasure (restr-to-subalg M F) A > 0›
      by (simp add: emeasure-restr-to-subalg null-setsD1 subalg)
  qed
  then show ?thesis using AE-iff-null-sets[OF ‹X ∈ sets M›] unfolding X-def
by auto
qed

lemma real-cond-exp-less-c:
  assumes [measurable]: integrable M f
      and AE x in M. f x < c
  shows AE x in M. real-cond-exp M F f x < c
proof −
  have AE x in M. real-cond-exp M F f x = −real-cond-exp M F (λx. −f x) x
    using real-cond-exp-cmult[OF ‹integrable M f›, of −1] by auto
  moreover have AE x in M. real-cond-exp M F (λx. −f x) x > −c
    apply (rule real-cond-exp-gr-c) using assms by auto
  ultimately show ?thesis by auto
qed

lemma real-cond-exp-ge-c:
  assumes [measurable]: integrable M f
```

and *AE x in M. f x ≥ c*
shows *AE x in M. real-cond-exp M F f x ≥ c*
proof −
obtain *u*::*nat* ⇒ *real* **where** *u*: ⋀*n*. *u n* < *c u* ⟶ *c*
using *approx-from-below-dense-linorder*[*of c*−*1 c*] **by** *auto*
have ∗: *AE x in M. real-cond-exp M F f x* > *u n* **for** *n*::*nat*
apply (*rule real-cond-exp-gr-c*) **using** *assms* ‹*u n* < *c*› **by** *auto*
have *AE x in M.* ∀ *n. real-cond-exp M F f x* > *u n*
by (*subst AE-all-countable*, *auto simp add*: ∗)
moreover have *real-cond-exp M F f x ≥ c* **if** ∀ *n. real-cond-exp M F f x* > *u n* **for** *x*
proof −
have *real-cond-exp M F f x ≥ u n* **for** *n* **using** *that less-imp-le* **by** *auto*
then show *?thesis* **using** *u*(*2*) *LIMSEQ-le-const2* **by** *metis*
qed
ultimately show *?thesis* **by** *auto*
qed

lemma *real-cond-exp-le-c*:
assumes [*measurable*]: *integrable M f*
and *AE x in M. f x ≤ c*
shows *AE x in M. real-cond-exp M F f x ≤ c*
proof −
have *AE x in M. real-cond-exp M F f x* = −*real-cond-exp M F* (λ*x*. −*f x*) *x*
using *real-cond-exp-cmult*[*OF* ‹*integrable M f*›, *of* −*1*] **by** *auto*
moreover have *AE x in M. real-cond-exp M F* (λ*x*. −*f x*) *x* ≥ −*c*
apply (*rule real-cond-exp-ge-c*) **using** *assms* **by** *auto*
ultimately show *?thesis* **by** *auto*
qed

lemma *real-cond-exp-mono-strict*:
assumes *AE x in M. f x* < *g x* **and** [*measurable*]: *integrable M f integrable M g*
shows *AE x in M. real-cond-exp M F f x* < *real-cond-exp M F g x*
proof −
have *AE x in M. real-cond-exp M F g x* − *real-cond-exp M F f x* = *real-cond-exp M F* (λ*x*. *g x* − *f x*) *x*
by (*rule AE-symmetric*[*OF real-cond-exp-diff*], *auto simp add*: *assms*)
moreover have *AE x in M. real-cond-exp M F* (λ*x*. *g x* − *f x*) *x* > *0*
by (*rule real-cond-exp-gr-c*, *auto simp add*: *assms*)
ultimately have *AE x in M. real-cond-exp M F g x* − *real-cond-exp M F f x* > *0* **by** *auto*
then show *?thesis* **by** *auto*
qed

lemma *real-cond-exp-nested-subalg* [*intro*, *simp*]:
assumes *subalgebra M G subalgebra G F*
and [*measurable*]: *integrable M f*
shows *AE x in M. real-cond-exp M F* (*real-cond-exp M G f*) *x* = *real-cond-exp M F f x*

proof (*rule real-cond-exp-charact*)
interpret *G*: *sigma-finite-subalgebra M G* **by** (*rule nested-subalg-is-sigma-finite*[*OF assms*(*1*) *assms*(*2*)])
show *integrable M* (*real-cond-exp M G f*) **by** (*auto simp add*: *assms G.real-cond-exp-int*(*1*))

fix *A* **assume** [*measurable*]: *A* ∈ *sets F*
then have [*measurable*]: *A* ∈ *sets G* **using** *assms*(*2*) **by** (*meson subsetD subalgebra-def*)
have *set-lebesgue-integral M A* (*real-cond-exp M G f*) = *set-lebesgue-integral M A f*
by (*rule G.real-cond-exp-intA*[*symmetric*], *auto simp add*: *assms*(*3*))
also have ... = *set-lebesgue-integral M A* (*real-cond-exp M F f*)
by (*rule real-cond-exp-intA*, *auto simp add*: *assms*(*3*))
finally show *set-lebesgue-integral M A* (*real-cond-exp M G f*) = *set-lebesgue-integral M A* (*real-cond-exp M F f*) **by** *auto*
qed (*auto simp add*: *assms real-cond-exp-int*(*1*))

lemma *real-cond-exp-sum* [*intro*, *simp*]:
fixes *f*::*'b* ⇒ *'a* ⇒ *real*
assumes [*measurable*]: ⋀*i*. *integrable M* (*f i*)
shows *AE x in M*. *real-cond-exp M F* (λ*x*. ∑ *i*∈*I*. *f i x*) *x* = (∑ *i*∈*I*. *real-cond-exp M F* (*f i*) *x*)
proof (*rule real-cond-exp-charact*)
fix *A* **assume** [*measurable*]: *A* ∈ *sets F*
then have *A-meas* [*measurable*]: *A* ∈ *sets M* **by** (*meson subsetD subalg subalgebra-def*)
have ∗: *integrable M* (λ*x*. *indicator A x* ∗ *f i x*) **for** *i*
using *integrable-mult-indicator*[*OF* ‹*A* ∈ *sets M*› *assms*(*1*)] **by** *auto*
have ∗∗: *integrable M* (λ*x*. *indicator A x* ∗ *real-cond-exp M F* (*f i*) *x*) **for** *i*
using *integrable-mult-indicator*[*OF* ‹*A* ∈ *sets M*› *real-cond-exp-int*(*1*)[*OF assms*(*1*)]]
by *auto*
have *inti*: (∫ *x*. *indicator A x* ∗ *f i x* ∂*M*) = (∫ *x*. *indicator A x* ∗ *real-cond-exp M F* (*f i*) *x* ∂*M*) **for** *i*
by (*rule real-cond-exp-intg*(*2*)[*symmetric*], *auto simp add*: ∗)

have (∫ *x*∈*A*. (∑ *i*∈*I*. *f i x*)∂*M*) = (∫ *x*. (∑ *i*∈*I*. *indicator A x* ∗ *f i x*)∂*M*)
by (*simp add*: *sum-distrib-left set-lebesgue-integral-def*)
also have ... = (∑ *i*∈*I*. (∫ *x*. *indicator A x* ∗ *f i x* ∂*M*))
by (*rule Bochner-Integration.integral-sum*, *simp add*: ∗)
also have ... = (∑ *i*∈*I*. (∫ *x*. *indicator A x* ∗ *real-cond-exp M F* (*f i*) *x* ∂*M*))
using *inti* **by** *auto*
also have ... = (∫ *x*. (∑ *i*∈*I*. *indicator A x* ∗ *real-cond-exp M F* (*f i*) *x*)∂*M*)
by (*rule Bochner-Integration.integral-sum*[*symmetric*], *simp add*: ∗∗)
also have ... = (∫ *x*∈*A*. (∑ *i*∈*I*. *real-cond-exp M F* (*f i*) *x*)∂*M*)
by (*simp add*: *sum-distrib-left set-lebesgue-integral-def*)
finally show (∫ *x*∈*A*. (∑ *i*∈*I*. *f i x*)∂*M*) = (∫ *x*∈*A*. (∑ *i*∈*I*. *real-cond-exp M F* (*f i*) *x*)∂*M*) **by** *auto*
qed (*auto simp add*: *assms real-cond-exp-int*(*1*)[*OF assms*(*1*)])

Jensen's inequality, describing the behavior of the integral under a convex

function, admits a version for the conditional expectation, as follows.

