Notable Examples in Isabelle/HOL

December 12, 2021

Contents

1 Ad Hoc Overloading

theory *Adhoc_Overloading_Examples*
imports
  Main
  HOL−Library.Infinite_Set
  HOL−Library.Adhoc_Overloading
begin

Adhoc overloading allows to overload a constant depending on its type. Typically this involves to introduce an uninterpreted constant (used for input and output) and then add some variants (used internally).

1.1 Plain Ad Hoc Overloading

Consider the type of first-order terms.

datatype ('a, 'b) *term* =
  Var 'b |
  Fun 'a ('a, 'b) *term list*

The set of variables of a term might be computed as follows.

fun $\mathit{term\_vars} :: ('a,\ 'b)\ \mathit{term} \Rightarrow\ 'b\ \mathit{set}$ **where**
  $\mathit{term\_vars}\ (\mathit{Var}\ x) = \{x\}\ |$
  $\mathit{term\_vars}\ (\mathit{Fun}\ f\ \mathit{ts}) = \bigcup(\mathit{set}\ (\mathit{map}\ \mathit{term\_vars}\ \mathit{ts}))$

However, also for *rules* (i.e., pairs of terms) and term rewrite systems (i.e., sets of rules), the set of variables makes sense. Thus we introduce an unspecified constant *vars*.

consts $\mathit{vars} ::\ 'a \Rightarrow\ 'b\ \mathit{set}$

Which is then overloaded with variants for terms, rules, and TRSs.

adhoc_overloading
  vars term_vars

value $[\mathit{nbe}]\ \mathit{vars}\ (\mathit{Fun}\ ''f''\ [\mathit{Var}\ 0,\ \mathit{Var}\ 1])$

fun $\mathit{rule\_vars} :: ('a,\ 'b)\ \mathit{term} \times ('a,\ 'b)\ \mathit{term} \Rightarrow\ 'b\ \mathit{set}$ **where**
  $\mathit{rule\_vars}\ (l,\ r) = \mathit{vars}\ l \cup \mathit{vars}\ r$

adhoc_overloading
  vars rule_vars

value $[\mathit{nbe}]\ \mathit{vars}\ (\mathit{Var}\ 1,\ \mathit{Var}\ 0)$

definition $\mathit{trs\_vars} :: (('a,\ 'b)\ \mathit{term} \times ('a,\ 'b)\ \mathit{term})\ \mathit{set} \Rightarrow\ 'b\ \mathit{set}$ **where**
  $\mathit{trs\_vars}\ R = \bigcup(\mathit{rule\_vars}\ `\ R)$

adhoc_overloading
  vars trs_vars

value $[\mathit{nbe}]\ \mathit{vars}\ \{(\mathit{Var}\ 1,\ \mathit{Var}\ 0)\}$

Sometimes it is necessary to add explicit type constraints before a variant can be determined.

value $\mathit{vars}\ (R :: (('a,\ 'b)\ \mathit{term} \times ('a,\ 'b)\ \mathit{term})\ \mathit{set})$

It is also possible to remove variants.

no_adhoc_overloading
  vars term_vars rule_vars

As stated earlier, the overloaded constant is only used for input and output. Internally, always a variant is used, as can be observed by the configuration option *show_variants*.

adhoc_overloading
  vars term_vars

declare [[*show_variants*]]

term *vars* (*Var 1*)

1.2 Adhoc Overloading inside Locales

As example we use permutations that are parametrized over an atom type $'a$.

definition *perms* :: $('a \Rightarrow 'a)$ *set* **where**
  perms = $\{f.$ *bij* $f \wedge$ *finite* $\{x.\ f\ x \neq x\}\}$

typedef $'a$ *perm* = *perms* :: $('a \Rightarrow 'a)$ *set*
  by *standard* (*auto simp*: *perms_def*)

First we need some auxiliary lemmas.

lemma *permsI* [*Pure.intro*]:
  assumes *bij f* **and** *MOST x. f x = x*
  shows $f \in$ *perms*
  using *assms* **by** (*auto simp*: *perms_def*) (*metis MOST_iff_finiteNeg*)

lemma *perms_imp_bij*:
  $f \in$ *perms* $\Longrightarrow$ *bij f*
  by (*simp add*: *perms_def*)

lemma *perms_imp_MOST_eq*:
  $f \in$ *perms* $\Longrightarrow$ *MOST x. f x = x*
  by (*simp add*: *perms_def*) (*metis MOST_iff_finiteNeg*)

lemma *id_perms* [*simp*]:
  id $\in$ *perms*
  $(\lambda x.\ x) \in$ *perms*
  by (*auto simp*: *perms_def bij_def*)

lemma *perms_comp* [*simp*]:
  assumes *f*: $f \in$ *perms* **and** *g*: $g \in$ *perms*
  shows $(f \circ g) \in$ *perms*
  apply (*intro permsI bij_comp*)
  apply (*rule perms_imp_bij* [*OF g*])
  apply (*rule perms_imp_bij* [*OF f*])
  apply (*rule MOST_rev_mp* [*OF perms_imp_MOST_eq* [*OF g*]])
  apply (*rule MOST_rev_mp* [*OF perms_imp_MOST_eq* [*OF f*]])
  by *simp*

lemma *perms_inv*:
  assumes *f*: $f \in$ *perms*
  shows *inv f* $\in$ *perms*
  apply (*rule permsI*)
  apply (*rule bij_imp_bij_inv*)
  apply (*rule perms_imp_bij* [*OF f*])

```
  apply (rule MOST_mono [OF perms_imp_MOST_eq [OF f]])
  apply (erule subst, rule inv_f_f)
  apply (rule bij_is_inj [OF perms_imp_bij [OF f]])
  done

lemma bij_Rep_perm: bij (Rep_perm p)
  using Rep_perm [of p] unfolding perms_def by simp

instantiation perm :: (type) group_add
begin

definition 0 = Abs_perm id
definition − p = Abs_perm (inv (Rep_perm p))
definition p + q = Abs_perm (Rep_perm p ∘ Rep_perm q)
definition (p1::'a perm) − p2 = p1 + − p2

lemma Rep_perm_0: Rep_perm 0 = id
  unfolding zero_perm_def by (simp add: Abs_perm_inverse)

lemma Rep_perm_add:
  Rep_perm (p1 + p2) = Rep_perm p1 ∘ Rep_perm p2
  unfolding plus_perm_def by (simp add: Abs_perm_inverse Rep_perm)

lemma Rep_perm_uminus:
  Rep_perm (− p) = inv (Rep_perm p)
  unfolding uminus_perm_def by (simp add: Abs_perm_inverse perms_inv Rep_perm)

instance
  apply standard
  unfolding Rep_perm_inject [symmetric]
  unfolding minus_perm_def
  unfolding Rep_perm_add
  unfolding Rep_perm_uminus
  unfolding Rep_perm_0
  apply (simp_all add: o_assoc inv_o_cancel [OF bij_is_inj [OF bij_Rep_perm]])
  done

end

lemmas Rep_perm_simps =
  Rep_perm_0
  Rep_perm_add
  Rep_perm_uminus
```

2 Permutation Types

We want to be able to apply permutations to arbitrary types. To this end we introduce a constant *PERMUTE* together with convenient infix syntax.

consts *PERMUTE* :: *'a perm* ⇒ *'b* ⇒ *'b* (**infixr** · *75*)

Then we add a locale for types *'b* that support appliciation of permutations.

locale *permute* =
  fixes *permute* :: *'a perm* ⇒ *'b* ⇒ *'b*
  assumes *permute_zero* [*simp*]: *permute 0 x* = *x*
    and *permute_plus* [*simp*]: *permute* (*p* + *q*) *x* = *permute p* (*permute q x*)
begin

adhoc_overloading
  PERMUTE permute

end

Permuting atoms.

definition *permute_atom* :: *'a perm* ⇒ *'a* ⇒ *'a* **where**
  permute_atom p a = (*Rep_perm p*) *a*

adhoc_overloading
  PERMUTE permute_atom

interpretation *atom_permute*: *permute permute_atom*
  by *standard* (*simp_all add*: *permute_atom_def Rep_perm_simps*)

Permuting permutations.

definition *permute_perm* :: *'a perm* ⇒ *'a perm* ⇒ *'a perm* **where**
  permute_perm p q = *p* + *q* − *p*

adhoc_overloading
  PERMUTE permute_perm

interpretation *perm_permute*: *permute permute_perm*
  apply *standard*
  unfolding *permute_perm_def*
  apply *simp*
  apply (*simp only*: *diff_conv_add_uminus minus_add add.assoc*)
  done

Permuting functions.

locale *fun_permute* =
  dom: *permute perm1* + *ran*: *permute perm2*
  for *perm1* :: *'a perm* ⇒ *'b* ⇒ *'b*
  and *perm2* :: *'a perm* ⇒ *'c* ⇒ *'c*
begin

adhoc_overloading
  PERMUTE perm1 perm2

definition *permute_fun* :: $'a$ *perm* $\Rightarrow$ $('b \Rightarrow 'c) \Rightarrow ('b \Rightarrow 'c)$ **where**
permute_fun p f = $(\lambda x.\ p \cdot (f\ (-p \cdot x)))$

adhoc_overloading
PERMUTE permute_fun

end

sublocale *fun_permute* $\subseteq$ *permute permute_fun*
by (*unfold_locales*, *auto simp*: *permute_fun_def*)
(*metis dom.permute_plus minus_add*)

lemma (*Abs_perm id* :: *nat perm*) $\cdot$ *Suc 0* = *Suc 0*
unfolding *permute_atom_def*
by (*metis Rep_perm_0 id_apply zero_perm_def*)

interpretation *atom_fun_permute*: *fun_permute permute_atom permute_atom*
by (*unfold_locales*)

adhoc_overloading
PERMUTE atom_fun_permute.permute_fun

lemma (*Abs_perm id* :: $'a$ *perm*) $\cdot$ *id* = *id*
unfolding *atom_fun_permute.permute_fun_def*
unfolding *permute_atom_def*
by (*metis Rep_perm_0 id_def inj_imp_inv_eq inj_on_id uminus_perm_def zero_perm_def*)

end

3 A Tail-Recursive, Stack-Based Ackermann's Function

theory *Ackermann* **imports** *Main*

begin

This theory investigates a stack-based implementation of Ackermann's function. Let's recall the traditional definition, as modified by Rózsa Péter and Raphael Robinson.

fun *ack* :: [*nat*,*nat*] $\Rightarrow$ *nat* **where**
ack 0 n = *Suc n*
| *ack* (*Suc m*) *0* = *ack m 1*
| *ack* (*Suc m*) (*Suc n*) = *ack m* (*ack* (*Suc m*) *n*)

Here is the stack-based version, which uses lists.

function (*domintros*) *ackloop* :: *nat list* $\Rightarrow$ *nat* **where**
ackloop (*n* # *0* # *l*) = *ackloop* (*Suc n* # *l*)

| *ackloop* (*0* # *Suc m* # *l*) = *ackloop* (*1* # *m* # *l*)
| *ackloop* (*Suc n* # *Suc m* # *l*) = *ackloop* (*n* # *Suc m* # *m* # *l*)
| *ackloop* [*m*] = *m*
| *ackloop* [] = *0*
by *pat_completeness auto*

The key task is to prove termination. In the first recursive call, the head of the list gets bigger while the list gets shorter, suggesting that the length of the list should be the primary termination criterion. But in the third recursive call, the list gets longer. The idea of trying a multiset-based termination argument is frustrated by the second recursive call when m = 0: the list elements are simply permuted.

