

Notable Examples in Isabelle/HOL

March 13, 2025

Contents

1	Ad Hoc Overloading	3
1.1	Plain Ad Hoc Overloading	4
1.2	Adhoc Overloading inside Locales	5
2	Permutation Types	6
3	A Tail-Recursive, Stack-Based Ackermann's Function	8
3.1	Example of proving termination by reasoning about the domain	8
3.2	Example of proving termination using a multiset ordering . .	10
4	Cantor's Theorem	11
4.1	Mathematical statement and proof	11
4.2	Automated proofs	11
4.3	Elementary version in higher-order predicate logic	11
4.4	Classic Isabelle/HOL example	12
5	Coherent Logic Problems	12
5.1	Equivalence of two versions of Pappus' Axiom	12
5.2	Preservation of the Diamond Property under reflexive closure	14
6	Some Isar command definitions	14
6.1	Diagnostic command: no state change	14
6.2	Old-style global theory declaration	14
6.3	Local theory specification	15
7	The Drinker's Principle	15
8	Examples of function definitions	16
8.1	Very basic	16
8.2	Currying	16
8.3	Nested recursion	16
8.3.1	Here comes McCarthy's 91-function	17

8.3.2	Here comes Takeuchi's function	17
8.4	More general patterns	18
8.4.1	Overlapping patterns	18
8.4.2	Guards	18
8.5	Mutual Recursion	18
8.6	Definitions in local contexts	19
8.7	<i>fun_cases</i>	20
8.7.1	Predecessor	20
8.7.2	List to option	20
8.7.3	Boolean Functions	20
8.7.4	Many parameters	21
8.8	Partial Function Definitions	21
8.9	Regression tests	21
8.9.1	Context recursion	22
8.9.2	A combination of context and nested recursion	22
8.9.3	Context, but no recursive call	22
8.9.4	Tupled nested recursion	22
8.9.5	Let	22
8.9.6	Abbreviations	22
8.9.7	Simple Higher-Order Recursion	23
8.9.8	Pattern matching on records	23
8.9.9	The diagonal function	23
8.9.10	Many equations (quadratic blowup)	23
8.9.11	Automatic pattern splitting	24
8.9.12	Polymorphic partial-function	24
9	Gauss Numbers: integral gauss numbers	24
9.1	Basic arithmetic	25
9.2	The Gauss Number i	26
9.3	Gauss Conjugation	28
9.4	Algebraic division	30
10	Groebner Basis Examples	31
10.1	Basic examples	31
10.2	Lemmas for Lagrange's theorem	32
10.3	Colinearity is invariant by rotation	33
11	Example of Declaring an Oracle	33
11.1	Oracle declaration	33
11.2	Oracle as low-level rule	33
11.3	Oracle as proof method	34
12	Examples of automatically derived induction rules	34
12.1	Some simple induction principles on nat	34

13 Textbook-style reasoning: the Knaster-Tarski Theorem	35
13.1 Prose version	35
13.2 Formal versions	35
14 Isabelle/ML basics	36
14.1 ML expressions	36
14.2 Antiquotations	36
14.3 Recursive ML evaluation	37
14.4 IDE support	37
14.5 Example: factorial and ackermann function in Isabelle/ML .	37
14.6 Parallel Isabelle/ML	37
14.7 Function specifications in Isabelle/HOL	38
15 Peirce’s Law	38
16 Using extensible records in HOL – points and coloured points	39
16.1 Points	39
16.1.1 Introducing concrete records and record schemes . . .	40
16.1.2 Record selection and record update	40
16.1.3 Some lemmas about records	40
16.2 Coloured points: record extension	41
16.2.1 Non-coercive structural subtyping	42
16.3 Other features	42
16.4 Simprocs for update and equality	43
16.5 A more complex record expression	45
16.6 Some code generation	45
17 The rewrite Proof Method by Example	45
18 Finite sequences	50
19 Square roots of primes are irrational	50

1 Ad Hoc Overloading

```

theory Adhoc_Overloading
imports
  Main
  HOL-Library.Infinite_Set
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  $\equiv$  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  $\equiv$  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  $\equiv$  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  $\equiv$  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
 $vars \Rightarrow term_vars$

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**
 $perms = \{f. \text{bij } f \wedge \text{finite } \{x. f\ x \neq x\}\}$

typedef *'a perm* = *perms* :: (*'a* \Rightarrow *'a*) *set*
 $\langle proof \rangle$

First we need some auxiliary lemmas.

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

lemma *perms_imp_bij*:
 $f \in perms \implies \text{bij } f$
 $\langle proof \rangle$

lemma *perms_imp_MOST_eq*:
 $f \in perms \implies \text{MOST } x. f\ x = x$
 $\langle proof \rangle$

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

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

lemma *perms_inv*:
assumes $f: f \in perms$
shows $\text{inv } f \in perms$
 $\langle proof \rangle$

```

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 ∘ 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 ∘ 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 ⇒ 'b ⇒ 'b (infixr <·> 75)

```

Then we add a locale for types '*b*' that support application 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 \Rightarrow 'a \Rightarrow 'a **where**
 permute_atom p a = (Rep_perm p) a

adhoc_overloading

PERMUTE \Rightarrow *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 \Rightarrow *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 \Rightarrow *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 \Rightarrow *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  $\Rightarrow$  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 HOL-Library.Multiset_Order HOL-Library.Product_Lexorder
begin