theorem *real-cond-exp-jensens-inequality*:
fixes $q :: real \Rightarrow real$
assumes X: *integrable M X AE x in M. X x ∈ I*
assumes I: $I = \{a <..< b\} \vee I = \{a <..\} \vee I = \{..< b\} \vee I = UNIV$
assumes q: *integrable M (λx. q (X x)) convex-on I q q ∈ borel-measurable borel*
shows *AE x in M. real-cond-exp M F X x ∈ I*
AE x in M. q (real-cond-exp M F X x) ≤ real-cond-exp M F (λx. q (X x)) x
proof −
have *open I* **using** I **by** *auto*
then have *interior I = I* **by** (*simp add*: *interior-eq*)
have [*measurable*]: *I ∈ sets borel* **using** I **by** *auto*
define *phi* **where** *phi = (λx. Inf ((λt. (q x − q t) / (x − t)) ' ({x<..} ∩ I)))*
have ∗∗: *q (X x) ≥ q (real-cond-exp M F X x) + phi (real-cond-exp M F X x) ∗ (X x − real-cond-exp M F X x)*
if *X x ∈ I real-cond-exp M F X x ∈ I* **for** *x*
unfolding *phi-def* **apply** (*rule convex-le-Inf-differential*)
using ‹*convex-on I q*› *that* ‹*interior I = I*› **by** *auto*

It is not clear that the function ϕ is measurable. We replace it by a version which is better behaved.

define *psi* **where** *psi = (λx. phi x ∗ indicator I x)*
have A: *psi y = phi y* **if** *y ∈ I* **for** *y* **unfolding** *psi-def indicator-def* **using** *that* **by** *auto*
have ∗: *q (X x) ≥ q (real-cond-exp M F X x) + psi (real-cond-exp M F X x) ∗ (X x − real-cond-exp M F X x)*
if *X x ∈ I real-cond-exp M F X x ∈ I* **for** *x*
unfolding *A*[*OF* ‹*real-cond-exp M F X x ∈ I*›] **using** ∗∗ *that* **by** *auto*

note I
moreover have *AE x in M. real-cond-exp M F X x > a* **if** *I ⊆ {a <..}* **for** *a*
apply (*rule real-cond-exp-gr-c*) **using** X *that* **by** *auto*
moreover have *AE x in M. real-cond-exp M F X x < b* **if** *I ⊆ {..<b}* **for** *b*
apply (*rule real-cond-exp-less-c*) **using** X *that* **by** *auto*
ultimately show *AE x in M. real-cond-exp M F X x ∈ I*
by (*elim disjE*) (*auto simp*: *subset-eq*)
then have *main-ineq*: *AE x in M. q (X x) ≥ q (real-cond-exp M F X x) + psi (real-cond-exp M F X x) ∗ (X x − real-cond-exp M F X x)*
using ∗ *X*(*2*) **by** *auto*

Then, one wants to take the conditional expectation of this inequality. On the left, one gets the conditional expectation of $q \circ X$. On the right, the last term vanishes, and one is left with q of the conditional expectation, as desired. Unfortunately, this argument only works if $\psi \cdot X$ and $q(E(X|F))$ are integrable, and there is no reason why this should be true. The trick is to multiply by a F-measurable function which is small enough to make everything integrable.

obtain *f*::*'a ⇒ real* **where** [*measurable*]: *f ∈ borel-measurable* (*restr-to-subalg M*

F)
integrable (*restr-to-subalg M F*) *f*
and *f*: ⋀*x*. *f x* > *0* ⋀*x*. *f x* ≤ *1*
using *sigma-finite-measure.obtain-positive-integrable-function*[*OF sigma-fin-subalg*]
by *metis*
then have [*measurable*]: *f* ∈ *borel-measurable F* **by** (*simp add*: *subalg*)
then have [*measurable*]: *f* ∈ *borel-measurable M* **using** *measurable-from-subalg*[*OF subalg*] **by** *blast*
define *g* **where** *g* = (λ*x*. *f x*/(*1*+ |*psi* (*real-cond-exp M F X x*)| + |*q* (*real-cond-exp M F X x*)|))
define *G* **where** *G* = (λ*x*. *g x* ∗ *psi* (*real-cond-exp M F X x*))
have *g*: *g x* > *0* *g x* ≤ *1* **for** *x* **unfolding** *G-def g-def* **using** *f*[*of x*] **by** (*auto simp add*: *abs-mult*)
have *G*: |*G x*| ≤ *1* **for** *x* **unfolding** *G-def g-def* **using** *f*[*of x*]
proof (*auto simp add*: *abs-mult*)
have *f x* ∗ |*psi* (*real-cond-exp M F X x*)| ≤ *1* ∗ |*psi* (*real-cond-exp M F X x*)|
apply (*rule mult-mono*) **using** *f*[*of x*] **by** *auto*
also have ... ≤ *1* + |*psi* (*real-cond-exp M F X x*)| + |*q* (*real-cond-exp M F X x*)| **by** *auto*
finally show *f x* ∗ |*psi* (*real-cond-exp M F X x*)| ≤ *1* + |*psi* (*real-cond-exp M F X x*)| + |*q* (*real-cond-exp M F X x*)|
by *simp*
qed
have *AE x in M*. *g x* ∗ *q* (*X x*) ≥ *g x* ∗ (*q* (*real-cond-exp M F X x*) + *psi* (*real-cond-exp M F X x*) ∗ (*X x* − *real-cond-exp M F X x*))
using *main-ineq g* **by** (*auto simp add*: *divide-simps*)
then have *main-G*: *AE x in M*. *g x* ∗ *q* (*X x*) ≥ *g x* ∗ *q* (*real-cond-exp M F X x*) + *G x* ∗ (*X x* − *real-cond-exp M F X x*)
unfolding *G-def* **by** (*auto simp add*: *algebra-simps*)

To proceed, we need to know that ψ is measurable.

