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

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

adhoc_overloading
  vars term_vars

value [*nbe*] *vars* (*Fun* $''f''$ [*Var 0*, *Var 1*])

fun *rule_vars* :: ($'a$, $'b$) *term* $\times$ ($'a$, $'b$) *term* $\Rightarrow$ $'b$ *set* **where**
  rule_vars (*l*, *r*) = *vars l* $\cup$ *vars r*

adhoc_overloading
  vars rule_vars

value [*nbe*] *vars* (*Var 1*, *Var 0*)

definition *trs_vars* :: (($'a$, $'b$) *term* $\times$ ($'a$, $'b$) *term*) *set* $\Rightarrow$ $'b$ *set* **where**
  trs_vars R = $\bigcup$(*rule_vars* ‘ *R*)

adhoc_overloading
  vars trs_vars

value [*nbe*] *vars* {(*Var 1*, *Var 0*)}

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

value *vars* (*R* :: (($'a$, $'b$) *term* $\times$ ($'a$, $'b$) *term*) *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*
⟨*proof*⟩

First we need some auxiliary lemmas.

lemma *permsI* [*Pure.intro*]:
assumes *bij f* **and** *MOST x. f x = x*
shows $f \in perms$
⟨*proof*⟩

lemma *perms_imp_bij*:
$f \in perms \Longrightarrow bij\ f$
⟨*proof*⟩

lemma *perms_imp_MOST_eq*:
$f \in perms \Longrightarrow MOST\ x.\ f\ x = x$
⟨*proof*⟩

lemma *id_perms* [*simp*]:
$id \in perms$
$(\lambda x.\ x) \in perms$
⟨*proof*⟩

lemma *perms_comp* [*simp*]:
assumes f: $f \in perms$ **and** g: $g \in perms$
shows $(f \circ g) \in perms$
⟨*proof*⟩

lemma *perms_inv*:
assumes f: $f \in perms$
shows $inv\ f \in perms$
⟨*proof*⟩

lemma *bij_Rep_perm*: *bij* (*Rep_perm p*)
⟨*proof*⟩

instantiation *perm* :: (*type*) *group_add*
begin

definition $0 = Abs\_perm\ id$
definition $-\ p = Abs\_perm\ (inv\ (Rep\_perm\ p))$
definition $p + q = Abs\_perm\ (Rep\_perm\ p \circ Rep\_perm\ q)$
definition $(p1::'a\ perm) - p2 = p1 + -\ p2$

lemma Rep_perm_0: $Rep\_perm\ 0 = id$
⟨*proof*⟩

lemma Rep_perm_add:
$Rep\_perm\ (p1 + p2) = Rep\_perm\ p1 \circ Rep\_perm\ p2$
⟨*proof*⟩

lemma Rep_perm_uminus:
$Rep\_perm\ (-\ p) = inv\ (Rep\_perm\ p)$
⟨*proof*⟩

instance
⟨*proof*⟩

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 $PERMUTE :: 'a\ perm \Rightarrow 'b \Rightarrow 'b$ (**infixr** $\cdot$ *75*)

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

locale *permute* =
fixes $permute :: 'a\ perm \Rightarrow 'b \Rightarrow 'b$
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 $permute\_atom :: 'a\ perm \Rightarrow 'a \Rightarrow 'a$ **where**
$permute\_atom\ p\ a = (Rep\_perm\ p)\ a$

adhoc_overloading
  PERMUTE permute_atom

interpretation *atom_permute*: *permute permute_atom*
  ⟨*proof*⟩

Permuting permutations.

definition *permute_perm* :: $'a$ *perm* $\Rightarrow$ $'a$ *perm* $\Rightarrow$ $'a$ *perm* **where**
  permute_perm $p\ q = p + q - p$

adhoc_overloading
  PERMUTE permute_perm

interpretation *perm_permute*: *permute permute_perm*
  ⟨*proof*⟩

Permuting functions.