```

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

```

3.1 Example of proving termination by reasoning about the domain

The stack-based version 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
  <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.dominintros* [simp]

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

lemma *ackloop_dom (ack m n # l) ==> 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) ==> 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

$$\llbracket \bigwedge n\ m\ l. P\ (ack\ m\ n\ \# \ l) \implies P\ (n\ \# \ m\ \# \ l); \bigwedge m. P\ [m]; P\ [] \rrbracket \implies P\ a0$$

.

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*: $ackloop\ l = acklist\ l$
 $\langle proof \rangle$

theorem *ack*: $ack\ m\ n = ackloop\ [n, m]$
 $\langle proof \rangle$

3.2 Example of proving termination using a multiset ordering

This termination proof uses the argument from Nachum Dershowitz and Zohar Manna. Proving termination with multiset orderings. Communications of the ACM 22 (8) 1979, 465–476.

Setting up the termination proof. Note that Dershowitz had z as a global variable. The top two stack elements are treated differently from the rest.

fun *ack_mset* :: $nat\ list \Rightarrow (nat \times nat)\ multiset$
where
 $ack_mset\ [] = \{\#\}$
 $| ack_mset\ [x] = \{\#\}$
 $| ack_mset\ (z\#y\#l) = mset\ ((y, z) \# map\ (\lambda x. (Suc\ x, 0))\ l)$

lemma *case1*: $ack_mset\ (Suc\ n\ \# l) < add_mset\ (0, n)\ \{\#\ (Suc\ x, 0). x \in \# mset\ l\ \#\}$
 $\langle proof \rangle$

The stack-based version again. We need a fresh copy because we’ve already proved the termination of *ackloop*.

function *Ackloop* :: $nat\ list \Rightarrow nat$
where
 $Ackloop\ (n\ \# 0\ \# l) = Ackloop\ (Suc\ n\ \# l)$
 $| Ackloop\ (0\ \# Suc\ m\ \# l) = Ackloop\ (1\ \# m\ \# l)$
 $| Ackloop\ (Suc\ n\ \# Suc\ m\ \# l) = Ackloop\ (n\ \# Suc\ m\ \# m\ \# l)$
 $| Ackloop\ [m] = m$
 $| Ackloop\ [] = 0$
 $\langle proof \rangle$

In each recursive call, the function *ack_mset* decreases according to the multiset ordering.

termination
 $\langle proof \rangle$

Another shortcut compared with before: equivalence follows directly from this lemma.

lemma *Ackloop_ack*: $Ackloop\ (n\ \# m\ \# l) = Ackloop\ (ack\ m\ n\ \# l)$
 $\langle proof \rangle$

theorem *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*: $\nexists f :: 'a \Rightarrow 'a \text{ set}. \forall A. \exists x. A = f\ x$
<proof>

4.2 Automated proofs

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

theorem $\nexists f :: 'a \Rightarrow 'a \text{ set}. \forall A. \exists x. f\ x = A$
<proof>

theorem $\nexists f :: 'a \Rightarrow 'a \text{ set}. \forall A. \exists x. f\ x = A$
<proof>

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>

theorem *Cantor'*: $\nexists f :: 'a \Rightarrow 'a \Rightarrow \text{bool}. \forall A. \exists x. A = f\ x$
<proof>

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:

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

Viewing types as sets, $\alpha \Rightarrow \text{bool}$ 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 $\alpha \text{ set}$ and the operator $\text{range} :: (\alpha \Rightarrow \beta) \Rightarrow \beta \text{ set}$.

theorem $\exists S. S \notin \text{range } (f :: 'a \Rightarrow 'a \text{ set})$
<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 $\exists S. S \notin \text{range } (f :: 'a \Rightarrow 'a \text{ set})$
<proof>

end

5 Coherent Logic Problems

theory *Coherent*
imports *Main*
begin

5.1 Equivalence of two versions of Pappus’ Axiom

no_notation *comp* (infixl $\langle o \rangle$ 55)
unbundle *no_relcomp_syntax*

lemma *p1p2*:
assumes $\text{col } a\ b\ c\ l \wedge \text{col } d\ e\ f\ m$
and $\text{col } b\ f\ g\ n \wedge \text{col } c\ e\ g\ o$
and $\text{col } b\ d\ h\ p \wedge \text{col } a\ e\ h\ q$

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

```

and  $\bigwedge A B C D E F G H I J K L M N O P Q.$ 
   $col A B C J \implies col D E F K \implies col B F G L \implies col C E G M \implies$ 
   $col B D H N \implies col A E H O \implies col C D I P \implies col A F I Q \implies$ 
   $(\exists R. col G H I R) \vee el L M \vee el N O \vee el P Q$ 
and  $\bigwedge A B C D. pl C A \implies pl C B \implies pl D A \implies pl D B \implies ep C D \vee el$ 
 $A B$ 
and  $\bigwedge A B C. ep A A \implies ep B B \implies \exists C. pl A C \wedge pl B C$ 
shows goal <proof>

```

5.2 Preservation of the Diamond Property under reflexive closure

lemma *diamond*:

```

assumes reflexive_rewrite a b reflexive_rewrite a c
and  $\bigwedge A. reflexive\_rewrite b A \implies reflexive\_rewrite c A \implies goal$ 
and  $\bigwedge A. equalish A A$ 
and  $\bigwedge A B. equalish A B \implies equalish B A$ 
and  $\bigwedge A B C. equalish A B \implies reflexive\_rewrite B C \implies reflexive\_rewrite A$ 
 $C$ 
and  $\bigwedge A B. equalish A B \implies reflexive\_rewrite A B$ 
and  $\bigwedge A B. rewrite A B \implies reflexive\_rewrite A B$ 
and  $\bigwedge A B. reflexive\_rewrite A B \implies equalish A B \vee rewrite A B$ 
and  $\bigwedge A B C. rewrite A B \implies rewrite A C \implies \exists D. rewrite B D \wedge rewrite C$ 
 $D$ 
shows goal <proof>

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>

```

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

```

6.2 Old-style global theory declaration

<ML>

```

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  $\Rightarrow$  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)
 $\langle$ proof $\rangle$ 
```

```
lemma nz_is_zero: — A lemma we need to prove termination
assumes trm: nz_dom x
shows nz x = 0
```


$\langle proof \rangle$

termination *nz*

$\langle proof \rangle$

thm *nz.simps*

thm *nz.induct*

8.3.1 Here comes McCarthy's 91-function

function *f91* :: *nat* \Rightarrow *nat*

where

f91 *n* = (if 100 < *n* then *n* - 10 else *f91* (*f91* (*n* + 11)))

$\langle proof \rangle$

Prove a lemma before attempting a termination proof:

lemma *f91_estimate*:

assumes *trm*: *f91_dom* *n*

shows *n* < *f91* *n* + 11

$\langle proof \rangle$

termination

$\langle proof \rangle$

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

lemma *f91* *n* = (if 100 < *n* then *n* - 10 else 91)