have *phi-mono*: *phi x* ≤ *phi y* **if** *x* ≤ *y x* ∈ *I y* ∈ *I* **for** *x y*
proof (*cases x* < *y*)
case *True*
have *q x* + *phi x* ∗ (*y*−*x*) ≤ *q y*
unfolding *phi-def* **apply** (*rule convex-le-Inf-differential*) **using** ‹*convex-on I q*› *that* ‹*interior I* = *I*› **by** *auto*
then have *phi x* ≤ (*q x* − *q y*) / (*x* − *y*)
using *that* ‹*x* < *y*› **by** (*auto simp add*: *field-split-simps algebra-simps*)
moreover have (*q x* − *q y*)/(*x* − *y*) ≤ *phi y*
unfolding *phi-def* **proof** (*rule cInf-greatest*, *auto*)
fix *t* **assume** *t* ∈ *I y* < *t*
have (*q x* − *q y*) / (*x* − *y*) ≤ (*q x* − *q t*) / (*x* − *t*)
apply (*rule convex-on-diff*[*OF q*(*2*)]) **using** ‹*y* < *t*› ‹*x* < *y*› ‹*t* ∈ *I*› ‹*x* ∈ *I*› **by** *auto*
also have ... ≤ (*q y* − *q t*) / (*y* − *t*)
apply (*rule convex-on-diff*[*OF q*(*2*)]) **using** ‹*y* < *t*› ‹*x* < *y*› ‹*t* ∈ *I*› ‹*x* ∈ *I*› **by** *auto*
finally show (*q x* − *q y*) / (*x* − *y*) ≤ (*q y* − *q t*) / (*y* − *t*) **by** *simp*

next
obtain *e* **where** *0 < e ball y e ⊆ I* **using** ‹*open I*› ‹*y ∈ I*› *openE* **by** *blast*
then have *y + e/2 ∈ {y<..} ∩ I* **by** (*auto simp*: *dist-real-def*)
then show *{y<..} ∩ I = {} ⟹ False* **by** *auto*
qed
ultimately show *phi x ≤ phi y* **by** *auto*
next
case *False*
then have *x = y* **using** ‹*x ≤ y*› **by** *auto*
then show *?thesis* **by** *auto*
qed
have [*measurable*]: *psi ∈ borel-measurable borel*
by (*rule borel-measurable-piecewise-mono*[*of* {*I*, −*I*}])
(*auto simp add*: *psi-def indicator-def phi-mono intro*: *mono-onI*)
have [*measurable*]: *q ∈ borel-measurable borel* **using** *q* **by** *simp*

have [*measurable*]: *X ∈ borel-measurable M*
real-cond-exp M F X ∈ borel-measurable F
g ∈ borel-measurable F g ∈ borel-measurable M
G ∈ borel-measurable F G ∈ borel-measurable M
using *X measurable-from-subalg*[*OF subalg*] **unfolding** *G-def g-def* **by** *auto*
have *int1*: *integrable* (*restr-to-subalg M F*) (*λx. g x ∗ q* (*real-cond-exp M F X x*))
apply (*rule Bochner-Integration.integrable-bound*[*of - f*], *auto simp add*: *subalg* ‹*integrable* (*restr-to-subalg M F*) *f*›)
unfolding *g-def* **by** (*auto simp add*: *field-split-simps abs-mult algebra-simps*)
have *int2*: *integrable M* (*λx. G x ∗* (*X x − real-cond-exp M F X x*))
apply (*rule Bochner-Integration.integrable-bound*[*of - λx. |X x| + |real-cond-exp M F X x|*])
apply (*auto intro*!: *Bochner-Integration.integrable-add integrable-abs real-cond-exp-int* ‹*integrable M X*› *AE-I2*)
using *G* **unfolding** *abs-mult* **by** (*meson abs-ge-zero abs-triangle-ineq4 dual-order.trans mult-left-le-one-le*)
have *int3*: *integrable M* (*λx. g x ∗ q* (*X x*))
apply (*rule Bochner-Integration.integrable-bound*[*of - λx. q*(*X x*)], *auto simp add*: *q*(*1*) *abs-mult*)
using *g* **by** (*simp add*: *less-imp-le mult-left-le-one-le*)

Taking the conditional expectation of the main convexity inequality `main_G`, we get the following.

have *AE x in M. real-cond-exp M F* (*λx. g x ∗ q* (*X x*)) *x ≥ real-cond-exp M F* (*λx. g x ∗ q* (*real-cond-exp M F X x*) + *G x ∗* (*X x − real-cond-exp M F X x*)) *x*
apply (*rule real-cond-exp-mono*[*OF main-G*])
apply (*rule Bochner-Integration.integrable-add*[*OF integrable-from-subalg*[*OF subalg int1*]])
using *int2 int3* **by** *auto*

This reduces to the desired inequality thanks to the properties of conditional expectation, i.e., the conditional expectation of an F-measurable function is this function, and one can multiply an F-measurable function outside

of conditional expectations. Since all these equalities only hold almost everywhere, we formulate them separately, and then combine all of them to simplify the above equation, again almost everywhere.

moreover have *AE x in M. real-cond-exp M F* (λ*x. g x * q* (*X x*)) *x = g x * real-cond-exp M F* (λ*x. q* (*X x*)) *x*
by (*rule real-cond-exp-mult*, *auto simp add*: *int3*)
moreover have *AE x in M. real-cond-exp M F* (λ*x. g x * q* (*real-cond-exp M F X x*) + *G x ** (*X x − real-cond-exp M F X x*)) *x*
= real-cond-exp M F (λ*x. g x * q* (*real-cond-exp M F X x*)) *x* + *real-cond-exp M F* (λ*x. G x ** (*X x − real-cond-exp M F X x*)) *x*
by (*rule real-cond-exp-add*, *auto simp add*: *integrable-from-subalg*[*OF subalg int1*] *int2*)
moreover have *AE x in M. real-cond-exp M F* (λ*x. g x * q* (*real-cond-exp M F X x*)) *x = g x * q* (*real-cond-exp M F X x*)
by (*rule real-cond-exp-F-meas*, *auto simp add*: *integrable-from-subalg*[*OF subalg int1*])
moreover have *AE x in M. real-cond-exp M F* (λ*x. G x ** (*X x − real-cond-exp M F X x*)) *x = G x * real-cond-exp M F* (λ*x.* (*X x − real-cond-exp M F X x*)) *x*
by (*rule real-cond-exp-mult*, *auto simp add*: *int2*)
moreover have *AE x in M. real-cond-exp M F* (λ*x.* (*X x − real-cond-exp M F X x*)) *x = real-cond-exp M F X x − real-cond-exp M F* (λ*x. real-cond-exp M F X x*) *x*
by (*rule real-cond-exp-diff*, *auto intro*!: *real-cond-exp-int* ‹*integrable M X*›)
moreover have *AE x in M. real-cond-exp M F* (λ*x. real-cond-exp M F X x*) *x = real-cond-exp M F X x*
by (*rule real-cond-exp-F-meas*, *auto intro*!: *real-cond-exp-int* ‹*integrable M X*›)
ultimately have *AE x in M. g x * real-cond-exp M F* (λ*x. q* (*X x*)) *x ≥ g x * q* (*real-cond-exp M F X x*)
by *auto*
then show *AE x in M. real-cond-exp M F* (λ*x. q* (*X x*)) *x ≥ q* (*real-cond-exp M F X x*)
using *g*(*1*) **by** (*auto simp add*: *field-split-simps*)
qed

Jensen's inequality does not imply that $q(E(X|F))$ is integrable, as it only proves an upper bound for it. Indeed, this is not true in general, as the following counterexample shows:

on $[1, \infty)$ with Lebesgue measure, let F be the sigma-algebra generated by the intervals $[n, n+1)$ for integer n. Let $q(x) = -\sqrt{x}$ for $x \geq 0$. Define X which is equal to $1/n$ over $[n, n+1/n)$ and 2^{-n} on $[n+1/n, n+1)$. Then X is integrable as $\sum 1/n^2 < \infty$, and $q(X)$ is integrable as $\sum 1/n^{3/2} < \infty$. On the other hand, $E(X|F)$ is essentially equal to $1/n^2$ on $[n, n+1)$ (we neglect the term 2^{-n}, we only put it there because X should take its values in $I = (0, \infty)$). Hence, $q(E(X|F))$ is equal to $-1/n$ on $[n, n+1)$, hence it is not integrable.

However, this counterexample is essentially the only situation where this function is not integrable, as shown by the next lemma.