Fortunately, the function definition package allows us to define a function and only later identify its domain of termination. Instead, it makes all the recursion equations conditional on satisfying the function's domain predicate. Here we shall eventually be able to show that the predicate is always satisfied.

ackloop_dom (*Suc n* # *l*) $\Longrightarrow$ *ackloop_dom* (*n* # *0* # *l*)
ackloop_dom (*Suc 0* # *m* # *l*) $\Longrightarrow$ *ackloop_dom* (*0* # *Suc m* # *l*)
ackloop_dom (*n* # *Suc m* # *m* # *l*) $\Longrightarrow$ *ackloop_dom* (*Suc n* # *Suc m* # *l*)
ackloop_dom [*m*]
ackloop_dom []

declare *ackloop.domintros* [*simp*]

Termination is trivial if the length of the list is less then two. The following lemma is the key to proving termination for longer lists.

lemma *ackloop_dom* (*ack m n* # *l*) $\Longrightarrow$ *ackloop_dom* (*n* # *m* # *l*)
proof (*induction m arbitrary*: *n l*)
  case *0*
  then show *?case*
    by *auto*
next
  case (*Suc m*)
  show *?case*
    using *Suc.prems*
    by (*induction n arbitrary*: *l*) (*simp_all add*: *Suc*)
qed

The proof above (which actually is unused) can be expressed concisely as follows.

lemma *ackloop_dom_longer*:
  ackloop_dom (*ack m n* # *l*) $\Longrightarrow$ *ackloop_dom* (*n* # *m* # *l*)
  by (*induction m n arbitrary*: *l rule*: *ack.induct*) *auto*

lemma *ackloop_dom* (*ack m n # l*) $\Longrightarrow$ *ackloop_dom* (*n # m # l*)
by (*induction m n arbitrary*: *l rule*: *ack.induct*) *auto*

This function codifies what *ackloop* is designed to do. Proving the two functions equivalent also shows that *ackloop* can be used to compute Ackermann's function.

fun *acklist* :: *nat list* ⇒ *nat* **where**
acklist (*n#m#l*) = *acklist* (*ack m n # l*)
| *acklist* [*m*] = *m*
| *acklist* [] = *0*

The induction rule for *acklist* is

$$[\![\bigwedge n\ m\ l.\ P\ (ack\ m\ n\ \#\ l) \Longrightarrow P\ (n\ \#\ m\ \#\ l);\ \bigwedge m.\ P\ [m];\ P\ []]\!] \Longrightarrow P\ a0$$

.

lemma *ackloop_dom*: *ackloop_dom l*
by (*induction l rule*: *acklist.induct*) (*auto simp*: *ackloop_dom_longer*)

termination *ackloop*
by (*simp add*: *ackloop_dom*)

This result is trivial even by inspection of the function definitions (which faithfully follow the definition of Ackermann's function). All that we needed was termination.

lemma *ackloop_acklist*: *ackloop l* = *acklist l*
by (*induction l rule*: *ackloop.induct*) *auto*

theorem *ack*: *ack m n* = *ackloop* [*n*,*m*]
by (*simp add*: *ackloop_acklist*)

end

4 Cantor's Theorem

theory *Cantor*
imports *Main*
begin

4.1 Mathematical statement and proof

Cantor's Theorem states that there is no surjection from a set to its powerset. The proof works by diagonalization. E.g. see

- http://mathworld.wolfram.com/CantorDiagonalMethod.html
- https://en.wikipedia.org/wiki/Cantor's_diagonal_argument

theorem *Cantor*: $\nexists f :: {'a} \Rightarrow {'a}\ set.\ \forall A.\ \exists x.\ A = f\ x$
proof
  assume $\exists f :: {'a} \Rightarrow {'a}\ set.\ \forall A.\ \exists x.\ A = f\ x$
  then obtain $f :: {'a} \Rightarrow {'a}\ set$ **where** $*$: $\forall A.\ \exists x.\ A = f\ x$ **..**
  let $?D = \{x.\ x \notin f\ x\}$
  from $*$ **obtain** a **where** $?D = f\ a$ **by** *blast*
  moreover have $a \in\ ?D \longleftrightarrow a \notin f\ a$ **by** *blast*
  ultimately show *False* **by** *blast*
qed

4.2 Automated proofs

These automated proofs are much shorter, but lack information why and how it works.

theorem $\nexists f :: {'a} \Rightarrow {'a}\ set.\ \forall A.\ \exists x.\ f\ x = A$
  by *best*

theorem $\nexists f :: {'a} \Rightarrow {'a}\ set.\ \forall A.\ \exists x.\ f\ x = A$
  by *force*

4.3 Elementary version in higher-order predicate logic

The subsequent formulation bypasses set notation of HOL; it uses elementary λ-calculus and predicate logic, with standard introduction and elimination rules. This also shows that the proof does not require classical reasoning.

lemma *iff_contradiction*:
  assumes $*$: $\neg A \longleftrightarrow A$
  shows *False*
proof (*rule notE*)
  show $\neg A$
  proof
    assume A
    with $*$ **have** $\neg A$ **..**
    from *this* **and** $\langle A \rangle$ **show** *False* **..**
  qed
  with $*$ **show** A **..**
qed

theorem *Cantor'*: $\nexists f :: {'a} \Rightarrow {'a} \Rightarrow bool.\ \forall A.\ \exists x.\ A = f\ x$
proof
  assume $\exists f :: {'a} \Rightarrow {'a} \Rightarrow bool.\ \forall A.\ \exists x.\ A = f\ x$
  then obtain $f :: {'a} \Rightarrow {'a} \Rightarrow bool$ **where** $*$: $\forall A.\ \exists x.\ A = f\ x$ **..**
  let $?D = \lambda x.\ \neg f\ x\ x$
  from $*$ **have** $\exists x.\ ?D = f\ x$ **..**
  then obtain a **where** $?D = f\ a$ **..**
  then have $?D\ a \longleftrightarrow f\ a\ a$ **by** (*rule arg_cong*)
  then have $\neg f\ a\ a \longleftrightarrow f\ a\ a$ **.**

```
  then show False by (rule iff_contradiction)
qed
```

4.4 Classic Isabelle/HOL example

The following treatment of Cantor's Theorem follows the classic example from the early 1990s, e.g. see the file `92/HOL/ex/set.ML` in Isabelle92 or [2, §18.7]. The old tactic scripts synthesize key information of the proof by refinement of schematic goal states. In contrast, the Isar proof needs to say explicitly what is proven.

Cantor's Theorem states that every set has more subsets than it has elements. It has become a favourite basic example in pure higher-order logic since it is so easily expressed:

$$\forall f{::}\alpha \Rightarrow \alpha \Rightarrow bool.\ \exists S{::}\alpha \Rightarrow bool.\ \forall x{::}\alpha.\ f\ x \neq S$$

Viewing types as sets, $\alpha \Rightarrow bool$ represents the powerset of α. This version of the theorem states that for every function from α to its powerset, some subset is outside its range. The Isabelle/Isar proofs below uses HOL's set theory, with the type α *set* and the operator *range* :: $(\alpha \Rightarrow \beta) \Rightarrow \beta$ *set*.

```
theorem ∃S. S ∉ range (f :: 'a ⇒ 'a set)
proof
  let ?S = {x. x ∉ f x}
  show ?S ∉ range f
  proof
    assume ?S ∈ range f
    then obtain y where ?S = f y ..
    then show False
    proof (rule equalityCE)
      assume y ∈ f y
      assume y ∈ ?S
      then have y ∉ f y ..
      with ‹y ∈ f y› show ?thesis by contradiction
    next
      assume y ∉ ?S
      assume y ∉ f y
      then have y ∈ ?S ..
      with ‹y ∉ ?S› show ?thesis by contradiction
    qed
  qed
qed
```

How much creativity is required? As it happens, Isabelle can prove this theorem automatically using best-first search. Depth-first search would diverge, but best-first search successfully navigates through the large search space.

The context of Isabelle's classical prover contains rules for the relevant constructs of HOL's set theory.

theorem $\exists S.\ S \notin \mathit{range}\ (f :: 'a \Rightarrow 'a\ \mathit{set})$
  by *best*

end

5 Coherent Logic Problems

theory *Coherent*
imports *Main*
begin

5.1 Equivalence of two versions of Pappus' Axiom

no_notation
  comp (**infixl** *o 55*) **and**
  relcomp (**infixr** *O 75*)