$\langle proof \rangle$

8.3.2 Here comes Takeuchi's function

definition *tak_m1* **where** *tak_m1* = ($\lambda(x,y,z).$ if $x \leq y$ then 0 else 1)

definition *tak_m2* **where** *tak_m2* = ($\lambda(x,y,z).$ nat ($\text{Max } \{x, y, z\} - \text{Min } \{x, y, z\}$))

definition *tak_m3* **where** *tak_m3* = ($\lambda(x,y,z).$ nat ($x - \text{Min } \{x, y, z\}$))

function *tak* :: *int* \Rightarrow *int* \Rightarrow *int* \Rightarrow *int* **where**

tak *x* *y* *z* = (if $x \leq y$ then *y* else *tak* (*tak* (*x*-1) *y* *z*) (*tak* (*y*-1) *z* *x*) (*tak* (*z*-1) *x* *y*))

$\langle proof \rangle$

lemma *tak_pcorrect*:

tak_dom (*x*, *y*, *z*) \implies *tak* *x* *y* *z* = (if $x \leq y$ then *y* else if $y \leq z$ then *z* else *x*)

$\langle proof \rangle$

termination

$\langle proof \rangle$

theorem *tak_correct*: *tak* *x* *y* *z* = (if $x \leq y$ then *y* else if $y \leq z$ then *z* else *x*)

$\langle proof \rangle$

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
  ⟨proof⟩
termination ⟨proof⟩

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⟩
termination ⟨proof⟩

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  $\Rightarrow$  '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 ⇒ 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 ⇒ n = 0 ∨ 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 ⇒ 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 ⇒ 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  $f :: nat \Rightarrow nat$ 
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  $h :: nat \Rightarrow nat$ 
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 ⇒ '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⟩
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 Gauss Numbers: integral gauss numbers

```

theory Gauss_Numbers

```

```

  imports HOL-Library.Centered_Division

```

```

begin

```

```

codatatype gauss = Gauss (Re: int) (Im: int)

```


lemma *gauss_eqI* [intro?]:
 $\langle x = y \rangle$ **if** $\langle \text{Re } x = \text{Re } y \rangle \langle \text{Im } x = \text{Im } y \rangle$
 $\langle \text{proof} \rangle$

lemma *gauss_eq_iff*:
 $\langle x = y \rangle \longleftrightarrow \text{Re } x = \text{Re } y \wedge \text{Im } x = \text{Im } y$
 $\langle \text{proof} \rangle$

9.1 Basic arithmetic

instantiation *gauss* :: *comm_ring_1*
begin

primcorec *zero_gauss* :: $\langle \text{gauss} \rangle$
where
 $\langle \text{Re } 0 = 0 \rangle$
 $|\langle \text{Im } 0 = 0 \rangle$

primcorec *one_gauss* :: $\langle \text{gauss} \rangle$
where
 $\langle \text{Re } 1 = 1 \rangle$
 $|\langle \text{Im } 1 = 0 \rangle$

primcorec *plus_gauss* :: $\langle \text{gauss} \Rightarrow \text{gauss} \Rightarrow \text{gauss} \rangle$
where
 $\langle \text{Re } (x + y) = \text{Re } x + \text{Re } y \rangle$
 $|\langle \text{Im } (x + y) = \text{Im } x + \text{Im } y \rangle$

primcorec *uminus_gauss* :: $\langle \text{gauss} \Rightarrow \text{gauss} \rangle$
where
 $\langle \text{Re } (-x) = -\text{Re } x \rangle$
 $|\langle \text{Im } (-x) = -\text{Im } x \rangle$

primcorec *minus_gauss* :: $\langle \text{gauss} \Rightarrow \text{gauss} \Rightarrow \text{gauss} \rangle$
where
 $\langle \text{Re } (x - y) = \text{Re } x - \text{Re } y \rangle$
 $|\langle \text{Im } (x - y) = \text{Im } x - \text{Im } y \rangle$

primcorec *times_gauss* :: $\langle \text{gauss} \Rightarrow \text{gauss} \Rightarrow \text{gauss} \rangle$
where
 $\langle \text{Re } (x * y) = \text{Re } x * \text{Re } y - \text{Im } x * \text{Im } y \rangle$
 $|\langle \text{Im } (x * y) = \text{Re } x * \text{Im } y + \text{Im } x * \text{Re } y \rangle$

instance
 $\langle \text{proof} \rangle$

end

lemma *of_nat_gauss*:
 $\langle \text{of_nat } n = \text{Gauss } (\text{int } n) \ 0 \rangle$
 $\langle \text{proof} \rangle$

lemma *numeral_gauss*:
 $\langle \text{numeral } n = \text{Gauss } (\text{numeral } n) \ 0 \rangle$
 $\langle \text{proof} \rangle$

lemma *of_int_gauss*:
 $\langle \text{of_int } k = \text{Gauss } k \ 0 \rangle$
 $\langle \text{proof} \rangle$

lemma *conversion_simps* [simp]:
 $\langle \text{Re } (\text{numeral } m) = \text{numeral } m \rangle$
 $\langle \text{Im } (\text{numeral } m) = 0 \rangle$
 $\langle \text{Re } (\text{of_nat } n) = \text{int } n \rangle$
 $\langle \text{Im } (\text{of_nat } n) = 0 \rangle$
 $\langle \text{Re } (\text{of_int } k) = k \rangle$
 $\langle \text{Im } (\text{of_int } k) = 0 \rangle$
 $\langle \text{proof} \rangle$

lemma *gauss_eq_0*:
 $\langle z = 0 \longleftrightarrow (\text{Re } z)^2 + (\text{Im } z)^2 = 0 \rangle$
 $\langle \text{proof} \rangle$

lemma *gauss_neq_0*:
 $\langle z \neq 0 \longleftrightarrow (\text{Re } z)^2 + (\text{Im } z)^2 > 0 \rangle$
 $\langle \text{proof} \rangle$

lemma *Re_sum* [simp]:
 $\langle \text{Re } (\text{sum } f \ s) = (\sum x \in s. \text{Re } (f \ x)) \rangle$
 $\langle \text{proof} \rangle$

lemma *Im_sum* [simp]:
 $\langle \text{Im } (\text{sum } f \ s) = (\sum x \in s. \text{Im } (f \ x)) \rangle$
 $\langle \text{proof} \rangle$

instance *gauss* :: *idom*
 $\langle \text{proof} \rangle$

9.2 The Gauss Number i

primcorec *imaginary_unit* :: *gauss* ($\langle i \rangle$)
where
 $\langle \text{Re } i = 0 \rangle$
 $| \langle \text{Im } i = 1 \rangle$

lemma *Gauss_eq*:
 $\langle \text{Gauss } a \ b = \text{of_int } a + i * \text{of_int } b \rangle$