lemma *integrable-convex-cond-exp*:
fixes $q :: real \Rightarrow real$
assumes X: *integrable M X AE x in M. X x ∈ I*
assumes I: $I = \{a <..< b\} \vee I = \{a <..\} \vee I = \{..< b\} \vee I = UNIV$
assumes q: *integrable M* $(\lambda x.\ q\ (X\ x))$ *convex-on I q q ∈ borel-measurable borel*
assumes H: *emeasure M* $(space\ M) = \infty \Longrightarrow 0 \in I$
shows *integrable M* $(\lambda x.\ q\ (real\text{-}cond\text{-}exp\ M\ F\ X\ x))$
proof −
have [*measurable*]: $(\lambda x.\ q\ (real\text{-}cond\text{-}exp\ M\ F\ X\ x)) \in borel\text{-}measurable\ M$
$q \in borel\text{-}measurable\ borel$
$X \in borel\text{-}measurable\ M$
using *X*(*1*) *q*(*3*) **by** *auto*
have *open I* **using** *I* **by** *auto*
then have *interior I = I* **by** (*simp add*: *interior-eq*)

consider *emeasure M* $(space\ M) = 0 \mid emeasure\ M\ (space\ M) > 0 \wedge emeasure\ M\ (space\ M) < \infty \mid emeasure\ M\ (space\ M) = \infty$
by (*metis infinity-ennreal-def not-gr-zero top.not-eq-extremum*)
then show *?thesis*
proof (*cases*)
case *1*
show *?thesis* **by** (*subst integrable-cong-AE*[*of - - λx. 0*], *auto intro*: *emeasure-0-AE*[*OF 1*])
next
case *2*
interpret *finite-measure M* **using** *2* **by** (*auto intro*!: *finite-measureI*)

have $I \neq \{\}$
using ‹*AE x in M. X x ∈ I*› *2 eventually-mono integral-less-AE-space* **by** *fastforce*
then obtain *z* **where** $z \in I$ **by** *auto*

define *A* **where** $A = Inf\ ((\lambda t.\ (q\ z - q\ t)\ /\ (z - t))\ `\ (\{z<..\} \cap I))$
have $q\ y \geq q\ z + A * (y - z)$ **if** $y \in I$ **for** *y* **unfolding** *A-def* **apply** (*rule convex-le-Inf-differential*)
using ‹*z ∈ I*› ‹*y ∈ I*› ‹*interior I = I*› *q*(*2*) **by** *auto*
then have *AE x in M.* $q\ (real\text{-}cond\text{-}exp\ M\ F\ X\ x) \geq q\ z + A * (real\text{-}cond\text{-}exp\ M\ F\ X\ x - z)$
using *real-cond-exp-jensens-inequality*(*1*)[*OF X I q*] **by** *auto*
moreover have *AE x in M.* $q\ (real\text{-}cond\text{-}exp\ M\ F\ X\ x) \leq real\text{-}cond\text{-}exp\ M\ F\ (\lambda x.\ q\ (X\ x))\ x$
using *real-cond-exp-jensens-inequality*(*2*)[*OF X I q*] **by** *auto*
moreover have $|a| \leq |b| + |c|$ **if** $b \leq a \wedge a \leq c$ **for** *a b c*::*real*
using *that* **by** *auto*
ultimately have ∗: *AE x in M.* $|q\ (real\text{-}cond\text{-}exp\ M\ F\ X\ x)| \leq |real\text{-}cond\text{-}exp\ M\ F\ (\lambda x.\ q\ (X\ x))\ x| + |q\ z + A * (real\text{-}cond\text{-}exp\ M\ F\ X\ x - z)|$
by *auto*

show *integrable M* (λ*x*. *q* (*real-cond-exp M F X x*))
apply (*rule Bochner-Integration.integrable-bound*[*of* - λ*x*. |*real-cond-exp M F* (λ*x*. *q* (*X x*)) *x*| + |*q z* + *A* ∗ (*real-cond-exp M F X x* − *z*)|])
apply (*auto intro*!: *Bochner-Integration.integrable-add integrable-abs integrable-mult-right Bochner-Integration.integrable-diff real-cond-exp-int*(*1*))
using *X*(*1*) *q*(*1*) ∗ **by** *auto*
next
case *3*
then have *0* ∈ *I* **using** *H finite-measure.finite-emeasure-space* **by** *auto*
have *q*(*0*) = *0*
proof (*rule ccontr*)
assume ∗: ¬(*q*(*0*) = *0*)
define *e* **where** *e* = |*q*(*0*)| / *2*
then have *e* > *0* **using** ∗ **by** *auto*
have *continuous* (*at 0*) *q*
using *q*(*2*) ‹*0* ∈ *I*› ‹*open I*› ‹*interior I* = *I*› *continuous-on-interior convex-on-continuous* **by** *blast*
then obtain *d* **where** *d*: *d* > *0* ⋀*y*. |*y* − *0*| < *d* ⟹ |*q y* − *q 0*| < *e* **using** ‹*e* > *0*›
by (*metis continuous-at-real-range real-norm-def*)
then have ∗: |*q*(*y*)| > *e* **if** |*y*| < *d* **for** *y*
proof −
have |*q 0*| ≤ |*q 0* − *q y*| + |*q y*| **by** *auto*
also have ... < *e* + |*q y*| **using** *d*(*2*) *that* **by** *force*
finally have |*q y*| > |*q 0*| − *e* **by** *auto*
then show *?thesis* **unfolding** *e-def* **by** *simp*
qed
have *emeasure M* {*x* ∈ *space M*. |*X x*| < *d*} ≤ *emeasure M* ({*x* ∈ *space M*. *1* ≤ *ennreal*(*1*/*e*) ∗ |*q*(*X x*)|})
by (*rule emeasure-mono*, *auto simp add*: ∗ ‹*e*>*0*› *less-imp-le ennreal-mult*′′[*symmetric*])
also have ... ≤ (*1*/*e*) ∗ (∫ +*x*. *ennreal*(|*q*(*X x*)|) ∗ *indicator* (*space M*) *x* ∂*M*)
by (*rule nn-integral-Markov-inequality*, *auto*)
also have ... = (*1*/*e*) ∗ (∫ +*x*. *ennreal*(|*q*(*X x*)|) ∂*M*) **by** *auto*
also have ... = (*1*/*e*) ∗ *ennreal*(∫ *x*. |*q*(*X x*)| ∂*M*)
using *nn-integral-eq-integral*[*OF integrable-abs*[*OF q*(*1*)]] **by** *auto*
also have ... < ∞
by (*simp add*: *ennreal-mult-less-top*)
finally have *A*: *emeasure M* {*x* ∈ *space M*. |*X x*| < *d*} < ∞ **by** *simp*

have {*x* ∈ *space M*. |*X x*| ≥ *d*} = {*x* ∈ *space M*. *1* ≤ *ennreal*(*1*/*d*) ∗ |*X x*|} ∩ *space M*
by (*auto simp add*: ‹*d*>*0*› *ennreal-mult*′′[*symmetric*])
then have *emeasure M* {*x* ∈ *space M*. |*X x*| ≥ *d*} = *emeasure M* ({*x* ∈ *space M*. *1* ≤ *ennreal*(*1*/*d*) ∗ |*X x*|})
by *auto*
also have ... ≤ (*1*/*d*) ∗ (∫ +*x*. *ennreal*(|*X x*|) ∗ *indicator* (*space M*) *x* ∂*M*)
by (*rule nn-integral-Markov-inequality*, *auto*)
also have ... = (*1*/*d*) ∗ (∫ +*x*. *ennreal*(|*X x*|) ∂*M*) **by** *auto*
also have ... = (*1*/*d*) ∗ *ennreal*(∫ *x*. |*X x*| ∂*M*)

using *nn-integral-eq-integral*[*OF integrable-abs*[*OF X*(*1*)]] **by** *auto*
also have ... < ∞
by (*simp add*: *ennreal-mult-less-top*)
finally have *B*: *emeasure M* {*x* ∈ *space M*. |*X x*| ≥ *d*} < ∞ **by** *simp*