locale *fun_permute* =
  dom: *permute perm1* + *ran*: *permute perm2*
  for *perm1* :: $'a$ *perm* $\Rightarrow$ $'b \Rightarrow 'b$
  and *perm2* :: $'a$ *perm* $\Rightarrow$ $'c \Rightarrow 'c$
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*
  ⟨*proof*⟩

lemma (*Abs_perm id* :: *nat perm*) $\cdot$ *Suc 0* = *Suc 0*
  ⟨*proof*⟩

interpretation *atom_fun_permute*: *fun_permute permute_atom permute_atom*
  ⟨*proof*⟩

adhoc_overloading
  PERMUTE atom_fun_permute.permute_fun

lemma (*Abs_perm id* :: $'a$ *perm*) $\cdot$ *id* = *id*
  ⟨*proof*⟩

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] ⇒ 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 ⇒ 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
  ⟨proof⟩
```

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) ⟹ ackloop_dom (n # 0 # l)
ackloop_dom (Suc 0 # m # l) ⟹ ackloop_dom (0 # Suc m # l)
ackloop_dom (n # Suc m # m # l) ⟹ 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*⟩

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*)
⟨*proof*⟩

lemma *ackloop_dom* (*ack m n* # *l*) $\Longrightarrow$ *ackloop_dom* (*n* # *m* # *l*)
⟨*proof*⟩

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*
⟨*proof*⟩

termination *ackloop*
⟨*proof*⟩

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*
⟨*proof*⟩

theorem *ack*: *ack m n* = *ackloop* [*n*,*m*]
⟨*proof*⟩

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*: $\nexists f :: 'a \Rightarrow 'a\ set.\ \forall A.\ \exists x.\ A = f\ x$
$\langle proof \rangle$

4.2 Automated proofs

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

theorem $\nexists f :: 'a \Rightarrow 'a\ set.\ \forall A.\ \exists x.\ f\ x = A$
$\langle proof \rangle$

theorem $\nexists f :: 'a \Rightarrow 'a\ set.\ \forall A.\ \exists x.\ f\ x = A$
$\langle proof \rangle$

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*
$\langle proof \rangle$

theorem *Cantor'*: $\nexists f :: 'a \Rightarrow 'a \Rightarrow bool.\ \forall A.\ \exists x.\ A = f\ x$
$\langle proof \rangle$

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 $\exists S.\ S \notin range\ (f :: 'a \Rightarrow 'a\ set)$
⟨*proof*⟩

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 $\exists S.\ S \notin range\ (f :: 'a \Rightarrow 'a\ set)$
  ⟨*proof*⟩

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

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* $\langle proof \rangle$

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* $\langle proof \rangle$

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

$\langle ML \rangle$

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

6.2 Old-style global theory declaration

$\langle ML \rangle$

global_test *a*
global_test *b*
print_test *a*

6.3 Local theory specification

$\langle ML \rangle$

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 $\exists\, x.\ \neg\ P\ x$
$\langle proof \rangle$