$\langle \text{proof} \rangle$

lemma *gauss_eq*:

$\langle a = \text{of_int } (\text{Re } a) + i * \text{of_int } (\text{Im } a) \rangle$
 $\langle \text{proof} \rangle$

lemma *gauss_i_not_zero* [simp]:

$\langle i \neq 0 \rangle$
 $\langle \text{proof} \rangle$

lemma *gauss_i_not_one* [simp]:

$\langle i \neq 1 \rangle$
 $\langle \text{proof} \rangle$

lemma *gauss_i_not_numeral* [simp]:

$\langle i \neq \text{numeral } n \rangle$
 $\langle \text{proof} \rangle$

lemma *gauss_i_not_neg_numeral* [simp]:

$\langle i \neq - \text{numeral } n \rangle$
 $\langle \text{proof} \rangle$

lemma *i_mult_i_eq* [simp]:

$\langle i * i = - 1 \rangle$
 $\langle \text{proof} \rangle$

lemma *gauss_i_mult_minus* [simp]:

$\langle i * (i * x) = - x \rangle$
 $\langle \text{proof} \rangle$

lemma *i_squared* [simp]:

$\langle i^2 = - 1 \rangle$
 $\langle \text{proof} \rangle$

lemma *i_even_power* [simp]:

$\langle i ^ (n * 2) = (- 1) ^ n \rangle$
 $\langle \text{proof} \rangle$

lemma *Re_i_times* [simp]:

$\langle \text{Re } (i * z) = - \text{Im } z \rangle$
 $\langle \text{proof} \rangle$

lemma *Im_i_times* [simp]:

$\langle \text{Im } (i * z) = \text{Re } z \rangle$
 $\langle \text{proof} \rangle$

lemma *i_times_eq_iff*:

$\langle i * w = z \longleftrightarrow w = - (i * z) \rangle$
 $\langle \text{proof} \rangle$

lemma *is_unit_i* [*simp*]:

⟨i dvd 1⟩
 ⟨proof⟩

lemma *gauss_numeral* [*code_post*]:

⟨Gauss 0 0 = 0⟩
 ⟨Gauss 1 0 = 1⟩
 ⟨Gauss (- 1) 0 = - 1⟩
 ⟨Gauss (numeral n) 0 = numeral n⟩
 ⟨Gauss (- numeral n) 0 = - numeral n⟩
 ⟨Gauss 0 1 = i⟩
 ⟨Gauss 0 (- 1) = - i⟩
 ⟨Gauss 0 (numeral n) = numeral n * i⟩
 ⟨Gauss 0 (- numeral n) = - numeral n * i⟩
 ⟨Gauss 1 1 = 1 + i⟩
 ⟨Gauss (- 1) 1 = - 1 + i⟩
 ⟨Gauss (numeral n) 1 = numeral n + i⟩
 ⟨Gauss (- numeral n) 1 = - numeral n + i⟩
 ⟨Gauss 1 (- 1) = 1 - i⟩
 ⟨Gauss 1 (numeral n) = 1 + numeral n * i⟩
 ⟨Gauss 1 (- numeral n) = 1 - numeral n * i⟩
 ⟨Gauss (- 1) (- 1) = - 1 - i⟩
 ⟨Gauss (numeral n) (- 1) = numeral n - i⟩
 ⟨Gauss (- numeral n) (- 1) = - numeral n - i⟩
 ⟨Gauss (- 1) (numeral n) = - 1 + numeral n * i⟩
 ⟨Gauss (- 1) (- numeral n) = - 1 - numeral n * i⟩
 ⟨Gauss (numeral m) (numeral n) = numeral m + numeral n * i⟩
 ⟨Gauss (- numeral m) (numeral n) = - numeral m + numeral n * i⟩
 ⟨Gauss (numeral m) (- numeral n) = numeral m - numeral n * i⟩
 ⟨Gauss (- numeral m) (- numeral n) = - numeral m - numeral n * i⟩
 ⟨proof⟩

9.3 Gauss Conjugation

primcorec *cnj* :: ⟨gauss ⇒ gauss⟩

where

⟨Re (cnj z) = Re z⟩
 | ⟨Im (cnj z) = - Im z⟩

lemma *gauss_cnj_cancel_iff* [*simp*]:

⟨cnj x = cnj y ⟷ x = y⟩
 ⟨proof⟩

lemma *gauss_cnj_cnj* [*simp*]:

⟨cnj (cnj z) = z⟩
 ⟨proof⟩

lemma *gauss_cnj_zero* [*simp*]:

$\langle \text{cnj } 0 = 0 \rangle$
 $\langle \text{proof} \rangle$

lemma *gauss_cnj_zero_iff* [iff]:
 $\langle \text{cnj } z = 0 \iff z = 0 \rangle$
 $\langle \text{proof} \rangle$

lemma *gauss_cnj_one_iff* [simp]:
 $\langle \text{cnj } z = 1 \iff z = 1 \rangle$
 $\langle \text{proof} \rangle$

lemma *gauss_cnj_add* [simp]:
 $\langle \text{cnj } (x + y) = \text{cnj } x + \text{cnj } y \rangle$
 $\langle \text{proof} \rangle$

lemma *cnj_sum* [simp]:
 $\langle \text{cnj } (\text{sum } f \ s) = (\sum x \in s. \text{cnj } (f \ x)) \rangle$
 $\langle \text{proof} \rangle$

lemma *gauss_cnj_diff* [simp]:
 $\langle \text{cnj } (x - y) = \text{cnj } x - \text{cnj } y \rangle$
 $\langle \text{proof} \rangle$

lemma *gauss_cnj_minus* [simp]:
 $\langle \text{cnj } (-x) = - \text{cnj } x \rangle$
 $\langle \text{proof} \rangle$