have *space M* = {*x* ∈ *space M*. |*X x*| < *d*} ∪ {*x* ∈ *space M*. |*X x*| ≥ *d*} **by** *auto*
then have *emeasure M* (*space M*) = *emeasure M* ({*x* ∈ *space M*. |*X x*| < *d*} ∪ {*x* ∈ *space M*. |*X x*| ≥ *d*})
by *simp*
also have ... ≤ *emeasure M* {*x* ∈ *space M*. |*X x*| < *d*} + *emeasure M* {*x* ∈ *space M*. |*X x*| ≥ *d*}
by (*auto intro*!: *emeasure-subadditive*)
also have ... < ∞ **using** *A B* **by** *auto*
finally show *False* **using** ‹*emeasure M* (*space M*) = ∞› **by** *auto*
qed

define *A* **where** *A* = *Inf* ((λ*t*. (*q 0* − *q t*) / (*0* − *t*)) ' ({*0*<..} ∩ *I*))
have *q y* ≥ *q 0* + *A* ∗ (*y* − *0*) **if** *y* ∈ *I* **for** *y* **unfolding** *A-def* **apply** (*rule convex-le-Inf-differential*)
using ‹*0* ∈ *I*› ‹*y* ∈ *I*› ‹*interior I* = *I*› *q*(*2*) **by** *auto*
then have *q y* ≥ *A* ∗ *y* **if** *y* ∈ *I* **for** *y* **using** ‹*q 0* = *0*› *that* **by** *auto*
then have *AE x in M*. *q* (*real-cond-exp M F X x*) ≥ *A* ∗ *real-cond-exp M F X x*
using *real-cond-exp-jensens-inequality*(*1*)[*OF X I q*] **by** *auto*
moreover have *AE x in M*. *q* (*real-cond-exp M F X x*) ≤ *real-cond-exp M F* (λ*x*. *q* (*X x*)) *x*
using *real-cond-exp-jensens-inequality*(*2*)[*OF X I q*] **by** *auto*
moreover have |*a*| ≤ |*b*| + |*c*| **if** *b* ≤ *a* ∧ *a* ≤ *c* **for** *a b c*::*real*
using *that* **by** *auto*
ultimately have ∗: *AE x in M*. |*q* (*real-cond-exp M F X x*)|
≤ |*real-cond-exp M F* (λ*x*. *q* (*X x*)) *x*| + |*A* ∗ *real-cond-exp M F X x*|
by *auto*

show *integrable M* (λ*x*. *q* (*real-cond-exp M F X x*))
apply (*rule Bochner-Integration.integrable-bound*[*of* - λ*x*. |*real-cond-exp M F* (λ*x*. *q* (*X x*)) *x*| + |*A* ∗ *real-cond-exp M F X x* |])
apply (*auto intro*!: *Bochner-Integration.integrable-add integrable-abs integrable-mult-right Bochner-Integration.integrable-diff real-cond-exp-int*(*1*))
using *X*(*1*) *q*(*1*) ∗ **by** *auto*
qed
qed

end

end

theory *Essential-Supremum*

imports *HOL−Analysis.Analysis*
begin

lemma *ae-filter-eq-bot-iff*: *ae-filter M = bot* ⟷ *emeasure M* (*space M*) = *0*
by (*simp add*: *AE-iff-measurable trivial-limit-def*)

29 The essential supremum

In this paragraph, we define the essential supremum and give its basic properties. The essential supremum of a function is its maximum value if one is allowed to throw away a set of measure 0. It is convenient to define it to be infinity for non-measurable functions, as it allows for neater statements in general. This is a prerequisiste to define the space L^∞.

definition *esssup*::*'a measure* ⇒ (*'a* ⇒ *'b*::{*second-countable-topology*, *dense-linorder*, *linorder-topology*, *complete-linorder*}) ⇒ *'b*
where *esssup M f* = (*if f* ∈ *borel-measurable M then Limsup* (*ae-filter M*) *f else top*)

lemma *esssup-non-measurable*: *f* ∉ *M* →$_M$ *borel* ⟹ *esssup M f* = *top*
by (*simp add*: *esssup-def*)

lemma *esssup-eq-AE*:
assumes *f*: *f* ∈ *M* →$_M$ *borel* **shows** *esssup M f* = *Inf* {*z*. *AE x in M*. *f x* ≤ *z*}
unfolding *esssup-def if-P*[*OF f*] *Limsup-def*
proof (*intro antisym INF-greatest Inf-greatest*; *clarsimp*)
fix *y* **assume** *AE x in M*. *f x* ≤ *y*
then have (λ*x*. *f x* ≤ *y*) ∈ {*P*. *AE x in M*. *P x*}
by *simp*
then show (*INF P*∈{*P*. *AE x in M*. *P x*}. *SUP x*∈*Collect P*. *f x*) ≤ *y*
by (*rule INF-lower2*) (*auto intro*: *SUP-least*)
next
fix *P* **assume** *P*: *AE x in M*. *P x*
show *Inf* {*z*. *AE x in M*. *f x* ≤ *z*} ≤ (*SUP x*∈*Collect P*. *f x*)
proof (*rule Inf-lower*; *clarsimp*)
show *AE x in M*. *f x* ≤ (*SUP x*∈*Collect P*. *f x*)
using *P* **by** (*auto elim*: *eventually-mono simp*: *SUP-upper*)
qed
qed

lemma *esssup-eq*: *f* ∈ *M* →$_M$ *borel* ⟹ *esssup M f* = *Inf* {*z*. *emeasure M* {*x* ∈ *space M*. *f x* > *z*} = *0*}
by (*auto simp add*: *esssup-eq-AE not-less*[*symmetric*] *AE-iff-measurable*[*OF - refl*] *intro*!: *arg-cong*[**where** *f*=*Inf*])

lemma *esssup-zero-measure*:
emeasure M {*x* ∈ *space M*. *f x* > *esssup M f*} = *0*
proof (*cases esssup M f* = *top*)
case *True*

then show *?thesis* **by** *auto*
next
case *False*
then have *f*[*measurable*]: $f \in M \rightarrow_M$ *borel* **unfolding** *esssup-def* **by** *meson*
have *esssup M f* < *top* **using** *False* **by** (*auto simp*: *less-top*)
have ∗: $\{x \in$ *space M*. *f x* > *z*$\} \in$ *null-sets M* **if** *z* > *esssup M f* **for** *z*
proof −
have ∃ *w*. *w* < *z* ∧ *emeasure M* $\{x \in$ *space M*. *f x* > *w*$\}$ = *0*
using ‹*z* > *esssup M f*› *f* **by** (*auto simp*: *esssup-eq Inf-less-iff*)
then obtain *w* **where** *w* < *z emeasure M* $\{x \in$ *space M*. *f x* > *w*$\}$ = *0* **by** *auto*
then have *a*: $\{x \in$ *space M*. *f x* > *w*$\} \in$ *null-sets M* **by** *auto*
have *b*: $\{x \in$ *space M*. *f x* > *z*$\} \subseteq \{x \in$ *space M*. *f x* > *w*$\}$ **using** ‹*w* < *z*› **by** *auto*
show *?thesis* **using** *null-sets-subset*[*OF a - b*] **by** *simp*
qed
obtain *u*::*nat* ⇒ *'b* **where** *u*: $\bigwedge n.$ *u n* > *esssup M f u* ⟶ *esssup M f*
using *approx-from-above-dense-linorder*[*OF* ‹*esssup M f* < *top*›] **by** *auto*
have $\{x \in$ *space M*. *f x* > *esssup M f*$\} = (\bigcup n. \{x \in$ *space M*. *f x* > *u n*$\})$
using *u* **apply** *auto*
apply (*metis* (*mono-tags*, *lifting*) *order-tendsto-iff eventually-mono LIMSEQ-unique*)
using *less-imp-le less-le-trans* **by** *blast*
also have ... ∈ *null-sets M*