lemma *p1p2*:
  assumes $\mathit{col}\ a\ b\ c\ l \wedge \mathit{col}\ d\ e\ f\ m$
    and $\mathit{col}\ b\ f\ g\ n \wedge \mathit{col}\ c\ e\ g\ o$
    and $\mathit{col}\ b\ d\ h\ p \wedge \mathit{col}\ a\ e\ h\ q$
    and $\mathit{col}\ c\ d\ i\ r \wedge \mathit{col}\ a\ f\ i\ s$
    and $\mathit{el}\ n\ o \Longrightarrow \mathit{goal}$
    and $\mathit{el}\ p\ q \Longrightarrow \mathit{goal}$
    and $\mathit{el}\ s\ r \Longrightarrow \mathit{goal}$
    and $\bigwedge A.\ \mathit{el}\ A\ A \Longrightarrow \mathit{pl}\ g\ A \Longrightarrow \mathit{pl}\ h\ A \Longrightarrow \mathit{pl}\ i\ A \Longrightarrow \mathit{goal}$
    and $\bigwedge A\ B\ C\ D.\ \mathit{col}\ A\ B\ C\ D \Longrightarrow \mathit{pl}\ A\ D$
    and $\bigwedge A\ B\ C\ D.\ \mathit{col}\ A\ B\ C\ D \Longrightarrow \mathit{pl}\ B\ D$
    and $\bigwedge A\ B\ C\ D.\ \mathit{col}\ A\ B\ C\ D \Longrightarrow \mathit{pl}\ C\ D$
    and $\bigwedge A\ B.\ \mathit{pl}\ A\ B \Longrightarrow \mathit{ep}\ A\ A$
    and $\bigwedge A\ B.\ \mathit{ep}\ A\ B \Longrightarrow \mathit{ep}\ B\ A$
    and $\bigwedge A\ B\ C.\ \mathit{ep}\ A\ B \Longrightarrow \mathit{ep}\ B\ C \Longrightarrow \mathit{ep}\ A\ C$
    and $\bigwedge A\ B.\ \mathit{pl}\ A\ B \Longrightarrow \mathit{el}\ B\ B$
    and $\bigwedge A\ B.\ \mathit{el}\ A\ B \Longrightarrow \mathit{el}\ B\ A$
    and $\bigwedge A\ B\ C.\ \mathit{el}\ A\ B \Longrightarrow \mathit{el}\ B\ C \Longrightarrow \mathit{el}\ A\ C$
    and $\bigwedge A\ B\ C.\ \mathit{ep}\ A\ B \Longrightarrow \mathit{pl}\ B\ C \Longrightarrow \mathit{pl}\ A\ C$
    and $\bigwedge A\ B\ C.\ \mathit{pl}\ A\ B \Longrightarrow \mathit{el}\ B\ C \Longrightarrow \mathit{pl}\ A\ C$
    and $\bigwedge A\ B\ C\ D\ E\ F\ G\ H\ I\ J\ K\ L\ M\ N\ O\ P\ Q.$
        $\mathit{col}\ A\ B\ C\ D \Longrightarrow \mathit{col}\ E\ F\ G\ H \Longrightarrow \mathit{col}\ B\ G\ I\ J \Longrightarrow \mathit{col}\ C\ F\ I\ K \Longrightarrow$
        $\mathit{col}\ B\ E\ L\ M \Longrightarrow \mathit{col}\ A\ F\ L\ N \Longrightarrow \mathit{col}\ C\ E\ O\ P \Longrightarrow \mathit{col}\ A\ G\ O\ Q \Longrightarrow$
        $(\exists\ R.\ \mathit{col}\ I\ L\ O\ R) \vee \mathit{pl}\ A\ H \vee \mathit{pl}\ B\ H \vee \mathit{pl}\ C\ H \vee \mathit{pl}\ E\ D \vee \mathit{pl}\ F\ D \vee \mathit{pl}$
$G\ D$
    and $\bigwedge A\ B\ C\ D.\ \mathit{pl}\ A\ B \Longrightarrow \mathit{pl}\ A\ C \Longrightarrow \mathit{pl}\ D\ B \Longrightarrow \mathit{pl}\ D\ C \Longrightarrow \mathit{ep}\ A\ D \vee \mathit{el}$
$B\ C$
    and $\bigwedge A\ B.\ \mathit{ep}\ A\ A \Longrightarrow \mathit{ep}\ B\ B \Longrightarrow \exists C.\ \mathit{pl}\ A\ C \wedge \mathit{pl}\ B\ C$
  shows *goal* **using** *assms*

by *coherent*

lemma *p2p1*:
assumes $col\ a\ b\ c\ l \wedge col\ d\ e\ f\ m$
and $col\ b\ f\ g\ n \wedge col\ c\ e\ g\ o$
and $col\ b\ d\ h\ p \wedge col\ a\ e\ h\ q$
and $col\ c\ d\ i\ r \wedge col\ a\ f\ i\ s$
and $pl\ a\ m \Longrightarrow goal$
and $pl\ b\ m \Longrightarrow goal$
and $pl\ c\ m \Longrightarrow goal$
and $pl\ d\ l \Longrightarrow goal$
and $pl\ e\ l \Longrightarrow goal$
and $pl\ f\ l \Longrightarrow goal$
and $\bigwedge A.\ pl\ g\ A \Longrightarrow pl\ h\ A \Longrightarrow pl\ i\ A \Longrightarrow goal$
and $\bigwedge A\ B\ C\ D.\ col\ A\ B\ C\ D \Longrightarrow pl\ A\ D$
and $\bigwedge A\ B\ C\ D.\ col\ A\ B\ C\ D \Longrightarrow pl\ B\ D$
and $\bigwedge A\ B\ C\ D.\ col\ A\ B\ C\ D \Longrightarrow pl\ C\ D$
and $\bigwedge A\ B.\ pl\ A\ B \Longrightarrow ep\ A\ A$
and $\bigwedge A\ B.\ ep\ A\ B \Longrightarrow ep\ B\ A$
and $\bigwedge A\ B\ C.\ ep\ A\ B \Longrightarrow ep\ B\ C \Longrightarrow ep\ A\ C$
and $\bigwedge A\ B.\ pl\ A\ B \Longrightarrow el\ B\ B$
and $\bigwedge A\ B.\ el\ A\ B \Longrightarrow el\ B\ A$
and $\bigwedge A\ B\ C.\ el\ A\ B \Longrightarrow el\ B\ C \Longrightarrow el\ A\ C$
and $\bigwedge A\ B\ C.\ ep\ A\ B \Longrightarrow pl\ B\ C \Longrightarrow pl\ A\ C$
and $\bigwedge A\ B\ C.\ pl\ A\ B \Longrightarrow el\ B\ C \Longrightarrow pl\ A\ C$
and $\bigwedge A\ B\ C\ D\ E\ F\ G\ H\ I\ J\ K\ L\ M\ N\ O\ P\ Q.$
$col\ A\ B\ C\ J \Longrightarrow col\ D\ E\ F\ K \Longrightarrow col\ B\ F\ G\ L \Longrightarrow col\ C\ E\ G\ M \Longrightarrow$
$col\ B\ D\ H\ N \Longrightarrow col\ A\ E\ H\ O \Longrightarrow col\ C\ D\ I\ P \Longrightarrow col\ A\ F\ I\ Q \Longrightarrow$
$(\exists\ R.\ col\ G\ H\ I\ R) \vee el\ L\ M \vee el\ N\ O \vee el\ P\ Q$
and $\bigwedge A\ B\ C\ D.\ pl\ C\ A \Longrightarrow pl\ C\ B \Longrightarrow pl\ D\ A \Longrightarrow pl\ D\ B \Longrightarrow ep\ C\ D \vee el\ A\ B$
and $\bigwedge A\ B\ C.\ ep\ A\ A \Longrightarrow ep\ B\ B \Longrightarrow \exists\, C.\ pl\ A\ C \wedge pl\ B\ C$
shows *goal* **using** *assms*
by *coherent*

5.2 Preservation of the Diamond Property under reflexive closure

lemma *diamond*:
assumes *reflexive_rewrite a b reflexive_rewrite a c*
and $\bigwedge A.$ *reflexive_rewrite b A* $\Longrightarrow$ *reflexive_rewrite c A* $\Longrightarrow$ *goal*
and $\bigwedge A.$ *equalish A A*
and $\bigwedge A\ B.$ *equalish A B* $\Longrightarrow$ *equalish B A*
and $\bigwedge A\ B\ C.$ *equalish A B* $\Longrightarrow$ *reflexive_rewrite B C* $\Longrightarrow$ *reflexive_rewrite A C*
and $\bigwedge A\ B.$ *equalish A B* $\Longrightarrow$ *reflexive_rewrite A B*
and $\bigwedge A\ B.$ *rewrite A B* $\Longrightarrow$ *reflexive_rewrite A B*
and $\bigwedge A\ B.$ *reflexive_rewrite A B* $\Longrightarrow$ *equalish A B* $\vee$ *rewrite A B*
and $\bigwedge A\ B\ C.$ *rewrite A B* $\Longrightarrow$ *rewrite A C* $\Longrightarrow \exists\, D.$ *rewrite B D* $\wedge$ *rewrite C*

D
  shows *goal* **using** *assms*
  by *coherent*

end

6 Some Isar command definitions

theory *Commands*
imports *Main*
keywords
  print_test :: *diag* **and**
  global_test :: *thy_decl* **and**
  local_test :: *thy_decl*
begin

6.1 Diagnostic command: no state change

```
ML ‹
  Outer_Syntax.command command_keyword‹print_test› print term test
    (Parse.term >> (fn s => Toplevel.keep (fn st =>
      let
        val ctxt = Toplevel.context_of st;
        val t = Syntax.read_term ctxt s;
        val ctxt' = Proof_Context.augment t ctxt;
      in Pretty.writeln (Syntax.pretty_term ctxt' t) end)));
›
```

print_test x
print_test $\lambda x.\ x = a$

6.2 Old-style global theory declaration

```
ML ‹
  Outer_Syntax.command command_keyword‹global_test› test constant decla-
ration
    (Parse.binding >> (fn b => Toplevel.theory (fn thy =>
      let
        val thy' = Sign.add_consts [(b, typ ‹'a›, NoSyn)] thy;
      in thy' end)));
›
```

global_test a
global_test b
print_test a

6.3 Local theory specification

```
ML ‹
```

Outer_Syntax.local_theory ***command_keyword*** ‹*local_test*› *test local definition*
(*Parse.binding* −− (***keyword*** ‹=› |−− *Parse.term*) >> (*fn* (*b*, *s*) => *fn lthy* =>
let
val t = *Syntax.read_term lthy s*;
val (*def*, *lthy'*) = *Local_Theory.define* ((*b*, *NoSyn*), ((*Thm.def_binding b*, []), *t*)) *lthy*;
in lthy' end));
›

local_test *true* = *True*
print_test *true*
thm *true_def*

local_test *identity* = $\lambda x.\ x$
print_test *identity x*
thm *identity_def*

context fixes *x y* :: *nat*
begin

local_test *test* = $x + y$
print_test *test*
thm *test_def*

end

print_test *test 0 1*
thm *test_def*

end

7 The Drinker's Principle

theory *Drinker*
imports *Main*
begin

Here is another example of classical reasoning: the Drinker's Principle says that for some person, if he is drunk, everybody else is drunk!