lemma *gauss_cnj_one* [simp]:
 $\langle \text{cnj } 1 = 1 \rangle$
 $\langle \text{proof} \rangle$

lemma *gauss_cnj_mult* [simp]:
 $\langle \text{cnj } (x * y) = \text{cnj } x * \text{cnj } y \rangle$
 $\langle \text{proof} \rangle$

lemma *cnj_prod* [simp]:
 $\langle \text{cnj } (\text{prod } f \ s) = (\prod x \in s. \text{cnj } (f \ x)) \rangle$
 $\langle \text{proof} \rangle$

lemma *gauss_cnj_power* [simp]:
 $\langle \text{cnj } (x ^ n) = \text{cnj } x ^ n \rangle$
 $\langle \text{proof} \rangle$

lemma *gauss_cnj_numeral* [simp]:
 $\langle \text{cnj } (\text{numeral } w) = \text{numeral } w \rangle$
 $\langle \text{proof} \rangle$

lemma *gauss_cnj_of_nat* [simp]:
 $\langle \text{cnj } (\text{of_nat } n) = \text{of_nat } n \rangle$

$\langle \text{proof} \rangle$

lemma *gauss_cnj_of_int* [simp]:
 $\langle \text{cnj} (\text{of_int } z) = \text{of_int } z \rangle$
 $\langle \text{proof} \rangle$

lemma *gauss_cnj_i* [simp]:
 $\langle \text{cnj } i = -i \rangle$
 $\langle \text{proof} \rangle$

lemma *gauss_add_cnj*:
 $\langle z + \text{cnj } z = \text{of_int } (2 * \text{Re } z) \rangle$
 $\langle \text{proof} \rangle$

lemma *gauss_diff_cnj*:
 $\langle z - \text{cnj } z = \text{of_int } (2 * \text{Im } z) * i \rangle$
 $\langle \text{proof} \rangle$

lemma *gauss_mult_cnj*:
 $\langle z * \text{cnj } z = \text{of_int } ((\text{Re } z)^2 + (\text{Im } z)^2) \rangle$
 $\langle \text{proof} \rangle$

lemma *cnj_add_mult_eq_Re*:
 $\langle z * \text{cnj } w + \text{cnj } z * w = \text{of_int } (2 * \text{Re } (z * \text{cnj } w)) \rangle$
 $\langle \text{proof} \rangle$

lemma *gauss_In_mult_cnj_zero* [simp]:
 $\langle \text{Im } (z * \text{cnj } z) = 0 \rangle$
 $\langle \text{proof} \rangle$

9.4 Algebraic division

instantiation *gauss* :: *idom_modulo*
begin

primcorec *divide_gauss* :: $\langle \text{gauss} \Rightarrow \text{gauss} \Rightarrow \text{gauss} \rangle$
where
 $\langle \text{Re } (x \text{ div } y) = (\text{Re } x * \text{Re } y + \text{Im } x * \text{Im } y) \text{ cdiv } ((\text{Re } y)^2 + (\text{Im } y)^2) \rangle$
 $| \langle \text{Im } (x \text{ div } y) = (\text{Im } x * \text{Re } y - \text{Re } x * \text{Im } y) \text{ cdiv } ((\text{Re } y)^2 + (\text{Im } y)^2) \rangle$

primcorec *modulo_gauss* :: $\langle \text{gauss} \Rightarrow \text{gauss} \Rightarrow \text{gauss} \rangle$
where

$\langle \text{Re } (x \text{ mod } y) = \text{Re } x -$
 $((\text{Re } x * \text{Re } y + \text{Im } x * \text{Im } y) \text{ cdiv } ((\text{Re } y)^2 + (\text{Im } y)^2) * \text{Re } y -$
 $(\text{Im } x * \text{Re } y - \text{Re } x * \text{Im } y) \text{ cdiv } ((\text{Re } y)^2 + (\text{Im } y)^2) * \text{Im } y) \rangle$
 $| \langle \text{Im } (x \text{ mod } y) = \text{Im } x -$
 $((\text{Re } x * \text{Re } y + \text{Im } x * \text{Im } y) \text{ cdiv } ((\text{Re } y)^2 + (\text{Im } y)^2) * \text{Im } y +$
 $(\text{Im } x * \text{Re } y - \text{Re } x * \text{Im } y) \text{ cdiv } ((\text{Re } y)^2 + (\text{Im } y)^2) * \text{Re } y) \rangle$

```

instance ⟨proof⟩

end

instantiation gauss :: euclidean_ring
begin

definition euclidean_size_gauss :: ⟨gauss ⇒ nat⟩
  where ⟨euclidean_size x = nat ((Re x)2 + (Im x)2)⟩

instance ⟨proof⟩

end

end

```

10 Groebner Basis Examples

```

theory Groebner_Examples
imports Main
begin

```

10.1 Basic examples

```

lemma
  fixes x :: int
  shows x ^ 3 = x ^ 3
  ⟨proof⟩

lemma
  fixes x :: int
  shows (x - (-2)) ^ 5 = x ^ 5 + (10 * x ^ 4 + (40 * x ^ 3 + (80 * x2 + (80
* x + 32))))
  ⟨proof⟩

schematic_goal
  fixes x :: int
  shows (x - (-2)) ^ 5 * (y - 78) ^ 8 = ?X
  ⟨proof⟩

lemma ((-3) ^ (Suc (Suc (Suc 0)))) == (X::'a::{comm_ring_1})
  ⟨proof⟩

lemma ((x::int) + y) ^ 3 - 1 = (x - z) ^ 2 - 10 ⇒ x = z + 3 ⇒ x = - y
  ⟨proof⟩

lemma (4::nat) + 4 = 3 + 5
  ⟨proof⟩

```

lemma $(4::int) + 0 = 4$
 $\langle proof \rangle$

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

lemma $(x::int) \wedge 3 - x \wedge 2 - 5 * x - 3 = 0 \longleftrightarrow (x = 3 \vee x = -1)$
 $\langle proof \rangle$