using ∗[*OF u*(*1*)] **by** *auto*
finally show *?thesis* **by** *auto*
qed

lemma *esssup-AE*: *AE x in M*. *f x* ≤ *esssup M f*
proof (*cases f* ∈ *M* →$_M$ *borel*)
case *True* **then show** *?thesis*
by (*intro AE-I*[*OF - esssup-zero-measure*[*of - f*]]) *auto*
qed (*simp add*: *esssup-non-measurable*)

lemma *esssup-pos-measure*:
f ∈ *borel-measurable M* ⟹ *z* < *esssup M f* ⟹ *emeasure M* $\{x \in$ *space M*. *f x* > *z*$\}$ > *0*
using *Inf-less-iff mem-Collect-eq not-gr-zero* **by** (*force simp*: *esssup-eq*)

lemma *esssup-I* [*intro*]: *f* ∈ *borel-measurable M* ⟹ *AE x in M*. *f x* ≤ *c* ⟹ *esssup M f* ≤ *c*
unfolding *esssup-def* **by** (*simp add*: *Limsup-bounded*)

lemma *esssup-AE-mono*: *f* ∈ *borel-measurable M* ⟹ *AE x in M*. *f x* ≤ *g x* ⟹ *esssup M f* ≤ *esssup M g*
by (*auto simp*: *esssup-def Limsup-mono*)

lemma *esssup-mono*: *f* ∈ *borel-measurable M* ⟹ (⋀*x*. *f x* ≤ *g x*) ⟹ *esssup M f* ≤ *esssup M g*
by (*rule esssup-AE-mono*) *auto*

lemma *esssup-AE-cong*:
f ∈ *borel-measurable M* ⟹ *g* ∈ *borel-measurable M* ⟹ *AE x in M*. *f x* = *g x* ⟹ *esssup M f* = *esssup M g*
by (*auto simp*: *esssup-def intro*!: *Limsup-eq*)

lemma *esssup-const*: *emeasure M* (*space M*) ≠ *0* ⟹ *esssup M* (λ*x*. *c*) = *c*
by (*simp add*: *esssup-def Limsup-const ae-filter-eq-bot-iff*)

lemma *esssup-cmult*: **assumes** *c* > (*0*::*real*) **shows** *esssup M* (λ*x*. *c* ∗ *f x*::*ereal*) = *c* ∗ *esssup M f*
proof −
have (λ*x*. *ereal c* ∗ *f x*) ∈ *M* →$_M$ *borel* ⟹ *f* ∈ *M* →$_M$ *borel*
proof (*subst measurable-cong*)
fix ω **show** *f* ω = *ereal* (*1*/*c*) ∗ (*ereal c* ∗ *f* ω)
using ‹*0* < *c*› **by** (*cases f* ω) *auto*
qed *auto*
then have (λ*x*. *ereal c* ∗ *f x*) ∈ *M* →$_M$ *borel* ⟷ *f* ∈ *M* →$_M$ *borel*
by(*safe intro*!: *borel-measurable-ereal-times borel-measurable-const*)
with ‹*0*<*c*› **show** *?thesis*
by (*cases ae-filter M* = *bot*)
(*auto simp*: *esssup-def bot-ereal-def top-ereal-def Limsup-ereal-mult-left*)
qed

lemma *esssup-add*:
esssup M (λ*x*. *f x* + *g x*::*ereal*) ≤ *esssup M f* + *esssup M g*
proof (*cases f* ∈ *borel-measurable M* ∧ *g* ∈ *borel-measurable M*)
case *True*
then have [*measurable*]: (λ*x*. *f x* + *g x*) ∈ *borel-measurable M* **by** *auto*
have *f x* + *g x* ≤ *esssup M f* + *esssup M g* **if** *f x* ≤ *esssup M f g x* ≤ *esssup M g* **for** *x*
using *that add-mono* **by** *auto*
then have *AE x in M*. *f x* + *g x* ≤ *esssup M f* + *esssup M g*
using *esssup-AE*[*of f M*] *esssup-AE*[*of g M*] **by** *auto*
then show *?thesis* **using** *esssup-I* **by** *auto*
next
case *False*
then have *esssup M f* + *esssup M g* = ∞ **unfolding** *esssup-def top-ereal-def* **by** *auto*
then show *?thesis* **by** *auto*
qed

lemma *esssup-zero-space*:
emeasure M (*space M*) = *0* ⟹ *f* ∈ *borel-measurable M* ⟹ *esssup M f* = (−∞::*ereal*)
by (*simp add*: *esssup-def ae-filter-eq-bot-iff*[*symmetric*] *bot-ereal-def*)

end

30 Stopping times

theory *Stopping-Time*
imports *HOL−Analysis.Analysis*
begin

30.1 Stopping Time

This is also called strong stopping time. Then stopping time is T with alternative is $T\ x < t$ measurable.

definition *stopping-time* :: ('*t*::*linorder* ⇒ '*a measure*) ⇒ ('*a* ⇒ '*t*) ⇒ *bool*
where
stopping-time F T = (∀ *t*. *Measurable.pred* (*F t*) (λ*x*. *T x* ≤ *t*))

lemma *stopping-time-cong*: (⋀*t x*. *x* ∈ *space* (*F t*) ⟹ *T x* = *S x*) ⟹ *stopping-time F T* = *stopping-time F S*
unfolding *stopping-time-def* **by** (*intro arg-cong*[**where** *f*=*All*] *ext measurable-cong*) *simp*

lemma *stopping-timeD*: *stopping-time F T* ⟹ *Measurable.pred* (*F t*) (λ*x*. *T x* ≤ *t*)
by (*auto simp*: *stopping-time-def*)

lemma *stopping-timeD2*: *stopping-time F T* ⟹ *Measurable.pred* (*F t*) (λ*x*. *t* < *T x*)
unfolding *not-le*[*symmetric*] **by** (*auto intro*: *stopping-timeD Measurable.pred-intros-logic*)

lemma *stopping-timeI*[*intro?*]: (⋀*t*. *Measurable.pred* (*F t*) (λ*x*. *T x* ≤ *t*)) ⟹ *stopping-time F T*
by (*auto simp*: *stopping-time-def*)

lemma *measurable-stopping-time*:
fixes *T* :: '*a* ⇒ '*t*::{*linorder-topology*, *second-countable-topology*}
assumes *T*: *stopping-time F T*
and *M*: ⋀*t*. *sets* (*F t*) ⊆ *sets M* ⋀*t*. *space* (*F t*) = *space M*
shows *T* ∈ *M* →$_M$ *borel*
proof (*rule borel-measurableI-le*)
show {*x* ∈ *space M*. *T x* ≤ *t*} ∈ *sets M* **for** *t*
using *stopping-timeD*[*OF T*] *M* **by** (*auto simp*: *Measurable.pred-def*)
qed

lemma *stopping-time-const*: *stopping-time F* (λ*x*. *c*)
by (*auto simp*: *stopping-time-def*)

lemma *stopping-time-min*:
stopping-time F T ⟹ *stopping-time F S* ⟹ *stopping-time F* (λ*x*. *min* (*T x*) (*S x*))
by (*auto simp*: *stopping-time-def min-le-iff-disj intro*!: *pred-intros-logic*)

lemma *stopping-time-max*:
stopping-time F T $\Longrightarrow$ *stopping-time F S* $\Longrightarrow$ *stopping-time F* ($\lambda x.$ *max* (*T x*) (*S x*))
by (*auto simp*: *stopping-time-def intro*!: *pred-intros-logic*)

31 Filtration

locale *filtration* =
fixes Ω :: *'a set* **and** *F* :: *'t*::{*linorder-topology*, *second-countable-topology*} $\Rightarrow$ *'a measure*
assumes *space-F*: $\bigwedge i.$ *space* (*F i*) = Ω
assumes *sets-F-mono*: $\bigwedge i\ j.$ *i* $\leq$ *j* $\Longrightarrow$ *sets* (*F i*) $\subseteq$ *sets* (*F j*)
begin

31.1 σ-algebra of a Stopping Time