We first prove a classical part of de-Morgan's law.

lemma *de_Morgan*:
assumes $\neg\ (\forall x.\ P\ x)$
shows $\exists x.\ \neg\ P\ x$
proof (*rule classical*)
assume $\nexists x.\ \neg\ P\ x$
have $\forall x.\ P\ x$
proof

```
    fix x show P x
    proof (rule classical)
      assume ¬ P x
      then have ∃ x. ¬ P x ..
      with ‹∄ x. ¬ P x› show ?thesis by contradiction
    qed
  qed
  with ‹¬ (∀ x. P x)› show ?thesis by contradiction
qed

theorem Drinker's_Principle: ∃ x. drunk x ⟶ (∀ x. drunk x)
proof cases
  assume ∀ x. drunk x
  then have drunk a ⟶ (∀ x. drunk x) for a ..
  then show ?thesis ..
next
  assume ¬ (∀ x. drunk x)
  then have ∃ x. ¬ drunk x by (rule de_Morgan)
  then obtain a where ¬ drunk a ..
  have drunk a ⟶ (∀ x. drunk x)
  proof
    assume drunk a
    with ‹¬ drunk a› show ∀ x. drunk x by contradiction
  qed
  then show ?thesis ..
qed

end
```

8 Examples of function definitions

```
theory Functions
imports Main HOL−Library.Monad_Syntax
begin
```

8.1 Very basic

```
fun fib :: nat ⇒ nat
where
  fib 0 = 1
| fib (Suc 0) = 1
| fib (Suc (Suc n)) = fib n + fib (Suc n)
```

Partial simp and induction rules:

```
thm fib.psimps
thm fib.pinduct
```

There is also a cases rule to distinguish cases along the definition:

```
thm fib.cases
```

Total simp and induction rules:

thm *fib.simps*
thm *fib.induct*

Elimination rules:

thm *fib.elims*

8.2 Currying

fun *add*
where
  add 0 y = y
| *add (Suc x) y = Suc (add x y)*

thm *add.simps*
thm *add.induct* — Note the curried induction predicate

8.3 Nested recursion

function *nz*
where
  nz 0 = 0
| *nz (Suc x) = nz (nz x)*
by *pat_completeness auto*

lemma *nz_is_zero*: — A lemma we need to prove termination
  assumes *trm*: *nz_dom x*
  shows *nz x = 0*
using *trm*
by *induct* (*auto simp*: *nz.psimps*)

termination *nz*
  by (*relation less_than*) (*auto simp*:*nz_is_zero*)

thm *nz.simps*
thm *nz.induct*

8.3.1 Here comes McCarthy's 91-function

function *f91* :: *nat* ⇒ *nat*
where
  f91 n = (*if 100* < *n then n* − *10 else f91* (*f91* (*n* + *11*)))
by *pat_completeness auto*

Prove a lemma before attempting a termination proof:

lemma *f91_estimate*:
  assumes *trm*: *f91_dom n*
  shows *n* < *f91 n* + *11*
using *trm* **by** *induct* (*auto simp*: *f91.psimps*)

```
termination
proof
  let ?R = measure (λx. 101 − x)
  show wf ?R ..

  fix n :: nat
  assume ¬ 100 < n  — Inner call
  then show (n + 11, n) ∈ ?R by simp

  assume inner_trm: f91_dom (n + 11)  — Outer call
  with f91_estimate have n + 11 < f91 (n + 11) + 11 .
  with ‹¬ 100 < n› show (f91 (n + 11), n) ∈ ?R by simp
qed
```

Now trivial (even though it does not belong here):

```
lemma f91 n = (if 100 < n then n − 10 else 91)
  by (induct n rule: f91.induct) auto
```

8.3.2 Here comes Takeuchi's function

```
definition tak_m1 where tak_m1 = (λ(x,y,z). if x ≤ y then 0 else 1)
definition tak_m2 where tak_m2 = (λ(x,y,z). nat (Max {x, y, z} − Min {x, y,
z}))
definition tak_m3 where tak_m3 = (λ(x,y,z). nat (x − Min {x, y, z}))

function tak :: int ⇒ int ⇒ int ⇒ int where
  tak x y z = (if x ≤ y then y else tak (tak (x−1) y z) (tak (y−1) z x) (tak (z−1)
x y))
  by auto

lemma tak_pcorrect:
  tak_dom (x, y, z) ⟹ tak x y z = (if x ≤ y then y else if y ≤ z then z else x)
  by (induction x y z rule: tak.pinduct) (auto simp: tak.psimps)

termination
  by (relation tak_m1 <*mlex*> tak_m2 <*mlex*> tak_m3 <*mlex*> {})
    (auto simp: mlex_iff wf_mlex tak_pcorrect tak_m1_def tak_m2_def tak_m3_def
min_def max_def)

theorem tak_correct: tak x y z = (if x ≤ y then y else if y ≤ z then z else x)
  by (induction x y z rule: tak.induct) auto
```

8.4 More general patterns

8.4.1 Overlapping patterns

Currently, patterns must always be compatible with each other, since no automatic splitting takes place. But the following definition of GCD is OK, although patterns overlap:

```
fun gcd2 :: nat ⇒ nat ⇒ nat
where
  gcd2 x 0 = x
| gcd2 0 y = y
| gcd2 (Suc x) (Suc y) = (if x < y then gcd2 (Suc x) (y − x)
                                    else gcd2 (x − y) (Suc y))

thm gcd2.simps
thm gcd2.induct
```

8.4.2 Guards

We can reformulate the above example using guarded patterns:

```
function gcd3 :: nat ⇒ nat ⇒ nat
where
  gcd3 x 0 = x
| gcd3 0 y = y
| gcd3 (Suc x) (Suc y) = gcd3 (Suc x) (y − x) if x < y
| gcd3 (Suc x) (Suc y) = gcd3 (x − y) (Suc y) if ¬ x < y
  apply (case_tac x, case_tac a, auto)
  apply (case_tac ba, auto)
  done
termination by lexicographic_order

thm gcd3.simps
thm gcd3.induct
```

General patterns allow even strange definitions:

```
function ev :: nat ⇒ bool
where
  ev (2 ∗ n) = True
| ev (2 ∗ n + 1) = False
proof − — completeness is more difficult here . . .
  fix P :: bool
  fix x :: nat
  assume c1: ⋀n. x = 2 ∗ n ⟹ P
    and c2: ⋀n. x = 2 ∗ n + 1 ⟹ P
  have divmod: x = 2 ∗ (x div 2) + (x mod 2) by auto
  show P
  proof (cases x mod 2 = 0)
    case True
    with divmod have x = 2 ∗ (x div 2) by simp
    with c1 show P .
  next
    case False
    then have x mod 2 = 1 by simp
    with divmod have x = 2 ∗ (x div 2) + 1 by simp
```

with *c2* **show** *P* **.**
qed
qed *presburger+* — solve compatibility with presburger
termination by *lexicographic_order*

thm *ev.simps*
thm *ev.induct*
thm *ev.cases*

8.5 Mutual Recursion

fun *evn od* :: *nat* ⇒ *bool*
where
evn 0 = *True*
| *od 0* = *False*
| *evn* (*Suc n*) = *od n*
| *od* (*Suc n*) = *evn n*

thm *evn.simps*
thm *od.simps*

thm *evn_od.induct*
thm *evn_od.termination*

thm *evn.elims*
thm *od.elims*

8.6 Definitions in local contexts

locale *my_monoid* =
fixes *opr* :: *'a* ⇒ *'a* ⇒ *'a*
and *un* :: *'a*
assumes *assoc*: *opr* (*opr x y*) *z* = *opr x* (*opr y z*)
and *lunit*: *opr un x* = *x*
and *runit*: *opr x un* = *x*
begin

fun *foldR* :: *'a list* ⇒ *'a*
where
foldR [] = *un*
| *foldR* (*x* # *xs*) = *opr x* (*foldR xs*)

fun *foldL* :: *'a list* ⇒ *'a*
where
foldL [] = *un*
| *foldL* [*x*] = *x*
| *foldL* (*x* # *y* # *ys*) = *foldL* (*opr x y* # *ys*)

thm *foldL.simps*

lemma *foldR_foldL*: *foldR xs* = *foldL xs*
  by (*induct xs rule*: *foldL.induct*) (*auto simp*:*lunit runit assoc*)

thm *foldR_foldL*

end

thm *my_monoid.foldL.simps*
thm *my_monoid.foldR_foldL*

8.7 *fun_cases*

8.7.1 Predecessor

fun *pred* :: *nat* ⇒ *nat*
where
  pred 0 = *0*
| *pred* (*Suc n*) = *n*

thm *pred.elims*

lemma
  assumes *pred x* = *y*
  obtains *x* = *0 y* = *0* | *n* **where** *x* = *Suc n y* = *n*
  by (*fact pred.elims*[*OF assms*])

If the predecessor of a number is 0, that number must be 0 or 1.

fun_cases *pred0E*[*elim*]: *pred n* = *0*

lemma *pred n* = *0* ⟹ *n* = *0* ∨ *n* = *Suc 0*
  by (*erule pred0E*) *metis*+

Other expressions on the right-hand side also work, but whether the generated rule is useful depends on how well the simplifier can simplify it. This example works well:

fun_cases *pred42E*[*elim*]: *pred n* = *42*

lemma *pred n* = *42* ⟹ *n* = *43*
  by (*erule pred42E*)

8.7.2 List to option

fun *list_to_option* :: *'a list* ⇒ *'a option*
where
  list_to_option [*x*] = *Some x*
| *list_to_option* _ = *None*

fun_cases *list_to_option_NoneE*: *list_to_option xs* = *None*
  and *list_to_option_SomeE*: *list_to_option xs* = *Some x*

lemma $list\_to\_option\ xs = Some\ y \Longrightarrow xs = [y]$
  by $(erule\ list\_to\_option\_SomeE)$

8.7.3 Boolean Functions

fun $xor :: bool \Rightarrow bool \Rightarrow bool$
where
  $xor\ False\ False = False$
$|\ xor\ True\ True = False$
$|\ xor\ \_\ \_ = True$

thm $xor.elims$

fun_cases does not only recognise function equations, but also works with functions that return a boolean, e.g.:

fun_cases xor_TrueE: $xor\ a\ b$ **and** xor_FalseE: $\neg xor\ a\ b$
print_theorems

8.7.4 Many parameters

fun $sum4 :: nat \Rightarrow nat \Rightarrow nat \Rightarrow nat \Rightarrow nat$
  where $sum4\ a\ b\ c\ d = a + b + c + d$

fun_cases $sum40E$: $sum4\ a\ b\ c\ d = 0$

lemma $sum4\ a\ b\ c\ d = 0 \Longrightarrow a = 0$
  by $(erule\ sum40E)$

8.8 Partial Function Definitions

Partial functions in the option monad:

partial_function $(option)$
  $collatz :: nat \Rightarrow nat\ list\ option$
where
  $collatz\ n =$
    $(if\ n \leq 1\ then\ Some\ [n]$
     $else\ if\ even\ n$
       $then\ do\ \{\ ns \leftarrow collatz\ (n\ div\ 2);\ Some\ (n\ \#\ ns)\ \}$
       $else\ do\ \{\ ns \leftarrow collatz\ (3 * n + 1);\ \ Some\ (n\ \#\ ns)\})$

declare $collatz.simps[code]$
value $collatz\ 23$

Tail-recursive functions:

partial_function $(tailrec)\ fixpoint :: ('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a$
where
  $fixpoint\ f\ x = (if\ f\ x = x\ then\ x\ else\ fixpoint\ f\ (f\ x))$

8.9 Regression tests

The following examples mainly serve as tests for the function package.

fun *listlen* :: *'a list* ⇒ *nat*
where
  listlen [] = *0*
| *listlen* (*x*#*xs*) = *Suc* (*listlen xs*)

