# Notable Examples in Isabelle/HOL

October 24, 2026

## Contents


**1 Ad Hoc Overloading** **3**
- 1.1 Plain Ad Hoc Overloading . . . . . 4
- 1.2 Adhoc Overloading inside Locales . . . . . 5

**2 Permutation Types** **7**

**3 A Tail-Recursive, Stack-Based Ackermann's Function** **8**
- 3.1 Example of proving termination by reasoning about the domain 9
- 3.2 Example of proving termination using a multiset ordering . . 10

**4 Cantor's Theorem** **12**
- 4.1 Mathematical statement and proof . . . . . 12
- 4.2 Automated proofs . . . . . 12
- 4.3 Elementary version in higher-order predicate logic . . . . . 12
- 4.4 Classic Isabelle/HOL example . . . . . 13

**5 Coherent Logic Problems** **14**
- 5.1 Equivalence of two versions of Pappus' Axiom . . . . . 14
- 5.2 Preservation of the Diamond Property under reflexive closure 16

**6 Some Isar command definitions** **16**
- 6.1 Diagnostic command: no state change . . . . . 16
- 6.2 Old-style global theory declaration . . . . . 16
- 6.3 Local theory specification . . . . . 17

**7 The Drinker's Principle** **18**

**8 Examples of function definitions** **18**
- 8.1 Very basic . . . . . 19
- 8.2 Currying . . . . . 19
- 8.3 Nested recursion . . . . . 19
  - 8.3.1 Here comes McCarthy's 91-function . . . . . 20

8.3.2 Here comes Takeuchi's function . . . 20
8.4 More general patterns . . . 21
8.4.1 Overlapping patterns . . . 21
8.4.2 Guards . . . 21
8.5 Mutual Recursion . . . 22
8.6 Definitions in local contexts . . . 22
8.7 *fun_cases* . . . 23
8.7.1 Predecessor . . . 23
8.7.2 List to option . . . 24
8.7.3 Boolean Functions . . . 24
8.7.4 Many parameters . . . 24
8.8 Partial Function Definitions . . . 24
8.9 Regression tests . . . 25
8.9.1 Context recursion . . . 25
8.9.2 A combination of context and nested recursion . . . 25
8.9.3 Context, but no recursive call . . . 25
8.9.4 Tupled nested recursion . . . 26
8.9.5 Let . . . 26
8.9.6 Abbreviations . . . 26
8.9.7 Simple Higher-Order Recursion . . . 26
8.9.8 Pattern matching on records . . . 27
8.9.9 The diagonal function . . . 27
8.9.10 Many equations (quadratic blowup) . . . 27
8.9.11 Automatic pattern splitting . . . 28
8.9.12 Polymorphic partial-function . . . 28

**9 Gauss Numbers: integral gauss numbers 28**
9.1 Basic arithmetic . . . 28
9.2 The Gauss Number $i$ . . . 30
9.3 Gauss Conjugation . . . 32
9.4 Algebraic division . . . 34

**10 Groebner Basis Examples 36**
10.1 Basic examples . . . 36
10.2 Lemmas for Lagrange's theorem . . . 38
10.3 Colinearity is invariant by rotation . . . 38

**11 Example of Declaring an Oracle 39**
11.1 Oracle declaration . . . 39
11.2 Oracle as low-level rule . . . 39
11.3 Oracle as proof method . . . 40

**12 Examples of automatically derived induction rules 40**
12.1 Some simple induction principles on nat . . . 40

**13 Textbook-style reasoning: the Knaster-Tarski Theorem** **41**
13.1 Prose version . . . . . 41
13.2 Formal versions . . . . . 42

**14 Isabelle/ML basics** **43**
14.1 ML expressions . . . . . 43
14.2 Antiquotations . . . . . 44
14.3 Recursive ML evaluation . . . . . 44
14.4 IDE support . . . . . 44
14.5 Example: factorial and ackermann function in Isabelle/ML . 45
14.6 Parallel Isabelle/ML . . . . . 45
14.7 Function specifications in Isabelle/HOL . . . . . 45