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

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

10.2 Lemmas for Lagrange's theorem

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

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

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⟩

```

10.3 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*)

⟨proof⟩

lemma $\exists (d::int). a*y - a*x = n*d \implies \exists u\ v. a*u + n*v = 1 \implies \exists e. y - x = n*e$

⟨proof⟩

end

11 Example of Declaring an Oracle

theory *Iff_Oracle*

imports *Main*

begin

11.1 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.

⟨ML⟩

11.2 Oracle as low-level rule

⟨ML⟩

These oracle calls had better fail.

⟨ML⟩

11.3 Oracle as proof method

$\langle ML \rangle$

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

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

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

lemma A
 $\langle proof \rangle$

end

12 Examples of automatically derived induction rules

theory *Induction_Schema*
imports *Main*
begin

12.1 Some simple induction principles on nat

lemma *nat_standard_induct*:
 $\llbracket P\ 0; \bigwedge n. P\ n \implies P\ (Suc\ n) \rrbracket \implies P\ x$
 $\langle proof \rangle$

lemma *nat_induct2*:
 $\llbracket P\ 0; P\ (Suc\ 0); \bigwedge k. P\ k \implies P\ (Suc\ k) \implies P\ (Suc\ (Suc\ k)) \rrbracket$
 $\implies P\ n$
 $\langle proof \rangle$

lemma *minus_one_induct*:
 $\llbracket \bigwedge n::nat. (n \neq 0 \implies P\ (n - 1)) \rrbracket \implies P\ n \implies P\ x$
 $\langle proof \rangle$

theorem *diff_induct*:
 $(!!x. P\ x\ 0) \implies (!!y. P\ 0\ (Suc\ y)) \implies$
 $(!!x\ y. P\ x\ y \implies P\ (Suc\ x)\ (Suc\ y)) \implies P\ m\ n$
 $\langle proof \rangle$

lemma *list_induct2'*:
 $\llbracket P\ [] \rrbracket;$
 $\bigwedge x\ xs. P\ (x\#xs)\ [];$

```


$$\bigwedge y \text{ } ys. P \text{ } \llbracket (y \# ys) \rrbracket;$$


$$\bigwedge x \text{ } xs \text{ } y \text{ } ys. P \text{ } xs \text{ } ys \implies P \text{ } (x \# xs) \text{ } (y \# ys) \rrbracket$$


$$\implies P \text{ } xs \text{ } ys$$


$$\langle proof \rangle$$


```

```

theorem even_odd_induct:
  assumes  $R \text{ } 0$ 
  assumes  $Q \text{ } 0$ 
  assumes  $\bigwedge n. Q \text{ } n \implies R \text{ } (Suc \text{ } n)$ 
  assumes  $\bigwedge n. R \text{ } n \implies Q \text{ } (Suc \text{ } n)$ 
  shows  $R \text{ } n \text{ } Q \text{ } n$ 
   $\langle proof \rangle$ 

```

end

13 Textbook-style reasoning: the Knaster-Tarski Theorem

```

theory Knaster_Tarski
  imports Main
begin

```

```

unbundle lattice_syntax

```

13.1 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 $f: L \rightarrow L$ an order-preserving map. Then $\bigcap \{x \in L \mid f(x) \leq x\}$ is a fixpoint of f .

Proof. Let $H = \{x \in L \mid f(x) \leq x\}$ and $a = \bigcap H$. 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 $a \leq f(a)$.

13.2 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.

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

```

theorem Knaster_Tarski:
  fixes f :: 'a::complete_lattice  $\Rightarrow$  'a
  assumes mono f
  shows  $\exists a. f\ a = a$ 
 $\langle proof \rangle$ 

```

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  $\exists a. f\ a = a$ 
 $\langle proof \rangle$ 

```

end

14 Isabelle/ML basics

```

theory ML
  imports Main
begin

```

14.1 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.

$\langle ML \rangle$

14.2 Antiquotations

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

$\langle ML \rangle$

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

term $1 + 1$ — term within theory source

$\langle ML \rangle$

14.3 Recursive ML evaluation

$\langle ML \rangle$

14.4 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

$\langle ML \rangle$

14.5 Example: factorial and ackermann function in Isabelle/ML

$\langle ML \rangle$

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

$\langle ML \rangle$

14.6 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.

$\langle ML \rangle$

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

$\langle ML \rangle$

14.7 Function specifications in Isabelle/HOL

```
fun factorial :: nat  $\Rightarrow$  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  $\Rightarrow$  nat  $\Rightarrow$  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
```

15 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>
```

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*.

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

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>

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 their 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

16 Using extensible records in HOL – points and coloured points

theory *Records*
imports *Main*
begin

16.1 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:

$point = \langle xpos :: nat, ypos :: nat \rangle = () \text{ point_ext_type}$
 $'a \text{ point_scheme} = \langle xpos :: nat, ypos :: nat, \dots :: 'a \rangle = 'a \text{ point_ext_type}$

consts $foo2 :: \langle xpos :: nat, ypos :: nat \rangle$
consts $foo4 :: 'a \Rightarrow \langle xpos :: nat, ypos :: nat, \dots :: 'a \rangle$

16.1.1 Introducing concrete records and record schemes

definition $foo1 :: point$
where $foo1 = \langle xpos = 1, ypos = 0 \rangle$

definition $foo3 :: 'a \Rightarrow 'a \text{ point_scheme}$
where $foo3 \text{ ext} = \langle xpos = 1, ypos = 0, \dots = \text{ext} \rangle$

16.1.2 Record selection and record update

definition $getX :: 'a \text{ point_scheme} \Rightarrow nat$
where $getX \ r = xpos \ r$

definition $setX :: 'a \text{ point_scheme} \Rightarrow nat \Rightarrow 'a \text{ point_scheme}$
where $setX \ r \ n = r \ \langle xpos := n \rangle$

16.1.3 Some lemmas about records

Basic simplifications.

lemma $point.make \ n \ p = \langle xpos = n, ypos = p \rangle$
 $\langle proof \rangle$

lemma $xpos \ \langle xpos = m, ypos = n, \dots = p \rangle = m$
 $\langle proof \rangle$

lemma $\langle xpos = m, ypos = n, \dots = p \rangle \langle xpos := 0 \rangle = \langle xpos = 0, ypos = n, \dots = p \rangle$
 $\langle proof \rangle$

Equality of records.

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

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

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

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

Surjective pairing

lemma $r = \llbracket xpos = xpos\ r, ypos = ypos\ r \rrbracket$
 $\langle proof \rangle$

lemma $r = \llbracket xpos = xpos\ r, ypos = ypos\ r, \dots = point.more\ r \rrbracket$
 $\langle proof \rangle$

Representation of records by cases or (degenerate) induction.

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

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

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

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

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

Concrete records are type instances of record schemes.

definition $foo5 :: nat$
where $foo5 = getX\ \llbracket xpos = 1, ypos = 0 \rrbracket$

Manipulating the “...” (more) part.

definition $incX :: 'a\ point_scheme \Rightarrow 'a\ point_scheme$
where $incX\ r = \llbracket xpos = xpos\ r + 1, ypos = ypos\ r, \dots = point.more\ r \rrbracket$

lemma $incX\ r = setX\ r\ (Suc\ (getX\ r))$
 $\langle proof \rangle$

An alternative definition.

definition $incX' :: 'a\ point_scheme \Rightarrow 'a\ point_scheme$
where $incX'\ r = r(\llbracket xpos := xpos\ r + 1 \rrbracket)$

16.2 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, ... :: '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)
```