8.9.1 Context recursion

fun *f* :: *nat* ⇒ *nat*
where
  zero: *f 0* = *0*
| *succ*: *f* (*Suc n*) = (*if f n* = *0 then 0 else f n*)

8.9.2 A combination of context and nested recursion

function *h* :: *nat* ⇒ *nat*
where
  h 0 = *0*
| *h* (*Suc n*) = (*if h n* = *0 then h* (*h n*) *else h n*)
by *pat_completeness auto*

8.9.3 Context, but no recursive call

fun *i* :: *nat* ⇒ *nat*
where
  i 0 = *0*
| *i* (*Suc n*) = (*if n* = *0 then 0 else i n*)

8.9.4 Tupled nested recursion

fun *fa* :: *nat* ⇒ *nat* ⇒ *nat*
where
  fa 0 y = *0*
| *fa* (*Suc n*) *y* = (*if fa n y* = *0 then 0 else fa n y*)

8.9.5 Let

fun *j* :: *nat* ⇒ *nat*
where
  j 0 = *0*
| *j* (*Suc n*) = (*let u* = *n in Suc* (*j u*))

There were some problems with fresh names ...

function *k* :: *nat* ⇒ *nat*
where
  k x = (*let a* = *x*; *b* = *x in k x*)
  by *pat_completeness auto*

```
function f2 :: (nat × nat) ⇒ (nat × nat)
where
  f2 p = (let (x,y) = p in f2 (y,x))
  by pat_completeness auto
```

8.9.6 Abbreviations

```
fun f3 :: 'a set ⇒ bool
where
  f3 x = finite x
```

8.9.7 Simple Higher-Order Recursion

```
datatype 'a tree = Leaf 'a | Branch 'a tree list

fun treemap :: ('a ⇒ 'a) ⇒ 'a tree ⇒ 'a tree
where
  treemap fn (Leaf n) = (Leaf (fn n))
| treemap fn (Branch l) = (Branch (map (treemap fn) l))

fun tinc :: nat tree ⇒ nat tree
where
  tinc (Leaf n) = Leaf (Suc n)
| tinc (Branch l) = Branch (map tinc l)

fun testcase :: 'a tree ⇒ 'a list
where
  testcase (Leaf a) = [a]
| testcase (Branch x) =
    (let xs = concat (map testcase x);
         ys = concat (map testcase x) in
     xs @ ys)
```

8.9.8 Pattern matching on records

```
record point =
  Xcoord :: int
  Ycoord :: int

function swp :: point ⇒ point
where
  swp (| Xcoord = x, Ycoord = y |) = (| Xcoord = y, Ycoord = x |)
proof −
  fix P x
  assume ⋀xa y. x = (|Xcoord = xa, Ycoord = y|) ⟹ P
  then show P by (cases x)
qed auto
termination by rule auto
```

8.9.9 The diagonal function

fun *diag* :: *bool* ⇒ *bool* ⇒ *bool* ⇒ *nat*
where
diag x True False = *1*
| *diag False y True* = *2*
| *diag True False z* = *3*
| *diag True True True* = *4*
| *diag False False False* = *5*

8.9.10 Many equations (quadratic blowup)

datatype *DT* =
A | *B* | *C* | *D* | *E* | *F* | *G* | *H* | *I* | *J* | *K* | *L* | *M* | *N* | *P*
| *Q* | *R* | *S* | *T* | *U* | *V*

fun *big* :: *DT* ⇒ *nat*
where
big A = *0*
| *big B* = *0*
| *big C* = *0*
| *big D* = *0*
| *big E* = *0*
| *big F* = *0*
| *big G* = *0*
| *big H* = *0*
| *big I* = *0*
| *big J* = *0*
| *big K* = *0*
| *big L* = *0*
| *big M* = *0*
| *big N* = *0*
| *big P* = *0*
| *big Q* = *0*
| *big R* = *0*
| *big S* = *0*
| *big T* = *0*
| *big U* = *0*
| *big V* = *0*

8.9.11 Automatic pattern splitting

fun *f4* :: *nat* ⇒ *nat* ⇒ *bool*
where
f4 0 0 = *True*
| *f4* _ _ = *False*

8.9.12 Polymorphic partial-function

partial_function (*option*) *f5* :: *'a list* ⇒ *'a option*

where
  f5 x = f5 x

end

9 Groebner Basis Examples

theory *Groebner_Examples*
imports *Main*
begin

9.1 Basic examples

lemma
  fixes *x* :: *int*
  shows *x ^ 3 = x ^ 3*
  apply (*tactic* ‹*ALLGOALS* (*CONVERSION*
   (*Conv.arg_conv* (*Conv.arg1_conv* (*Semiring_Normalizer.semiring_normalize_conv*
context)))))›)
  by (*rule refl*)

lemma
  fixes *x* :: *int*
  shows (*x* − (−*2*))^*5* = *x* ^ *5* + (*10* ∗ *x* ^ *4* + (*40* ∗ *x* ^ *3* + (*80* ∗ x^2 + (*80*
∗ *x* + *32*))))
  apply (*tactic* ‹*ALLGOALS* (*CONVERSION*
   (*Conv.arg_conv* (*Conv.arg1_conv* (*Semiring_Normalizer.semiring_normalize_conv*
context)))))›)
  by (*rule refl*)

schematic_goal
  fixes *x* :: *int*
  shows (*x* − (−*2*))^*5* ∗ (*y* − *78*) ^ *8* = *?X*
  apply (*tactic* ‹*ALLGOALS* (*CONVERSION*
   (*Conv.arg_conv* (*Conv.arg1_conv* (*Semiring_Normalizer.semiring_normalize_conv*
context)))))›)
  by (*rule refl*)

lemma ((−*3*) ^(*Suc* (*Suc* (*Suc* *0*)))) == (*X*::′*a*::{*comm_ring_1*})
  apply (*simp only*: *power_Suc power_0*)
  apply (*simp only*: *semiring_norm*)
  oops

lemma ((*x*::*int*) + *y*)^*3* − *1* = (*x* − *z*)^*2* − *10* ⟹ *x* = *z* + *3* ⟹ *x* = − *y*
  by *algebra*

lemma (*4*::*nat*) + *4* = *3* + *5*
  by *algebra*

lemma $(4::int) + 0 = 4$
apply *algebra?*
by *simp*

lemma
assumes $a * x^2 + b * x + c = (0::int)$ **and** $d * x^2 + e * x + f = 0$
shows $d^2 * c^2 - 2 * d * c * a * f + a^2 * f^2 - e * d * b * c - e * b * a * f + a * e^2 * c + f * d * b^2 = 0$
using *assms* **by** *algebra*

lemma $(x::int)\hat{}3 - x\hat{}2 - 5*x - 3 = 0 \longleftrightarrow (x = 3 \vee x = -1)$
by *algebra*

theorem $x* (x^2 - x - 5) - 3 = (0::int) \longleftrightarrow (x = 3 \vee x = -1)$
by *algebra*

lemma
fixes $x::'a::idom$
shows $x^2*y = x^2 \ \& \ x*y^2 = y^2 \longleftrightarrow x = 1 \ \& \ y = 1 \mid x = 0 \ \& \ y = 0$
by *algebra*

9.2 Lemmas for Lagrange's theorem

definition
$sq :: 'a::times => 'a$ **where**
$sq\ x == x*x$

lemma
fixes $x1 :: 'a::\{idom\}$
shows
$(sq\ x1 + sq\ x2 + sq\ x3 + sq\ x4) * (sq\ y1 + sq\ y2 + sq\ y3 + sq\ y4) =$
$sq\ (x1*y1 - x2*y2 - x3*y3 - x4*y4) +$
$sq\ (x1*y2 + x2*y1 + x3*y4 - x4*y3) +$
$sq\ (x1*y3 - x2*y4 + x3*y1 + x4*y2) +$
$sq\ (x1*y4 + x2*y3 - x3*y2 + x4*y1)$
by (*algebra add*: *sq_def*)

lemma
fixes $p1 :: 'a::\{idom\}$
shows
$(sq\ p1 + sq\ q1 + sq\ r1 + sq\ s1 + sq\ t1 + sq\ u1 + sq\ v1 + sq\ w1) *$
$(sq\ p2 + sq\ q2 + sq\ r2 + sq\ s2 + sq\ t2 + sq\ u2 + sq\ v2 + sq\ w2)$
$= sq\ (p1*p2 - q1*q2 - r1*r2 - s1*s2 - t1*t2 - u1*u2 - v1*v2 - w1*w2)$
$+$
$sq\ (p1*q2 + q1*p2 + r1*s2 - s1*r2 + t1*u2 - u1*t2 - v1*w2 + w1*v2)$
$+$
$sq\ (p1*r2 - q1*s2 + r1*p2 + s1*q2 + t1*v2 + u1*w2 - v1*t2 - w1*u2)$
$+$
$sq\ (p1*s2 + q1*r2 - r1*q2 + s1*p2 + t1*w2 - u1*v2 + v1*u2 - w1*t2)$

+
    $sq\ (p1{*}t2 - q1{*}u2 - r1{*}v2 - s1{*}w2 + t1{*}p2 + u1{*}q2 + v1{*}r2 + w1{*}s2)$
+
    $sq\ (p1{*}u2 + q1{*}t2 - r1{*}w2 + s1{*}v2 - t1{*}q2 + u1{*}p2 - v1{*}s2 + w1{*}r2)$
+
    $sq\ (p1{*}v2 + q1{*}w2 + r1{*}t2 - s1{*}u2 - t1{*}r2 + u1{*}s2 + v1{*}p2 - w1{*}q2)$
+
    $sq\ (p1{*}w2 - q1{*}v2 + r1{*}u2 + s1{*}t2 - t1{*}s2 - u1{*}r2 + v1{*}q2 + w1{*}p2)$
  by (*algebra add*: *sq_def*)

9.3 Colinearity is invariant by rotation

type_synonym *point* = *int* × *int*

definition *collinear* ::*point* ⇒ *point* ⇒ *point* ⇒ *bool* **where**
  collinear ≡ λ(*Ax*,*Ay*) (*Bx*,*By*) (*Cx*,*Cy*).
    ((*Ax* − *Bx*) ∗ (*By* − *Cy*) = (*Ay* − *By*) ∗ (*Bx* − *Cx*))

lemma *collinear_inv_rotation*:
  assumes *collinear* (*Ax*, *Ay*) (*Bx*, *By*) (*Cx*, *Cy*) **and** $c^2 + s^2 = 1$
  shows *collinear* (*Ax* ∗ *c* − *Ay* ∗ *s*, *Ay* ∗ *c* + *Ax* ∗ *s*)
    (*Bx* ∗ *c* − *By* ∗ *s*, *By* ∗ *c* + *Bx* ∗ *s*) (*Cx* ∗ *c* − *Cy* ∗ *s*, *Cy* ∗ *c* + *Cx* ∗ *s*)
  using *assms*
  by (*algebra add*: *collinear_def split_def fst_conv snd_conv*)

lemma ∃ (*d*::*int*). *a*∗*y* − *a*∗*x* = *n*∗*d* ⟹ ∃ *u v*. *a*∗*u* + *n*∗*v* = *1* ⟹ ∃ *e*. *y* − *x* = *n*∗*e*
  by *algebra*

end

10 Example of Declaring an Oracle

theory *Iff_Oracle*
  imports *Main*
begin

10.1 Oracle declaration

This oracle makes tautologies of the form $P = (P = (P = P))$. The length is specified by an integer, which is checked to be even and positive.

oracle *iff_oracle* = ‹
  let
    fun mk_iff 1 = *Var* ((*P*, *0*), ***typ*** ‹*bool*›)
      | *mk_iff n* = *HOLogic.mk_eq* (*Var* ((*P*, *0*), ***typ*** ‹*bool*›), *mk_iff* (*n* − *1*));
  in
    fn (*thy*, *n*) =>
      if n > *0 andalso n mod 2* = *0*

then Thm.global_cterm_of thy (HOLogic.mk_Trueprop (mk_iff n))
else raise Fail (iff_oracle: ^ string_of_int n)
end
›

10.2 Oracle as low-level rule

ML ‹*iff_oracle* (***theory***, *2*)›
ML ‹*iff_oracle* (***theory***, *10*)›

ML ‹
assert (*map* (*#1 o #1*) (*Thm_Deps.all_oracles* [*iff_oracle* (***theory***, *10*)]) = [***oracle_name*** ‹*iff_oracle*›]);
›

These oracle calls had better fail.