theorem *Drinker's_Principle*: $\exists\, x.\ drunk\ x \longrightarrow (\forall\, x.\ drunk\ x)$
$\langle proof \rangle$

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*)
⟨*proof*⟩

lemma *nz_is_zero*: — A lemma we need to prove termination
  assumes *trm*: *nz_dom x*
  shows *nz x* = *0*
⟨*proof*⟩

termination *nz*
  ⟨*proof*⟩

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*)))
⟨*proof*⟩

Prove a lemma before attempting a termination proof:

lemma *f91_estimate*:
  assumes *trm*: *f91_dom n*
  shows *n* < *f91 n* + *11*
⟨*proof*⟩

termination
⟨*proof*⟩

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

lemma *f91 n* = (*if 100* < *n then n* − *10 else 91*)
  ⟨*proof*⟩

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*))
  ⟨*proof*⟩

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*)
  ⟨*proof*⟩

termination
  ⟨*proof*⟩

theorem *tak_correct*: *tak x y z* = (*if x* ≤ *y then y else if y* ≤ *z then z else x*)
  ⟨*proof*⟩

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 $gcd2 :: nat \Rightarrow nat \Rightarrow nat$
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 $gcd3 :: nat \Rightarrow nat \Rightarrow nat$
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** $\neg\ x < y$
$\langle proof \rangle$
termination $\langle proof \rangle$

thm *gcd3.simps*
thm *gcd3.induct*

General patterns allow even strange definitions:

function $ev :: nat \Rightarrow bool$
where
$ev\ (2 * n) = True$
$|\ ev\ (2 * n + 1) = False$
$\langle proof \rangle$
termination $\langle proof \rangle$

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

8.5 Mutual Recursion

fun $evn\ od :: nat \Rightarrow bool$
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* :: $'a \Rightarrow 'a \Rightarrow 'a$
    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* :: $'a\ list \Rightarrow 'a$
where
  $foldR\ [] = un$
| $foldR\ (x\ \#\ xs) = opr\ x\ (foldR\ xs)$

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

thm *foldL.simps*

lemma *foldR_foldL*: $foldR\ xs = foldL\ xs$
  ⟨*proof*⟩

thm *foldR_foldL*

end

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

8.7 *fun_cases*

8.7.1 Predecessor

fun *pred* :: $nat \Rightarrow nat$
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*
  ⟨*proof*⟩

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* $\Longrightarrow$ *n = 0* $\vee$ *n = Suc 0*
  ⟨*proof*⟩

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* $\Longrightarrow$ *n = 43*
  ⟨*proof*⟩

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*]
  ⟨*proof*⟩

8.7.3 Boolean Functions

fun *xor* :: *bool* ⇒ *bool* ⇒ *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*: ¬*xor a b*
print_theorems

8.7.4 Many parameters

fun *sum4* :: *nat* ⇒ *nat* ⇒ *nat* ⇒ *nat* ⇒ *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* ⟹ *a* = *0*
  ⟨*proof*⟩

8.8 Partial Function Definitions

Partial functions in the option monad:

```
partial_function (option)
  collatz :: nat ⇒ nat list option
where
  collatz n =
    (if n ≤ 1 then Some [n]
     else if even n
       then do { ns ← collatz (n div 2); Some (n # ns) }
       else do { ns ← collatz (3 ∗ n + 1);  Some (n # ns)})

declare collatz.simps[code]
value collatz 23
```

Tail-recursive functions:

```
partial_function (tailrec) fixpoint :: ('a ⇒ 'a) ⇒ 'a ⇒ '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
```

$h\ 0 = 0$
$|\ h\ (Suc\ n) = (if\ h\ n = 0\ then\ h\ (h\ n)\ else\ h\ n)$
$\langle proof \rangle$

8.9.3 Context, but no recursive call

fun $i :: nat \Rightarrow nat$
where
$i\ 0 = 0$
$|\ i\ (Suc\ n) = (if\ n = 0\ then\ 0\ else\ i\ n)$

8.9.4 Tupled nested recursion

fun $fa :: nat \Rightarrow nat \Rightarrow nat$
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 $j :: nat \Rightarrow nat$
where
$j\ 0 = 0$
$|\ j\ (Suc\ n) = (let\ u = n\ in\ Suc\ (j\ u))$

There were some problems with fresh names ...

function $k :: nat \Rightarrow nat$
where
$k\ x = (let\ a = x;\ b = x\ in\ k\ x)$
$\langle proof \rangle$

function $f2 :: (nat \times nat) \Rightarrow (nat \times nat)$
where
$f2\ p = (let\ (x,y) = p\ in\ f2\ (y,x))$
$\langle proof \rangle$

8.9.6 Abbreviations

fun $f3 :: 'a\ set \Rightarrow bool$
where
$f3\ x = finite\ x$

8.9.7 Simple Higher-Order Recursion

datatype $'a\ tree = Leaf\ 'a\ |\ Branch\ 'a\ tree\ list$

fun $treemap :: ('a \Rightarrow 'a) \Rightarrow 'a\ tree \Rightarrow 'a\ tree$
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*⟩
termination ⟨*proof*⟩

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*
  ⟨*proof*⟩

lemma
  fixes *x* :: *int*
  shows $(x - (-2))\hat{}5 = x \hat{}\ 5 + (10 * x \hat{}\ 4 + (40 * x \hat{}\ 3 + (80 * x^2 + (80 * x + 32))))$

⟨*proof*⟩

schematic_goal
  fixes x :: *int*
  shows $(x - (-2))\hat{}5 * (y - 78) \hat{} 8 = ?X$
  ⟨*proof*⟩

lemma $((-3) \hat{}(Suc\ (Suc\ (Suc\ 0)))) == (X::'a::\{comm\_ring\_1\})$
  ⟨*proof*⟩

lemma $((x::int) + y)\hat{}3 - 1 = (x - z)\hat{}2 - 10 \Longrightarrow x = z + 3 \Longrightarrow x = - y$
  ⟨*proof*⟩

lemma $(4::nat) + 4 = 3 + 5$
  ⟨*proof*⟩

lemma $(4::int) + 0 = 4$
  ⟨*proof*⟩

lemma
  assumes $a * x^2 + b * x + c = (0::int)$ **and** $d * x^2 + e * x + f = 0$
  shows $d^2 * c^2 - 2 * d * c * a * f + a^2 * f^2 - e * d * b * c - e * b * a * f +$
    $a * e^2 * c + f * d * b^2 = 0$
  ⟨*proof*⟩

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

theorem $x* (x^2 - x - 5) - 3 = (0::int) \longleftrightarrow (x = 3 \vee x = -1)$
  ⟨*proof*⟩

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

9.2 Lemmas for Lagrange's theorem

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

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

$sq\ (x1{*}y4\ +\ x2{*}y3\ -\ x3{*}y2\ +\ x4{*}y1)$
⟨*proof*⟩

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

9.3 Colinearity is invariant by rotation

type_synonym $point = int \times int$

definition $collinear\ ::point \Rightarrow point \Rightarrow point \Rightarrow bool$ **where**
$collinear \equiv \lambda(Ax,Ay)\ (Bx,By)\ (Cx,Cy).$
$((Ax - Bx) * (By - Cy) = (Ay - By) * (Bx - Cx))$

lemma *collinear_inv_rotation*:
assumes $collinear\ (Ax,\ Ay)\ (Bx,\ By)\ (Cx,\ Cy)$ **and** $c^2 + s^2 = 1$
shows $collinear\ (Ax * c - Ay * s,\ Ay * c + Ax * s)$
$(Bx * c - By * s,\ By * c + Bx * s)\ (Cx * c - Cy * s,\ Cy * c + Cx * s)$
⟨*proof*⟩

lemma $\exists\,(d::int).\ a{*}y - a{*}x = n{*}d \Longrightarrow \exists\,u\ v.\ a{*}u + n{*}v = 1 \Longrightarrow \exists\,e.\ y - x = n{*}e$
⟨*proof*⟩

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.

$\langle ML \rangle$

10.2 Oracle as low-level rule

$\langle ML \rangle$

These oracle calls had better fail.

$\langle ML \rangle$