**15 Peirce's Law** **46**

**16 Using extensible records in HOL – points and coloured points** **47**
16.1 Points . . . . . 48
16.1.1 Introducing concrete records and record schemes . . . 48
16.1.2 Record selection and record update . . . . . 48
16.1.3 Some lemmas about records . . . . . 48
16.2 Coloured points: record extension . . . . . 50
16.2.1 Non-coercive structural subtyping . . . . . 51
16.3 Other features . . . . . 51
16.4 Simprocs for update and equality . . . . . 53
16.5 A more complex record expression . . . . . 54
16.6 Some code generation . . . . . 54

**17 Examples for generic reflection and reification** **55**

**18 The rewrite Proof Method by Example** **65**

**19 Finite sequences** **71**

**20 Square roots of primes are irrational** **72**

# 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* $\rightleftharpoons$ *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* $\rightleftharpoons$ *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* $\rightleftharpoons$ *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* $\rightleftharpoons$ *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* :: ($'a$ ⇒ $'a$) *set* **where**
  *perms* = {*f*. *bij f* ∧ *finite* {*x*. *f x* ≠ *x*}}

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

First we need some auxiliary lemmas.

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

**lemma** *perms_imp_bij*:
  *f* ∈ *perms* ⟹ *bij f*
  **by** (*simp add*: *perms_def*)

**lemma** *perms_imp_MOST_eq*:
  *f* ∈ *perms* ⟹ *MOST x*. *f x* = *x*
  **by** (*simp add*: *perms_def*) (*metis MOST_iff_finiteNeg*)

**lemma** *id_perms* [*simp*]:
  *id* ∈ *perms*
  (λ*x*. *x*) ∈ *perms*
  **by** (*auto simp*: *perms_def bij_def*)

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

**lemma** *perms_inv*:
**assumes** *f*: $f \in perms$
**shows** $inv\ f \in perms$
**apply** (*rule permsI*)
**apply** (*rule bij_imp_bij_inv*)
**apply** (*rule perms_imp_bij* [*OF f*])
**apply** (*rule MOST_mono* [*OF perms_imp_MOST_eq* [*OF f*]])
**apply** (*erule subst*, *rule inv_f_f*)
**apply** (*rule bij_is_inj* [*OF perms_imp_bij* [*OF f*]])
**done**

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

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

**definition** $0 = Abs_perm\ id$
**definition** $- p = Abs_perm\ (inv\ (Rep_perm\ p))$
**definition** $p + q = Abs_perm\ (Rep_perm\ p \circ Rep_perm\ q)$
**definition** $(p1::'a\ perm) - p2 = p1 + - p2$

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

**lemma** *Rep_perm_add*:
$Rep_perm\ (p1 + p2) = Rep_perm\ p1 \circ Rep_perm\ p2$
**unfolding** *plus_perm_def* **by** (*simp add*: *Abs_perm_inverse Rep_perm*)

**lemma** *Rep_perm_uminus*:
$Rep_perm\ (- p) = inv\ (Rep_perm\ p)$
**unfolding** *uminus_perm_def* **by** (*simp add*: *Abs_perm_inverse perms_inv Rep_perm*)

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

**end**

**lemmas** *Rep_perm_simps* =
*Rep_perm_0*
*Rep_perm_add*

*Rep_perm_uminus*

## 2 Permutation Types

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

**consts** *PERMUTE* :: *'a perm* ⇒ *'b* ⇒ *'b* (**infixr** ‹•› *75*)

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

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

**adhoc_overloading**
  *PERMUTE* ⇌ *permute*

**end**

Permuting atoms.

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

**adhoc_overloading**
  *PERMUTE* ⇌ *permute_atom*

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

Permuting permutations.

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

**adhoc_overloading**
  *PERMUTE* ⇌ *permute_perm*

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

Permuting functions.