definition *pre-sigma* :: (*'a* $\Rightarrow$ *'t*) $\Rightarrow$ *'a measure*
where
pre-sigma T = *sigma* Ω {*A*. $\forall t.$ {$\omega$$\in$*A*. *T* ω $\leq$ *t*} $\in$ *sets* (*F t*)}

lemma *space-pre-sigma*: *space* (*pre-sigma T*) = Ω
unfolding *pre-sigma-def* **using** *sets.space-closed*[*of F* -]
by (*intro space-measure-of*) (*auto simp*: *space-F*)

lemma *measure-pre-sigma*[*simp*]: *emeasure* (*pre-sigma T*) = (λ-. *0*)
by (*simp add*: *pre-sigma-def emeasure-sigma*)

lemma *sigma-algebra-pre-sigma*:
assumes *T*: *stopping-time F T*
shows *sigma-algebra* Ω {*A*. $\forall t.$ {$\omega$$\in$*A*. *T* ω $\leq$ *t*} $\in$ *sets* (*F t*)}
unfolding *sigma-algebra-iff2*
proof (*intro sigma-algebra-iff2*[*THEN iffD2*] *conjI ballI allI impI CollectI*)
show {*A*. $\forall t.$ {ω $\in$ *A*. *T* ω $\leq$ *t*} $\in$ *sets* (*F t*)} $\subseteq$ *Pow* Ω
using *sets.space-closed*[*of F* -] **by** (*auto simp*: *space-F*)
next
fix *A t* **assume** *A* $\in$ {*A*. $\forall t.$ {ω $\in$ *A*. *T* ω $\leq$ *t*} $\in$ *sets* (*F t*)}
then have {ω $\in$ *space* (*F t*). *T* ω $\leq$ *t*} $-$ {ω $\in$ *A*. *T* ω $\leq$ *t*} $\in$ *sets* (*F t*)
using *T stopping-timeD*[*measurable*] **by** *auto*
also have {ω $\in$ *space* (*F t*). *T* ω $\leq$ *t*} $-$ {ω $\in$ *A*. *T* ω $\leq$ *t*} = {ω $\in$ Ω $-$ *A*. *T* ω $\leq$ *t*}
by (*auto simp*: *space-F*)
finally show {ω $\in$ Ω $-$ *A*. *T* ω $\leq$ *t*} $\in$ *sets* (*F t*) **.**
next
fix *AA* :: *nat* $\Rightarrow$ *'a set* **and** *t* **assume** *range AA* $\subseteq$ {*A*. $\forall t.$ {ω $\in$ *A*. *T* ω $\leq$ *t*} $\in$ *sets* (*F t*)}
then have ($\bigcup i.$ {ω $\in$ *AA i*. *T* ω $\leq$ *t*}) $\in$ *sets* (*F t*) **for** *t*
by *auto*
also have ($\bigcup i.$ {ω $\in$ *AA i*. *T* ω $\leq$ *t*}) = {ω $\in$ $\bigcup$(*AA* ' *UNIV*). *T* ω $\leq$ *t*}
by *auto*

finally show $\{\omega \in \bigcup (AA \text{ ` } UNIV).\ T\ \omega \leq t\} \in sets\ (F\ t)$.
qed *auto*

lemma *sets-pre-sigma*: *stopping-time F T* $\Longrightarrow$ *sets* (*pre-sigma T*) = $\{A.\ \forall\, t.\ \{\omega{\in}A.\ T\ \omega \leq t\} \in sets\ (F\ t)\}$
unfolding *pre-sigma-def* **by** (*rule sigma-algebra.sets-measure-of-eq*[*OF sigma-algebra-pre-sigma*])

lemma *sets-pre-sigmaI*: *stopping-time F T* $\Longrightarrow$ ($\bigwedge t.\ \{\omega{\in}A.\ T\ \omega \leq t\} \in sets\ (F\ t)) \Longrightarrow A \in sets\ (pre\text{-}sigma\ T)$
unfolding *sets-pre-sigma* **by** *auto*

lemma *pred-pre-sigmaI*:
assumes *T*: *stopping-time F T*
shows ($\bigwedge t.$ *Measurable.pred* (*F t*) ($\lambda\omega.\ P\ \omega \wedge T\ \omega \leq t$)) $\Longrightarrow$ *Measurable.pred* (*pre-sigma T*) *P*
unfolding *pred-def space-F space-pre-sigma* **by** (*intro sets-pre-sigmaI*[*OF T*]) *simp*

lemma *sets-pre-sigmaD*: *stopping-time F T* $\Longrightarrow A \in sets\ (pre\text{-}sigma\ T) \Longrightarrow \{\omega{\in}A.\ T\ \omega \leq t\} \in sets\ (F\ t)$
unfolding *sets-pre-sigma* **by** *auto*

lemma *stopping-time-le-const*: *stopping-time F T* $\Longrightarrow s \leq t \Longrightarrow$ *Measurable.pred* (*F t*) ($\lambda\omega.\ T\ \omega \leq s$)
using *stopping-timeD*[*of F T*] *sets-F-mono*[*of - t*] **by** (*auto simp*: *pred-def space-F*)

lemma *measurable-stopping-time-pre-sigma*:
assumes *T*: *stopping-time F T* **shows** $T \in pre\text{-}sigma\ T \rightarrow_M borel$
proof (*intro borel-measurableI-le sets-pre-sigmaI*[*OF T*])
fix *t t'*
have $\{\omega{\in}space\ (F\ (min\ t'\ t)).\ T\ \omega \leq min\ t'\ t\} \in sets\ (F\ (min\ t'\ t))$
using *T* **unfolding** *pred-def*[*symmetric*] **by** (*rule stopping-timeD*)
also have $\ldots \subseteq sets\ (F\ t)$
by (*rule sets-F-mono*) *simp*
finally show $\{\omega \in \{x \in space\ (pre\text{-}sigma\ T).\ T\ x \leq t'\}.\ T\ \omega \leq t\} \in sets\ (F\ t)$
by (*simp add*: *space-pre-sigma space-F*)
qed

lemma *mono-pre-sigma*:
assumes *T*: *stopping-time F T* **and** *S*: *stopping-time F S*
and *le*: $\bigwedge\omega.\ \omega \in \Omega \Longrightarrow T\ \omega \leq S\ \omega$
shows *sets* (*pre-sigma T*) $\subseteq$ *sets* (*pre-sigma S*)
unfolding *sets-pre-sigma*[*OF S*] *sets-pre-sigma*[*OF T*]
proof *safe*
interpret *sigma-algebra* Ω $\{A.\ \forall\, t.\ \{\omega{\in}A.\ T\ \omega \leq t\} \in sets\ (F\ t)\}$
using *T* **by** (*rule sigma-algebra-pre-sigma*)
fix *A t* **assume** *A*: $\forall\, t.\ \{\omega{\in}A.\ T\ \omega \leq t\} \in sets\ (F\ t)$
then have $A \subseteq \Omega$
using *sets-into-space* **by** *auto*

from A **have** $\{\omega{\in}A.\ T\ \omega \leq t\} \cap \{\omega{\in}space\ (F\ t).\ S\ \omega \leq t\} \in sets\ (F\ t)$
using *stopping-timeD*[*OF S*] **by** (*auto simp*: *pred-def*)
also have $\{\omega{\in}A.\ T\ \omega \leq t\} \cap \{\omega{\in}space\ (F\ t).\ S\ \omega \leq t\} = \{\omega{\in}A.\ S\ \omega \leq t\}$
using ‹$A \subseteq \Omega$› *sets-into-space*[*of A*] *le* **by** (*auto simp*: *space-F intro*: *order-trans*)
finally show $\{\omega{\in}A.\ S\ \omega \leq t\} \in sets\ (F\ t)$
by *auto*
qed

lemma *stopping-time-less-const*:
assumes *T*: *stopping-time F T* **shows** *Measurable.pred* $(F\ t)\ (\lambda\omega.\ T\ \omega < t)$
proof −
obtain $D :: 't\ set$
where *D*: *countable D* $\bigwedge X.$ *open* $X \Longrightarrow X \neq \{\} \Longrightarrow \exists\, d{\in}D.\ d \in X$
using *countable-dense-setE* **by** *auto*
show *?thesis*
proof *cases*