ML ‹
(*iff_oracle* (***theory***, *5*); *error Bad oracle*)
handle Fail _ => writeln Oracle failed, as expected
›

ML ‹
(*iff_oracle* (***theory***, *1*); *error Bad oracle*)
handle Fail _ => writeln Oracle failed, as expected
›

10.3 Oracle as proof method

method_setup *iff* =
‹*Scan.lift Parse.nat >> (fn n => fn ctxt =>*
SIMPLE_METHOD
(HEADGOAL (resolve_tac ctxt [iff_oracle (Proof_Context.theory_of ctxt, n)])
handle Fail _ => no_tac))›

lemma $A \longleftrightarrow A$
by (*iff 2*)

lemma $A \longleftrightarrow A \longleftrightarrow A \longleftrightarrow A \longleftrightarrow A \longleftrightarrow A \longleftrightarrow A \longleftrightarrow A \longleftrightarrow A \longleftrightarrow A$
by (*iff 10*)

lemma $A \longleftrightarrow A \longleftrightarrow A \longleftrightarrow A \longleftrightarrow A$
apply (*iff 5*)?
oops

lemma A
apply (*iff 1*)?
oops

end

11 Examples of automatically derived induction rules

theory *Induction_Schema*
imports *Main*
begin

11.1 Some simple induction principles on nat

lemma *nat_standard_induct*:
  $[\![P\ 0;\ \bigwedge n.\ P\ n \Longrightarrow P\ (Suc\ n)]\!] \Longrightarrow P\ x$
by *induction_schema* (*pat_completeness*, *lexicographic_order*)

lemma *nat_induct2*:
  $[\![\ P\ 0;\ P\ (Suc\ 0);\ \bigwedge k.\ P\ k ==> P\ (Suc\ k) ==> P\ (Suc\ (Suc\ k))\ ]\!]$
  $\Longrightarrow P\ n$
by *induction_schema* (*pat_completeness*, *lexicographic_order*)

lemma *minus_one_induct*:
  $[\![\bigwedge n{::}nat.\ (n \neq 0 \Longrightarrow P\ (n - 1)) \Longrightarrow P\ n]\!] \Longrightarrow P\ x$
by *induction_schema* (*pat_completeness*, *lexicographic_order*)

theorem *diff_induct*:
  $(!!x.\ P\ x\ 0) ==> (!!y.\ P\ 0\ (Suc\ y)) ==>$
    $(!!x\ y.\ P\ x\ y ==> P\ (Suc\ x)\ (Suc\ y)) ==> P\ m\ n$
by *induction_schema* (*pat_completeness*, *lexicographic_order*)

lemma *list_induct2'*:
  $[\![\ P\ []\ [];$
  $\bigwedge x\ xs.\ P\ (x\#xs)\ [];$
  $\bigwedge y\ ys.\ P\ []\ (y\#ys);$
   $\bigwedge x\ xs\ y\ ys.\ P\ xs\ ys\ \Longrightarrow P\ (x\#xs)\ (y\#ys)\ ]\!]$
 $\Longrightarrow P\ xs\ ys$
by *induction_schema* (*pat_completeness*, *lexicographic_order*)

theorem *even_odd_induct*:
  assumes $R\ 0$
  assumes $Q\ 0$
  assumes $\bigwedge n.\ Q\ n \Longrightarrow R\ (Suc\ n)$
  assumes $\bigwedge n.\ R\ n \Longrightarrow Q\ (Suc\ n)$
  shows $R\ n\ Q\ n$
  using *assms*
by *induction_schema* (*pat_completeness*+, *lexicographic_order*)

end

12 Textbook-style reasoning: the Knaster-Tarski Theorem

theory *Knaster_Tarski*
  imports *Main*
begin

unbundle *lattice_syntax*

12.1 Prose version

According to the textbook [1, pages 93–94], the Knaster-Tarski fixpoint theorem is as follows.[1]

The Knaster-Tarski Fixpoint Theorem. Let L be a complete lattice and f: $L \to L$ an order-preserving map. Then $\bigsqcap\{x \in L \mid f(x) \leq x\}$ is a fixpoint of f.

Proof. Let $H = \{x \in L \mid f(x) \leq x\}$ and $a = \bigsqcap H$. For all $x \in H$ we have $a \leq x$, so $f(a) \leq f(x) \leq x$. Thus $f(a)$ is a lower bound of H, whence $f(a) \leq a$. We now use this inequality to prove the reverse one (!) and thereby complete the proof that a is a fixpoint. Since f is order-preserving, $f(f(a)) \leq f(a)$. This says $f(a) \in H$, so $a \leq f(a)$.

12.2 Formal versions

The Isar proof below closely follows the original presentation. Virtually all of the prose narration has been rephrased in terms of formal Isar language elements. Just as many textbook-style proofs, there is a strong bias towards forward proof, and several bends in the course of reasoning.

```
theorem Knaster_Tarski:
  fixes f :: 'a::complete_lattice ⇒ 'a
  assumes mono f
  shows ∃ a. f a = a
proof
  let ?H = {u. f u ≤ u}
  let ?a = ⨅ ?H
  show f ?a = ?a
  proof −
    {
      fix x
      assume x ∈ ?H
      then have ?a ≤ x by (rule Inf_lower)
      with ‹mono f› have f ?a ≤ f x ..
      also from ‹x ∈ ?H› have . . . ≤ x ..
      finally have f ?a ≤ x .
```

[1] We have dualized the argument, and tuned the notation a little bit.

```
    }
    then have f ?a ≤ ?a by (rule Inf_greatest)
    {
      also presume . . . ≤ f ?a
      finally (order_antisym) show ?thesis .
    }
    from ‹mono f› and ‹f ?a ≤ ?a› have f (f ?a) ≤ f ?a ..
    then have f ?a ∈ ?H ..
    then show ?a ≤ f ?a by (rule Inf_lower)
  qed
qed
```

Above we have used several advanced Isar language elements, such as explicit block structure and weak assumptions. Thus we have mimicked the particular way of reasoning of the original text.

In the subsequent version the order of reasoning is changed to achieve structured top-down decomposition of the problem at the outer level, while only the inner steps of reasoning are done in a forward manner. We are certainly more at ease here, requiring only the most basic features of the Isar language.

```
theorem Knaster_Tarski':
  fixes f :: 'a::complete_lattice ⇒ 'a
  assumes mono f
  shows ∃ a. f a = a
proof
  let ?H = {u. f u ≤ u}
  let ?a = ⨅ ?H
  show f ?a = ?a
  proof (rule order_antisym)
    show f ?a ≤ ?a
    proof (rule Inf_greatest)
      fix x
      assume x ∈ ?H
      then have ?a ≤ x by (rule Inf_lower)
      with ‹mono f› have f ?a ≤ f x ..
      also from ‹x ∈ ?H› have . . . ≤ x ..
      finally show f ?a ≤ x .
    qed
    show ?a ≤ f ?a
    proof (rule Inf_lower)
      from ‹mono f› and ‹f ?a ≤ ?a› have f (f ?a) ≤ f ?a ..
      then show f ?a ∈ ?H ..
    qed
  qed
qed

end
```

13 Isabelle/ML basics

theory *ML*
  imports *Main*
begin

13.1 ML expressions

The Isabelle command **ML** allows to embed Isabelle/ML source into the formal text. It is type-checked, compiled, and run within that environment.

Note that side-effects should be avoided, unless the intention is to change global parameters of the run-time environment (rare).

ML top-level bindings are managed within the theory context.

```
ML ‹1 + 1›

ML ‹val a = 1›
ML ‹val b = 1›
ML ‹val c = a + b›
```

13.2 Antiquotations

There are some language extensions (via antiquotations), as explained in the "Isabelle/Isar implementation manual", chapter 0.

```
ML ‹length []›
ML ‹assert (length [] = 0)›
```

Formal entities from the surrounding context may be referenced as follows:

term *1 + 1* — term within theory source

```
ML ‹term‹1 + 1›  (* term as symbolic ML datatype value *)›

ML ‹term‹1 + (1::int)››


ML ‹
  (* formal source with position information *)
  val s = ‹1 + 1›;

  (* read term via old−style string interface *)
  val t = Syntax.read_term context (Syntax.implode_input s);
›
```

13.3 Recursive ML evaluation

```
ML ‹
  ML ‹ML ‹val a = @{thm refl}››;
  ML ‹val b = @{thm sym}›;
```

```
  val c = @{thm trans}
  val thms = [a, b, c];
›
```

13.4 IDE support

ML embedded into the Isabelle environment is connected to the Prover IDE. Poly/ML provides:

- precise positions for warnings / errors
- markup for defining positions of identifiers
- markup for inferred types of sub-expressions
- pretty-printing of ML values with markup
- completion of ML names
- source-level debugger

```
ML ‹fn i => fn list => length list + i›
```

13.5 Example: factorial and ackermann function in Isabelle/ML

```
ML ‹
  fun factorial 0 = 1
    | factorial n = n * factorial (n - 1)
›

ML ‹factorial 42›
ML ‹factorial 10000 div factorial 9999›
```

See http://mathworld.wolfram.com/AckermannFunction.html.

```
ML ‹
  fun ackermann 0 n = n + 1
    | ackermann m 0 = ackermann (m - 1) 1
    | ackermann m n = ackermann (m - 1) (ackermann m (n - 1))
›

ML ‹timeit (fn () => ackermann 3 10)›
```

13.6 Parallel Isabelle/ML

Future.fork/join/cancel manage parallel evaluation.