16.2.1 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
```

16.3 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
  <proof>
```

Only the most recent update to a component survives simplification.

```
lemma r(xpos := x, ypos := y, xpos := x') = r(ypos := y, xpos := x')
  <proof>
```

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
 $\langle proof \rangle$
end

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

lemma $\exists r. xpos\ r = x$
 $\langle proof \rangle$

16.4 Simprocs for update and equality

record *alph1* =
 $a :: nat$
 $b :: nat$

```
record alph2 = alph1 +
  c :: nat
  d :: nat
```

```
record alph3 = alph2 +
  e :: nat
  f :: nat
```

The simprocs that are activated by default are:

- `Record.simproc`: field selection of (nested) record updates.
- `Record.upd_simproc`: nested record updates.
- `Record.eq_simproc`: (componentwise) equality of records.

By default record updates are not ordered by simplification.

```
schematic__goal r⟦b := x, a:= y⟧ = ?X
  ⟨proof⟩
```

Normalisation towards an update ordering (string ordering of update function names) can be configured as follows.

```
schematic__goal r⟦b := y, a := x⟧ = ?X
  ⟨proof⟩
```

Note the interplay between update ordering and record equality. Without update ordering the following equality is handled by `Record.eq_simproc`. Record equality is thus solved by componentwise comparison of all the fields of the records which can be expensive in the presence of many fields.

```
lemma r⟦f := x1, a:= x2⟧ = r⟦a := x2, f:= x1⟧
  ⟨proof⟩
```

```
lemma r⟦f := x1, a:= x2⟧ = r⟦a := x2, f:= x1⟧
  ⟨proof⟩
```

With update ordering the equality is already established after update normalisation. There is no need for componentwise comparison.

```
lemma r⟦f := x1, a:= x2⟧ = r⟦a := x2, f:= x1⟧
  ⟨proof⟩
```

```
schematic__goal r⟦f := x1, e := x2, d:= x3, c:= x4, b:=x5, a:= x6⟧ = ?X
  ⟨proof⟩
```

```
schematic__goal r⟦f := x1, e := x2, d:= x3, c:= x4, e:=x5, a:= x6⟧ = ?X
  ⟨proof⟩
```

```
schematic__goal r⟦f := x1, e := x2, d:= x3, c:= x4, e:=x5, a:= x6⟧ = ?X
  ⟨proof⟩
```

16.5 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
```

16.6 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
```

⟨ML⟩

```
declare [[record_codegen]]
```

```
schematic_goal ⟨fld_1 (r⟨fld_300 := x300, fld_20 := x20, fld_200 := x200⟩)
= ?X⟩
  ⟨proof⟩
```

```
schematic_goal ⟨r⟨fld_300 := x300, fld_20 := x20, fld_200 := x200⟩ = ?X⟩
  ⟨proof⟩
```

```
end
theory Rewrite_Examples
imports Main HOL-Library.Rewrite
begin
```

17 The rewrite Proof Method by Example

This theory gives an overview over the features of the pattern-based rewrite proof method.

Documentation: <https://arxiv.org/abs/2111.04082>

```
lemma
  fixes a::int and b::int and c::int
  assumes P (b + a)
  shows P (a + b)
```

$\langle proof \rangle$

lemma

fixes $a\ b\ c :: int$

assumes $f\ (a - a + (a - a)) + f\ (0 + c) = f\ 0 + f\ c$

shows $f\ (a - a + (a - a)) + f\ ((a - a) + c) = f\ 0 + f\ c$

$\langle proof \rangle$

lemma

fixes $a\ b\ c :: int$

assumes $f\ (a - a + 0) + f\ ((a - a) + c) = f\ 0 + f\ c$

shows $f\ (a - a + (a - a)) + f\ ((a - a) + c) = f\ 0 + f\ c$

$\langle proof \rangle$

lemma

fixes $a\ b\ c :: int$

assumes $f\ (0 + (a - a)) + f\ ((a - a) + c) = f\ 0 + f\ c$

shows $f\ (a - a + (a - a)) + f\ ((a - a) + c) = f\ 0 + f\ c$

$\langle proof \rangle$

lemma

fixes $a\ b\ c :: int$

assumes $f\ (a - a + 0) + f\ ((a - a) + c) = f\ 0 + f\ c$

shows $f\ (a - a + (a - a)) + f\ ((a - a) + c) = f\ 0 + f\ c$

$\langle proof \rangle$

lemma

fixes $x\ y :: nat$

shows $x + y > c \implies y + x > c$

$\langle proof \rangle$

lemma

fixes $x\ y :: nat$

assumes $y + x > c \implies y + x > c$

shows $x + y > c \implies y + x > c$

$\langle proof \rangle$

lemma

fixes $x\ y :: nat$

assumes $y + x > c \implies y + x > c$

shows $x + y > c \implies y + x > c$

$\langle proof \rangle$

lemma

fixes $x\ y :: nat$

assumes $y + x > c \implies y + x > c$

shows $x + y > c \implies y + x > c$

$\langle proof \rangle$

lemma

assumes $P \{x::int. y + 1 = 1 + x\}$
shows $P \{x::int. y + 1 = x + 1\}$
 $\langle proof \rangle$