10.3 Oracle as proof method

$\langle ML \rangle$

lemma $A \longleftrightarrow A$
  $\langle proof \rangle$

lemma $A \longleftrightarrow A \longleftrightarrow A \longleftrightarrow A \longleftrightarrow A \longleftrightarrow A \longleftrightarrow A \longleftrightarrow A \longleftrightarrow A \longleftrightarrow A$
  $\langle proof \rangle$

lemma $A \longleftrightarrow A \longleftrightarrow A \longleftrightarrow A \longleftrightarrow A$
  $\langle proof \rangle$

lemma A
  $\langle proof \rangle$

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*:
  $[\![P\ 0;\ \bigwedge n.\ P\ n \Longrightarrow P\ (Suc\ n)]\!] \Longrightarrow P\ x$
$\langle proof \rangle$

lemma *nat_induct2*:

$\llbracket$ *P 0*; *P* (*Suc 0*); $\bigwedge$*k*. *P k* ==> *P* (*Suc k*) ==> *P* (*Suc* (*Suc k*)) $\rrbracket$
$\Longrightarrow$ *P n*
⟨*proof*⟩

lemma *minus_one_induct*:
$\llbracket\bigwedge$*n*::*nat*. (*n* ≠ *0* $\Longrightarrow$ *P* (*n* − *1*)) $\Longrightarrow$ *P n*$\rrbracket$ $\Longrightarrow$ *P x*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *list_induct2'*:
$\llbracket$ *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*) $\rrbracket$
$\Longrightarrow$ *P xs ys*
⟨*proof*⟩

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*
⟨*proof*⟩

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 $\bigsqcap\{x \in L \mid f(x) \le x\}$ is a

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

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* $\Rightarrow$ $'a$
  assumes *mono f*
  shows $\exists\, a.\ f\ a = a$
⟨*proof*⟩

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* $\Rightarrow$ $'a$
  assumes *mono f*
  shows $\exists\, a.\ f\ a = a$
⟨*proof*⟩

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*⟩

13.2 Antiquotations

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

⟨*ML*⟩

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

term *1 + 1* — term within theory source

⟨*ML*⟩

13.3 Recursive ML evaluation

⟨*ML*⟩

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*⟩

13.5 Example: factorial and ackermann function in Isabelle/ML

⟨*ML*⟩

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

⟨*ML*⟩

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*⟩

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

⟨*ML*⟩

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*⟩

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 $((A \longrightarrow B) \longrightarrow A) \longrightarrow A$
$\langle$*proof*$\rangle$

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 $((A \longrightarrow B) \longrightarrow A) \longrightarrow A$
$\langle$*proof*$\rangle$

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 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

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

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* ⇒ (|*xpos* :: *nat*, *ypos* :: *nat*, . . . :: *'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*|)
  ⟨*proof*⟩

lemma *xpos* (|*xpos* = *m*, *ypos* = *n*, . . . = *p*|) = *m*
  ⟨*proof*⟩

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

$\langle proof \rangle$

Equality of records.

lemma $n = n' \Longrightarrow p = p' \Longrightarrow (|xpos = n, ypos = p|) = (|xpos = n', ypos = p'|)$
— introduction of concrete record equality
$\langle proof \rangle$

lemma $(|xpos = n, ypos = p|) = (|xpos = n', ypos = p'|) \Longrightarrow n = n'$
— elimination of concrete record equality
$\langle proof \rangle$

lemma $r(|xpos := n|)(|ypos := m|) = r(|ypos := m|)(|xpos := n|)$
— introduction of abstract record equality
$\langle proof \rangle$

lemma $r(|xpos := n|) = r(|xpos := n'|)$ **if** $n = n'$