assume $*$: $\forall\, t'{<}t.\ \exists\, t''.\ t' < t'' \wedge t'' < t$
{ fix t' **assume** $t' < t$
with $*$ **have** $\{t' <..< t\} \neq \{\}$
by *fastforce*
with *D*(*2*)[*OF - this*]
have $\exists\, d{\in}D.\ t' < d \wedge d < t$
by *auto* **}**
note $** =$ *this*

show *?thesis*
proof (*rule measurable-cong*[*THEN iffD2*])
show $T\ \omega < t \longleftrightarrow (\exists\, r{\in}\{r{\in}D.\ r < t\}.\ T\ \omega \leq r)$ **for** ω
by (*auto dest*: $**$ *intro*: *less-imp-le*)
show *Measurable.pred* $(F\ t)\ (\lambda w.\ \exists\, r{\in}\{r \in D.\ r < t\}.\ T\ w \leq r)$
by (*intro measurable-pred-countable stopping-time-le-const*[*OF T*] *countable-Collect D*) *auto*
qed
next
assume $\neg\ (\forall\, t'{<}t.\ \exists\, t''.\ t' < t'' \wedge t'' < t)$
then obtain t' **where** t': $t' < t \bigwedge t''.\ t'' < t \Longrightarrow t'' \leq t'$
by (*auto simp*: *not-less*[*symmetric*])
show *?thesis*
proof (*rule measurable-cong*[*THEN iffD2*])
show $T\ \omega < t \longleftrightarrow T\ \omega \leq t'$ **for** ω
using t' **by** *auto*
show *Measurable.pred* $(F\ t)\ (\lambda w.\ T\ w \leq t')$
using ‹$t'{<}t$› **by** (*intro stopping-time-le-const*[*OF T*]) *auto*
qed
qed
qed

lemma *stopping-time-eq-const*: *stopping-time F T* $\Longrightarrow$ *Measurable.pred* $(F\ t)\ (\lambda\omega.\ T\ \omega = t)$

unfolding *eq-iff* **using** *stopping-time-less-const*[*of T t*]
by (*intro pred-intros-logic stopping-time-le-const*) (*auto simp*: *not-less*[*symmetric*]
)

lemma *stopping-time-less*:
assumes *T*: *stopping-time F T* **and** *S*: *stopping-time F S*
shows *Measurable.pred* (*pre-sigma T*) (λω. *T ω* < *S ω*)
proof (*rule pred-pre-sigmaI*[*OF T*])
fix *t*
obtain *D* :: *'t set*
where [*simp*]: *countable D* **and** *semidense-D*: ⋀*x y*. *x* < *y* ⟹ (∃ *b*∈*D*. *x* ≤ *b*
∧ *b* < *y*)
using *countable-separating-set-linorder2* **by** *auto*
show *Measurable.pred* (*F t*) (λω. *T ω* < *S ω* ∧ *T ω* ≤ *t*)
proof (*rule measurable-cong*[*THEN iffD2*])
let *?f* = λω. *if T ω* = *t then* ¬ *S ω* ≤ *t else* ∃ *s*∈{*s*∈*D*. *s* ≤ *t*}. *T ω* ≤ *s* ∧ ¬
(*S ω* ≤ *s*)
{ fix ω **assume** *T ω* ≤ *t T ω* ≠ *t T ω* < *S ω*
then have *T ω* < *min t* (*S ω*)
by *auto*
then obtain *r* **where** *r* ∈ *D T ω* ≤ *r r* < *min t* (*S ω*)
by (*metis semidense-D*)
then have ∃ *s*∈{*s*∈*D*. *s* ≤ *t*}. *T ω* ≤ *s* ∧ *s* < *S ω*
by *auto* **}**
then show (*T ω* < *S ω* ∧ *T ω* ≤ *t*) = *?f ω* **for** ω
by (*auto simp*: *not-le*)
show *Measurable.pred* (*F t*) *?f*
by (*intro pred-intros-logic measurable-If measurable-pred-countable count-
able-Collect*
stopping-time-le-const predE stopping-time-eq-const T S)
auto
qed
qed

end

lemma *stopping-time-SUP-enat*:
fixes *T* :: *nat* ⇒ (*'a* ⇒ *enat*)
shows (⋀*i*. *stopping-time F* (*T i*)) ⟹ *stopping-time F* (*SUP i*. *T i*)
unfolding *stopping-time-def SUP-apply SUP-le-iff* **by** (*auto intro*!: *pred-intros-countable*)

lemma *less-eSuc-iff*: *a* < *eSuc b* ⟷ (*a* ≤ *b* ∧ *a* ≠ ∞)
by (*cases a*) *auto*

lemma *stopping-time-Inf-enat*:
fixes *F* :: *enat* ⇒ *'a measure*
assumes *F*: *filtration* Ω *F*
assumes *P*: ⋀*i*. *Measurable.pred* (*F i*) (*P i*)
shows *stopping-time F* (λω. *Inf* {*i*. *P i ω*})

proof (*rule stopping-timeI*, *cases*)
fix t :: *enat* **assume** $t = \infty$ **then show** *Measurable.pred* (*F t*) ($\lambda\omega$. *Inf* {*i*. *P i* ω} $\leq$ *t*)
by *auto*
next
fix t :: *enat* **assume** $t \neq \infty$
moreover
{ fix *i* ω **assume** *Inf* {*i*. *P i* ω} $\leq$ *t*
with ‹*t* $\neq \infty$› **have** ($\exists$ *i*$\leq$*t*. *P i* ω)
unfolding *Inf-le-iff* **by** (*cases t*) (*auto elim*!: *allE*[*of* - *eSuc t*] *simp*: *less-eSuc-iff*)
}
ultimately have *:* $\bigwedge\omega$. *Inf* {*i*. *P i* ω} $\leq$ *t* $\longleftrightarrow$ ($\exists$ *i*$\in${..*t*}. *P i* ω)
by (*auto intro*!: *Inf-lower2*)
show *Measurable.pred* (*F t*) ($\lambda\omega$. *Inf* {*i*. *P i* ω} $\leq$ *t*)
unfolding $*$ **using** *filtration.sets-F-mono*[*OF F*, *of* - *t*] *P*
by (*intro pred-intros-countable-bounded*) (*auto simp*: *pred-def filtration.space-F*[*OF F*])
qed

lemma *stopping-time-Inf-nat*:
fixes *F* :: *nat* $\Rightarrow$ *'a measure*
assumes *F*: *filtration* Ω *F*
assumes *P*: $\bigwedge$*i*. *Measurable.pred* (*F i*) (*P i*) **and** *wf*: $\bigwedge$*i* ω. $\omega \in \Omega \Longrightarrow \exists$ *n*. *P n* ω
shows *stopping-time F* ($\lambda\omega$. *Inf* {*i*. *P i* ω})
unfolding *stopping-time-def*
proof (*intro allI*, *subst measurable-cong*)
fix *t* ω **assume** $\omega \in$ *space* (*F t*)
then have $\omega \in \Omega$
using *filtration.space-F*[*OF F*] **by** *auto*
from *wf*[*OF this*] **have** ((*LEAST n*. *P n* ω) $\leq$ *t*) = ($\exists$ *i*$\leq$*t*. *P i* ω)
by (*rule LeastI2-wellorder-ex*) *auto*
then show (*Inf* {*i*. *P i* ω} $\leq$ *t*) = ($\exists$ *i*$\in${..*t*}. *P i* ω)
by (*simp add*: *Inf-nat-def Bex-def*)
next
fix *t* **from** *P* **show** *Measurable.pred* (*F t*) (λ*w*. $\exists$ *i*$\in${..*t*}. *P i w*)
using *filtration.sets-F-mono*[*OF F*, *of* - *t*]
by (*intro pred-intros-countable-bounded*) (*auto simp*: *pred-def filtration.space-F*[*OF F*])
qed

end

theory *Probability*
imports
Central-Limit-Theorem
Discrete-Topology
PMF-Impl

Projective-Limit
Random-Permutations
SPMF
Product-PMF
Hoeffding
Stream-Space
Tree-Space
Conditional-Expectation
Essential-Supremum
Stopping-Time
begin

end