lemma

assumes $P \{x::int. y + 1 = 1 + x\}$
shows $P \{x::int. y + 1 = x + 1\}$
 $\langle proof \rangle$

lemma

assumes $P \{(x::nat, y::nat, z). x + z * 3 = Q (\lambda s t. s * t + y - 3)\}$
shows $P \{(x::nat, y::nat, z). x + z * 3 = Q (\lambda s t. y + s * t - 3)\}$
 $\langle proof \rangle$

lemma

assumes $PROP P \equiv PROP Q$
shows $PROP R \implies PROP P \implies PROP Q$
 $\langle proof \rangle$

lemma

assumes $PROP P \equiv PROP Q$
shows $PROP R \implies PROP R \implies PROP P \implies PROP Q$
 $\langle proof \rangle$

lemma

assumes $(PROP P \implies PROP Q) \equiv (PROP S \implies PROP R)$
shows $PROP S \implies (PROP P \implies PROP Q) \implies PROP R$
 $\langle proof \rangle$

lemma *test_theorem:*

fixes $x :: nat$
shows $x \leq y \implies x \geq y \implies x = y$
 $\langle proof \rangle$

lemma

fixes $f :: nat \Rightarrow nat$
shows $f x \leq 0 \implies f x \geq 0 \implies f x = 0$
 $\langle proof \rangle$

lemma

assumes *rewr*: $PROP\ P \implies PROP\ Q \implies PROP\ R \equiv PROP\ R'$
assumes *A1*: $PROP\ S \implies PROP\ T \implies PROP\ U \implies PROP\ P$
assumes *A2*: $PROP\ S \implies PROP\ T \implies PROP\ U \implies PROP\ Q$
assumes *C*: $PROP\ S \implies PROP\ R' \implies PROP\ T \implies PROP\ U \implies PROP\ V$
shows $PROP\ S \implies PROP\ R \implies PROP\ T \implies PROP\ U \implies PROP\ V$
<proof>

fun *f* :: $nat \Rightarrow nat$ **where** *f* *n* = *n*

definition *f_inv* (*I* :: $nat \Rightarrow bool$) *n* \equiv *f* *n*

lemma *annotate_f*: $f = f_inv\ I$

<proof>

lemma

assumes $P\ (\lambda n. f_inv\ (\lambda _. True)\ n + 1) = x$

shows $P\ (\lambda n. f\ n + 1) = x$

<proof>

lemma

assumes $P\ (\lambda n. f_inv\ (\lambda x. n < x + 1)\ n + 1) = x$

shows $P\ (\lambda n. f\ n + 1) = x$

<proof>

lemma

assumes $P\ (\lambda n. f_inv\ (\lambda x. n < x + 1)\ n + 1) = x$

shows $P\ (\lambda n. f\ n + 1) = x$

<proof>

lemma

assumes $P\ (2 + 1)$

shows $\bigwedge x\ y. P\ (1 + 2 :: nat)$

<proof>

lemma

assumes $\bigwedge x\ y. P\ (y + x)$

shows $\bigwedge x\ y. P\ (x + y :: nat)$

<proof>

lemma

assumes $\bigwedge x\ y\ z. y + x + z = z + y + (x :: int)$

shows $\bigwedge x\ y\ z. x + y + z = z + y + (x::int)$
 $\langle proof \rangle$

lemma
assumes $\bigwedge x\ y\ z. z + (x + y) = z + y + (x::int)$
shows $\bigwedge x\ y\ z. x + y + z = z + y + (x::int)$
 $\langle proof \rangle$

lemma
assumes $\bigwedge x\ y\ z. x + y + z = y + z + (x::int)$
shows $\bigwedge x\ y\ z. x + y + z = z + y + (x::int)$
 $\langle proof \rangle$

lemma
assumes $eq: \bigwedge x. P\ x \implies g\ x = x$
assumes $f1: \bigwedge x. Q\ x \implies P\ x$
assumes $f2: \bigwedge x. Q\ x \implies x$
shows $\bigwedge x. Q\ x \implies g\ x$
 $\langle proof \rangle$

lemma
assumes $(\bigwedge (x::int). x < 1 + x)$
and $(x::int) + 1 > x$
shows $(\bigwedge (x::int). x + 1 > x) \implies (x::int) + 1 > x$
 $\langle proof \rangle$

lemma
assumes $\bigwedge a\ b. P\ ((a + 1) * (1 + b))$
shows $\bigwedge a\ b :: nat. P\ ((a + 1) * (b + 1))$
 $\langle proof \rangle$

lemma
assumes $Q\ (\lambda b :: int. P\ (\lambda a. a + b)\ (\lambda a. a + b))$
shows $Q\ (\lambda b :: int. P\ (\lambda a. a + b)\ (\lambda a. b + a))$
 $\langle proof \rangle$

$\langle ML \rangle$

Some regression tests

$\langle ML \rangle$

lemma
assumes $eq: PROP\ A \implies PROP\ B \equiv PROP\ C$
assumes $f1: PROP\ D \implies PROP\ A$
assumes $f2: PROP\ D \implies PROP\ C$
shows $\bigwedge x. PROP\ D \implies PROP\ B$

<proof>

end

18 Finite sequences

theory *Seq*

imports *Main*

begin

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

fun *conc* :: *'a seq* \Rightarrow *'a seq* \Rightarrow *'a seq*

where

conc Empty ys = *ys*

| *conc (Seq x xs) ys* = *Seq x (conc xs ys)*

fun *reverse* :: *'a seq* \Rightarrow *'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

19 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* \notin \mathbb{Q}

<proof>

corollary *sqrt_2_not_rat*: $\text{sqrt } 2 \notin \mathbb{Q}$
 $\langle \text{proof} \rangle$

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 $p :: \text{nat}$
assumes *prime* p
shows $\text{sqrt } p \notin \mathbb{Q}$
 $\langle \text{proof} \rangle$

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

lemma $\exists a b :: \text{real}. a \notin \mathbb{Q} \wedge b \notin \mathbb{Q} \wedge a \text{ powr } b \in \mathbb{Q}$ (**is** $\exists a b. ?P a b$)
 $\langle \text{proof} \rangle$

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.