— elimination of abstract record equality (manual proof)
$\langle proof \rangle$

Surjective pairing

lemma $r = (|xpos = xpos\ r, ypos = ypos\ r|)$
$\langle proof \rangle$

lemma $r = (|xpos = xpos\ r, ypos = ypos\ r, \ldots = point.more\ r|)$
$\langle proof \rangle$

Representation of records by cases or (degenerate) induction.

lemma $r(|xpos := n|)(|ypos := m|) = r(|ypos := m|)(|xpos := n|)$
$\langle proof \rangle$

lemma $r(|xpos := n|)(|ypos := m|) = r(|ypos := m|)(|xpos := n|)$
$\langle proof \rangle$

lemma $r(|xpos := n|)(|xpos := m|) = r(|xpos := m|)$
$\langle proof \rangle$

lemma $r(|xpos := n|)(|xpos := m|) = r(|xpos := m|)$
$\langle proof \rangle$

lemma $r(|xpos := n|)(|xpos := m|) = r(|xpos := m|)$
$\langle proof \rangle$

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* ⇒ *'a point_scheme*
where *incX r* = (|*xpos* = *xpos r* + *1*, *ypos* = *ypos r*, ... = *point.more r*|)

lemma *incX r* = *setX r* (*Suc* (*getX r*))
⟨*proof*⟩

An alternative definition.

definition *incX'* :: *'a point_scheme* ⇒ *'a point_scheme*
where *incX' r* = *r*(|*xpos* := *xpos r* + *1*|)

15.2 Coloured points: record extension

datatype *colour* = *Red* | *Green* | *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*, ... :: *'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*, ... :: *'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 $r(\!|xpos := xpos\ r|\!) = r$
  $\langle proof \rangle$

Only the most recent update to a component survives simplification.

lemma $r(\!|xpos := x,\ ypos := y,\ xpos := x'|\!) = r(\!|ypos := y,\ xpos := x'|\!)$
  $\langle proof \rangle$

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 $(\forall r.\ P\ (xpos\ r)) \longrightarrow (\forall x.\ P\ x)$
  $\langle proof \rangle$

lemma $(\forall r.\ P\ (xpos\ r)) \longrightarrow (\forall x.\ P\ x)$
  $\langle proof \rangle$

lemma $(\exists r.\ P\ (xpos\ r)) \longrightarrow (\exists x.\ P\ x)$
  $\langle proof \rangle$

lemma $(\exists r.\ P\ (xpos\ r)) \longrightarrow (\exists x.\ P\ x)$
  $\langle proof \rangle$

lemma $\bigwedge r.\ P\ (xpos\ r) \Longrightarrow (\exists x.\ P\ x)$
  $\langle proof \rangle$

lemma $\bigwedge r.\ P\ (xpos\ r) \Longrightarrow (\exists x.\ P\ x)$
  $\langle proof \rangle$

lemma $P\ (xpos\ r) \Longrightarrow (\exists x.\ P\ x)$
  $\langle proof \rangle$

notepad

begin
⟨*proof*⟩
end

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

lemma $\exists\, r.\ xpos\ r = x$
⟨*proof*⟩

15.4 A more complex record expression

record $('a, 'b, 'c)\ bar = bar1 :: 'a$
$bar2 :: 'b$
$bar3 :: 'c$
$bar21 :: 'b \times 'a$
$bar32 :: 'c \times 'b$
$bar31 :: 'c \times 'a$

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* =
$bar520 :: nat$
$bar521 :: nat \times nat$

declare [[*record_codegen* = *true*]]

end

16 Finite sequences

theory *Seq*
imports *Main*
begin

datatype $'a\ seq = Empty \mid Seq\ 'a\ 'a\ seq$

fun $conc :: 'a\ seq \Rightarrow 'a\ seq \Rightarrow 'a\ seq$
where
$conc\ Empty\ ys = ys$
$\mid conc\ (Seq\ x\ xs)\ ys = Seq\ x\ (conc\ xs\ ys)$

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*
  ⟨*proof*⟩

lemma *conc_assoc*: *conc* (*conc xs ys*) *zs* = *conc xs* (*conc ys zs*)
  ⟨*proof*⟩

lemma *reverse_conc*: *reverse* (*conc xs ys*) = *conc* (*reverse ys*) (*reverse xs*)
  ⟨*proof*⟩

lemma *reverse_reverse*: *reverse* (*reverse xs*) = *xs*
  ⟨*proof*⟩

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*⟩

corollary *sqrt_2_not_rat*: *sqrt 2* ∉ ℚ
  ⟨*proof*⟩

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*⟩

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

lemma ∃ *a b*::*real*. *a* ∉ ℚ ∧ *b* ∉ ℚ ∧ *a powr b* ∈ ℚ (**is** ∃ *a b*. *?P a b*)
⟨*proof*⟩

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.