Note that within Isabelle theory documents, the top-level command boundary may not be transgressed without special precautions. This is normally managed by the system when performing parallel proof checking.

ML ‹
  val x = Future.fork (fn () => ackermann 3 10);
  val y = Future.fork (fn () => ackermann 3 10);
  val z = Future.join x + Future.join y
›

The `Par_List` module provides high-level combinators for parallel list operations.

ML ‹*timeit (fn () => map (fn n => ackermann 3 n) (1 upto 10))*›
ML ‹*timeit (fn () => Par_List.map (fn n => ackermann 3 n) (1 upto 10))*›

13.7 Function specifications in Isabelle/HOL

fun *factorial* :: *nat* ⇒ *nat*
where
  factorial 0 = 1
| *factorial (Suc n) = Suc n ∗ factorial n*

term *factorial 4* — symbolic term
value *factorial 4* — evaluation via ML code generation in the background

declare [[*ML_source_trace*]]
ML ‹***term***‹*factorial 4*›› — symbolic term in ML
ML ‹@{*code factorial*}› — ML code from function specification

fun *ackermann* :: *nat* ⇒ *nat* ⇒ *nat*
where
  ackermann 0 n = n + 1
| *ackermann (Suc m) 0 = ackermann m 1*
| *ackermann (Suc m) (Suc n) = ackermann m (ackermann (Suc m) n)*

value *ackermann 3 5*

end

14 Peirce's Law

theory *Peirce*
  imports *Main*
begin

We consider Peirce's Law: $((A \longrightarrow B) \longrightarrow A) \longrightarrow A$. This is an inherently non-intuitionistic statement, so its proof will certainly involve some form of classical contradiction.

The first proof is again a well-balanced combination of plain backward and forward reasoning. The actual classical step is where the negated goal may be introduced as additional assumption. This eventually leads to a contradiction.[2]

theorem $((A \longrightarrow B) \longrightarrow A) \longrightarrow A$
proof
  assume $(A \longrightarrow B) \longrightarrow A$
  show A
  proof (*rule classical*)
    assume $\neg\ A$
    have $A \longrightarrow B$
    proof
      assume A
      with ‹$\neg\ A$› **show** B **by** *contradiction*
    qed
    with ‹$(A \longrightarrow B) \longrightarrow A$› **show** A **..**
  qed
qed

In the subsequent version the reasoning is rearranged by means of "weak assumptions" (as introduced by **presume**). Before assuming the negated goal $\neg\ A$, its intended consequence $A \longrightarrow B$ is put into place in order to solve the main problem. Nevertheless, we do not get anything for free, but have to establish $A \longrightarrow B$ later on. The overall effect is that of a logical *cut*.

Technically speaking, whenever some goal is solved by **show** in the context of weak assumptions then the latter give rise to new subgoals, which may be established separately. In contrast, strong assumptions (as introduced by **assume**) are solved immediately.

theorem $((A \longrightarrow B) \longrightarrow A) \longrightarrow A$
proof
  assume $(A \longrightarrow B) \longrightarrow A$
  show A
  proof (*rule classical*)
    presume $A \longrightarrow B$
    with ‹$(A \longrightarrow B) \longrightarrow A$› **show** A **..**
  next
    assume $\neg\ A$
    show $A \longrightarrow B$
    proof
      assume A
      with ‹$\neg\ A$› **show** B **by** *contradiction*
    qed
  qed
qed

Note that the goals stemming from weak assumptions may be even left until qed time, where they get eventually solved "by assumption" as well. In

[2]The rule involved there is negation elimination; it holds in intuitionistic logic as well.

that case there is really no fundamental difference between the two kinds of assumptions, apart from the order of reducing the individual parts of the proof configuration.

Nevertheless, the "strong" mode of plain assumptions is quite important in practice to achieve robustness of proof text interpretation. By forcing both the conclusion *and* the assumptions to unify with the pending goal to be solved, goal selection becomes quite deterministic. For example, decomposition with rules of the "case-analysis" type usually gives rise to several goals that only differ in there local contexts. With strong assumptions these may be still solved in any order in a predictable way, while weak ones would quickly lead to great confusion, eventually demanding even some backtracking.

end

15 Using extensible records in HOL – points and coloured points

theory *Records*
 imports *Main*
begin

15.1 Points

record *point* =
 xpos :: *nat*
 ypos :: *nat*

Apart many other things, above record declaration produces the following theorems:

thm *point.simps*
thm *point.iffs*
thm *point.defs*

The set of theorems *point.simps* is added automatically to the standard simpset, *point.iffs* is added to the Classical Reasoner and Simplifier context.

Record declarations define new types and type abbreviations:

$point = (\!|xpos :: nat,\ ypos :: nat|\!) = ()\ point\_ext\_type$
$'a\ point\_scheme = (\!|xpos :: nat,\ ypos :: nat,\ ... :: 'a|\!)\ = 'a\ point\_ext\_type$

consts *foo2* :: $(\!|xpos :: nat,\ ypos :: nat|\!)$
consts *foo4* :: $'a \Rightarrow (\!|xpos :: nat,\ ypos :: nat,\ \ldots :: 'a|\!)$

15.1.1 Introducing concrete records and record schemes

definition *foo1* :: *point*
  where *foo1* = (| *xpos* = *1*, *ypos* = *0* |)

definition *foo3* :: *'a* ⇒ *'a point_scheme*
  where *foo3 ext* = (| *xpos* = *1*, *ypos* = *0*, ... = *ext* |)

15.1.2 Record selection and record update

definition *getX* :: *'a point_scheme* ⇒ *nat*
  where *getX r* = *xpos r*

definition *setX* :: *'a point_scheme* ⇒ *nat* ⇒ *'a point_scheme*
  where *setX r n* = *r* (| *xpos* := *n* |)

15.1.3 Some lemmas about records

Basic simplifications.

lemma *point.make n p* = (| *xpos* = *n*, *ypos* = *p* |)
  by (*simp only*: *point.make_def*)

lemma *xpos* (| *xpos* = *m*, *ypos* = *n*, ... = *p* |) = *m*
  by *simp*

lemma (| *xpos* = *m*, *ypos* = *n*, ... = *p* |)(| *xpos*:= *0* |) = (| *xpos* = *0*, *ypos* = *n*, ... = *p* |)
  by *simp*

Equality of records.

lemma *n* = *n'* ⟹ *p* = *p'* ⟹ (| *xpos* = *n*, *ypos* = *p* |) = (| *xpos* = *n'*, *ypos* = *p'* |)
  — introduction of concrete record equality
  by *simp*

lemma (| *xpos* = *n*, *ypos* = *p* |) = (| *xpos* = *n'*, *ypos* = *p'* |) ⟹ *n* = *n'*
  — elimination of concrete record equality
  by *simp*

lemma *r*(| *xpos* := *n* |)(| *ypos* := *m* |) = *r*(| *ypos* := *m* |)(| *xpos* := *n* |)
  — introduction of abstract record equality
  by *simp*

lemma *r*(| *xpos* := *n* |) = *r*(| *xpos* := *n'* |) **if** *n* = *n'*
  — elimination of abstract record equality (manual proof)
proof −
  let *?lhs* = *?rhs* = *?thesis*
  from *that* **have** *xpos ?lhs* = *xpos ?rhs* **by** *simp*
  then show *?thesis* **by** *simp*
qed

Surjective pairing

lemma $r = (\!|xpos = xpos\ r,\ ypos = ypos\ r|\!)$
  by *simp*

lemma $r = (\!|xpos = xpos\ r,\ ypos = ypos\ r,\ \ldots = point.more\ r|\!)$
  by *simp*

Representation of records by cases or (degenerate) induction.

lemma $r(\!|xpos := n|\!)(\!|ypos := m|\!) = r(\!|ypos := m|\!)(\!|xpos := n|\!)$
proof $(cases\ r)$
  fix *xpos ypos more*
  assume $r = (\!|xpos = xpos,\ ypos = ypos,\ \ldots = more|\!)$
  then show *?thesis* **by** *simp*
qed

lemma $r(\!|xpos := n|\!)(\!|ypos := m|\!) = r(\!|ypos := m|\!)(\!|xpos := n|\!)$
proof $(induct\ r)$
  fix *xpos ypos more*
  show $(\!|xpos = xpos,\ ypos = ypos,\ \ldots = more|\!)(\!|xpos := n,\ ypos := m|\!) =$
      $(\!|xpos = xpos,\ ypos = ypos,\ \ldots = more|\!)(\!|ypos := m,\ xpos := n|\!)$
    by *simp*
qed

lemma $r(\!|xpos := n|\!)(\!|xpos := m|\!) = r(\!|xpos := m|\!)$
proof $(cases\ r)$
  fix *xpos ypos more*
  assume $r = (\!|xpos = xpos,\ ypos = ypos,\ \ldots = more|\!)$
  then show *?thesis* **by** *simp*
qed

lemma $r(\!|xpos := n|\!)(\!|xpos := m|\!) = r(\!|xpos := m|\!)$
proof $(cases\ r)$
  case *fields*
  then show *?thesis* **by** *simp*
qed

lemma $r(\!|xpos := n|\!)(\!|xpos := m|\!) = r(\!|xpos := m|\!)$
  by $(cases\ r)$ *simp*

Concrete records are type instances of record schemes.

definition $foo5 :: nat$
  where $foo5 = getX\ (\!|xpos = 1,\ ypos = 0|\!)$

Manipulating the "..." (more) part.

definition $incX :: 'a\ point\_scheme \Rightarrow 'a\ point\_scheme$
  where $incX\ r = (\!|xpos = xpos\ r + 1,\ ypos = ypos\ r,\ \ldots = point.more\ r|\!)$

lemma $incX\ r = setX\ r\ (Suc\ (getX\ r))$
by $(simp\ add{:}\ getX\_def\ setX\_def\ incX\_def)$

An alternative definition.

definition $incX' :: 'a\ point\_scheme \Rightarrow 'a\ point\_scheme$
where $incX'\ r = r(\!|xpos := xpos\ r + 1|\!)$

15.2 Coloured points: record extension

datatype $colour = Red \mid Green \mid Blue$

record $cpoint = point +$
$colour :: colour$

The record declaration defines a new type constructor and abbreviations:

$cpoint = (\!|xpos :: nat,\ ypos :: nat,\ colour :: colour|\!) =$
$()\ cpoint\_ext\_type\ point\_ext\_type$
$'a\ cpoint\_scheme = (\!|xpos :: nat,\ ypos :: nat,\ colour :: colour,\ \ldots :: 'a|\!) =$
$'a\ cpoint\_ext\_type\ point\_ext\_type$

consts $foo6 :: cpoint$
consts $foo7 :: (\!|xpos :: nat,\ ypos :: nat,\ colour :: colour|\!)$
consts $foo8 :: 'a\ cpoint\_scheme$
consts $foo9 :: (\!|xpos :: nat,\ ypos :: nat,\ colour :: colour,\ \ldots :: 'a|\!)$

Functions on *point* schemes work for *cpoints* as well.

definition $foo10 :: nat$
where $foo10 = getX\ (\!|xpos = 2,\ ypos = 0,\ colour = Blue|\!)$

15.2.1 Non-coercive structural subtyping

Term *foo11* has type *cpoint*, not type *point* — Great!

definition $foo11 :: cpoint$
where $foo11 = setX\ (\!|xpos = 2,\ ypos = 0,\ colour = Blue|\!)\ 0$

15.3 Other features

Field names contribute to record identity.

record $point' =$
$xpos' :: nat$
$ypos' :: nat$

May not apply *getX* to $(\!|xpos' = 2,\ ypos' = 0|\!)$ — type error.

Polymorphic records.

record $'a\ point'' = point +$

content :: *'a*

type_synonym *cpoint'' = colour point''*

Updating a record field with an identical value is simplified.

lemma $r(\!|xpos := xpos\ r|\!) = r$
by *simp*

Only the most recent update to a component survives simplification.

lemma $r(\!|xpos := x,\ ypos := y,\ xpos := x'|\!) = r(\!|ypos := y,\ xpos := x'|\!)$
by *simp*

In some cases its convenient to automatically split (quantified) records. For this purpose there is the simproc `Record.split_simproc` and the tactic `Record.split_simp_tac`. The simplification procedure only splits the records, whereas the tactic also simplifies the resulting goal with the standard record simplification rules. A (generalized) predicate on the record is passed as parameter that decides whether or how 'deep' to split the record. It can peek on the subterm starting at the quantified occurrence of the record (including the quantifier). The value `0` indicates no split, a value greater `0` splits up to the given bound of record extension and finally the value `~1` completely splits the record. `Record.split_simp_tac` additionally takes a list of equations for simplification and can also split fixed record variables.

lemma $(\forall r.\ P\ (xpos\ r)) \longrightarrow (\forall x.\ P\ x)$
apply (*tactic* ‹*simp_tac* (*put_simpset HOL_basic_ss* ***context***
addsimprocs [*Record.split_simproc* (*K* ~*1*)]) *1*›)
apply *simp*
done

lemma $(\forall r.\ P\ (xpos\ r)) \longrightarrow (\forall x.\ P\ x)$
apply (*tactic* ‹*Record.split_simp_tac* ***context*** [] (*K* ~*1*) *1*›)
apply *simp*
done

lemma $(\exists r.\ P\ (xpos\ r)) \longrightarrow (\exists x.\ P\ x)$
apply (*tactic* ‹*simp_tac* (*put_simpset HOL_basic_ss* ***context***
addsimprocs [*Record.split_simproc* (*K* ~*1*)]) *1*›)
apply *simp*
done

lemma $(\exists r.\ P\ (xpos\ r)) \longrightarrow (\exists x.\ P\ x)$
apply (*tactic* ‹*Record.split_simp_tac* ***context*** [] (*K* ~*1*) *1*›)
apply *simp*
done

lemma $\bigwedge r.\ P\ (xpos\ r) \Longrightarrow (\exists x.\ P\ x)$
apply (*tactic* ‹*simp_tac* (*put_simpset HOL_basic_ss* ***context***

addsimprocs [*Record.split_simproc* (K ~*1*)]) *1*›)
apply *auto*
done

lemma $\bigwedge r.\ P\ (xpos\ r) \Longrightarrow (\exists\, x.\ P\ x)$
apply (*tactic* ‹*Record.split_simp_tac* ***context*** [] (K ~*1*) *1*›)
apply *auto*
done

lemma $P\ (xpos\ r) \Longrightarrow (\exists\, x.\ P\ x)$
apply (*tactic* ‹*Record.split_simp_tac* ***context*** [] (K ~*1*) *1*›)
apply *auto*
done

notepad
begin
have $\exists\, x.\ P\ x$
if $P\ (xpos\ r)$ **for** $P\ r$
apply (*insert that*)
apply (*tactic* ‹*Record.split_simp_tac* ***context*** [] (K ~*1*) *1*›)
apply *auto*
done
end

The effect of simproc `Record.ex_sel_eq_simproc` is illustrated by the following lemma.

lemma $\exists\, r.\ xpos\ r = x$
by (*tactic* ‹*simp_tac* (*put_simpset HOL_basic_ss* ***context***
addsimprocs [*Record.ex_sel_eq_simproc*]) *1*›)

15.4 A more complex record expression

record ('*a*, '*b*, '*c*) *bar* = *bar1* :: '*a*
bar2 :: '*b*
bar3 :: '*c*
bar21 :: '*b* × '*a*
bar32 :: '*c* × '*b*
bar31 :: '*c* × '*a*

print_record ('*a*,'*b*,'*c*) *bar*

15.5 Some code generation

export_code *foo1 foo3 foo5 foo10* **checking** *SML*

Code generation can also be switched off, for instance for very large records:

declare [[*record_codegen* = *false*]]

record *not_so_large_record* =

```
  bar520 :: nat
  bar521 :: nat × nat

declare [[record_codegen = true]]

end
```

16 Finite sequences

```
theory Seq
  imports Main
begin

datatype 'a seq = Empty | Seq 'a 'a seq

fun conc :: 'a seq ⇒ 'a seq ⇒ 'a seq
where
  conc Empty ys = ys
| conc (Seq x xs) ys = Seq x (conc xs ys)

fun reverse :: 'a seq ⇒ 'a seq
where
  reverse Empty = Empty
| reverse (Seq x xs) = conc (reverse xs) (Seq x Empty)

lemma conc_empty: conc xs Empty = xs
  by (induct xs) simp_all

lemma conc_assoc: conc (conc xs ys) zs = conc xs (conc ys zs)
  by (induct xs) simp_all

lemma reverse_conc: reverse (conc xs ys) = conc (reverse ys) (reverse xs)
  by (induct xs) (simp_all add: conc_empty conc_assoc)

lemma reverse_reverse: reverse (reverse xs) = xs
  by (induct xs) (simp_all add: reverse_conc)

end
```

17 Square roots of primes are irrational

```
theory Sqrt
  imports Complex_Main HOL−Computational_Algebra.Primes
begin
```

The square root of any prime number (including 2) is irrational.

```
theorem sqrt_prime_irrational:
  fixes p :: nat
```

```
  assumes prime p
  shows sqrt p ∉ ℚ
proof
  from ‹prime p› have p: p > 1 by (rule prime_gt_1_nat)
  assume sqrt p ∈ ℚ
  then obtain m n :: nat
    where n: n ≠ 0
      and sqrt_rat: |sqrt p| = m / n
      and coprime m n by (rule Rats_abs_nat_div_natE)
  have eq: m^2 = p * n^2
  proof −
    from n and sqrt_rat have m = |sqrt p| * n by simp
    then have m^2 = (sqrt p)^2 * n^2 by (simp add: power_mult_distrib)
    also have (sqrt p)^2 = p by simp
    also have ... * n^2 = p * n^2 by simp
    finally show ?thesis by linarith
  qed
  have p dvd m ∧ p dvd n
  proof
    from eq have p dvd m^2 ..
    with ‹prime p› show p dvd m by (rule prime_dvd_power)
    then obtain k where m = p * k ..
    with eq have p * n^2 = p^2 * k^2 by algebra
    with p have n^2 = p * k^2 by (simp add: power2_eq_square)
    then have p dvd n^2 ..
    with ‹prime p› show p dvd n by (rule prime_dvd_power)
  qed
  then have p dvd gcd m n by simp
  with ‹coprime m n› have p = 1 by simp
  with p show False by simp
qed

corollary sqrt_2_not_rat: sqrt 2 ∉ ℚ
  using sqrt_prime_irrational [of 2] by simp
```

Here is an alternative version of the main proof, using mostly linear forward-reasoning. While this results in less top-down structure, it is probably closer to proofs seen in mathematics.

```
theorem
  fixes p :: nat
  assumes prime p
  shows sqrt p ∉ ℚ
proof
  from ‹prime p› have p: p > 1 by (rule prime_gt_1_nat)
  assume sqrt p ∈ ℚ
  then obtain m n :: nat
    where n: n ≠ 0
      and sqrt_rat: |sqrt p| = m / n
      and coprime m n by (rule Rats_abs_nat_div_natE)
```

from n **and** *sqrt_rat* **have** $m = |sqrt\ p| * n$ **by** *simp*
then have $m^2 = (sqrt\ p)^2 * n^2$ **by** (*auto simp add*: *power2_eq_square*)
also have $(sqrt\ p)^2 = p$ **by** *simp*
also have $\ldots * n^2 = p * n^2$ **by** *simp*
finally have *eq*: $m^2 = p * n^2$ **by** *linarith*
then have $p\ dvd\ m^2$ **..**
with ‹*prime p*› **have** *dvd_m*: $p\ dvd\ m$ **by** (*rule prime_dvd_power*)
then obtain k **where** $m = p * k$ **..**
with *eq* **have** $p * n^2 = p^2 * k^2$ **by** *algebra*
with p **have** $n^2 = p * k^2$ **by** (*simp add*: *power2_eq_square*)
then have $p\ dvd\ n^2$ **..**
with ‹*prime p*› **have** $p\ dvd\ n$ **by** (*rule prime_dvd_power*)
with *dvd_m* **have** $p\ dvd\ gcd\ m\ n$ **by** (*rule gcd_greatest*)
with ‹*coprime m n*› **have** $p = 1$ **by** *simp*
with p **show** *False* **by** *simp*
qed

Another old chestnut, which is a consequence of the irrationality of *sqrt 2*.

lemma $\exists\, a\ b$::*real*. $a \notin \mathbb{Q} \wedge b \notin \mathbb{Q} \wedge a\ powr\ b \in \mathbb{Q}$ (**is** $\exists\, a\ b.\ ?P\ a\ b$)
proof (*cases sqrt 2 powr sqrt 2* $\in \mathbb{Q}$)
case *True*
with *sqrt_2_not_rat* **have** *?P* (*sqrt 2*) (*sqrt 2*) **by** *simp*
then show *?thesis* **by** *blast*
next
case *False*
with *sqrt_2_not_rat powr_powr* **have** *?P* (*sqrt 2 powr sqrt 2*) (*sqrt 2*) **by** *simp*
then show *?thesis* **by** *blast*
qed

end

References

[1] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 1990.

[2] L. C. Paulson. *Isabelle: A Generic Theorem Prover*. Springer, 1994. LNCS 828.