**locale** *fun_permute* =
  *dom*: *permute perm1* + *ran*: *permute perm2*

**for** *perm1* :: *'a perm* ⇒ *'b* ⇒ *'b*
**and** *perm2* :: *'a perm* ⇒ *'c* ⇒ *'c*
**begin**

**adhoc_overloading**
*PERMUTE* ⇌ *perm1 perm2*

**definition** *permute_fun* :: *'a perm* ⇒ (*'b* ⇒ *'c*) ⇒ (*'b* ⇒ *'c*) **where**
*permute_fun p f* = (λ*x*. *p* ∙ (*f* (−*p* ∙ *x*)))

**adhoc_overloading**
*PERMUTE* ⇌ *permute_fun*

**end**

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

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

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

**adhoc_overloading**
*PERMUTE* ⇌ *atom_fun_permute.permute_fun*

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

**end**

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

**theory** *Ackermann*
**imports** *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*] ⇒ *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* ⇒ *nat*
  **where**
    *ackloop* (*n* # *0* # *l*) = *ackloop* (*Suc n* # *l*)
  | *ackloop* (*0* # *Suc m* # *l*) = *ackloop* (*1* # *m* # *l*)
  | *ackloop* (*Suc n* # *Suc m* # *l*) = *ackloop* (*n* # *Suc m* # *m* # *l*)
  | *ackloop* [*m*] = *m*
  | *ackloop* [] = *0*
  **by** *pat_completeness auto*

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

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

*ackloop_dom* (*Suc n* # *l*) ⟹ *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*) ⟹ *ackloop_dom* (*n* # *m* # *l*)
**proof** (*induction m arbitrary*: *n l*)
  **case** *0*
  **then show** *?case*
    **by** *auto*
**next**

```
  case (Suc m)
  show ?case
    using Suc.prems
    by (induction n arbitrary: l) (simp_all add: Suc)
qed
```

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

```
lemma ackloop_dom_longer:
  ackloop_dom (ack m n # l) ⟹ ackloop_dom (n # m # l)
  by (induction m n arbitrary: l rule: ack.induct) auto
```

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

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

The induction rule for *acklist* is

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

.

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

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

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

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

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

## 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 ⇒ (nat×nat) multiset
  where
    ack_mset [] = {#}
  | ack_mset [x] = {#}
  | ack_mset (z#y#l) = mset ((y,z) # map (λx. (Suc x, 0)) l)

lemma case1: ack_mset (Suc n # l) < add_mset (0,n) {# (Suc x, 0). x ∈# mset l #}
proof (cases l)
  case (Cons m list)
  have {#(m, Suc n)#} < {#(Suc m, 0)#}
    by auto
  also have ... ≤ {#(Suc m, 0), (0,n)#}
    by auto
  finally show ?thesis
    by (simp add: Cons)
next
  case Nil
  then show ?thesis by auto
qed
```

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

```
function 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
  by pat_completeness auto
```

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

```
termination
  by (relation inv_image {(x,y). x<y} ack_mset) (auto simp: wf case1)
```

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

```
lemma Ackloop_ack: Ackloop (n # m # l) = Ackloop (ack m n # l)
  by (induction m n arbitrary: l rule: ack.induct) auto

theorem ack m n = Ackloop [n,m]
  by (simp add: Ackloop_ack)

end
```

# 4 Cantor's Theorem

**theory** *Cantor*
**imports** *Main*
**begin**

## 4.1 Mathematical statement and proof

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

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

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

## 4.2 Automated proofs

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

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

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

## 4.3 Elementary version in higher-order predicate logic

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

**lemma** *iff_contradiction*:
**assumes** $*$: $\neg\ A \longleftrightarrow A$
**shows** *False*
**proof** (*rule notE*)
**show** $\neg\ A$
**proof**

```
    assume A
    with ∗ have ¬ A ..
    from this and ‹A› show False ..
  qed
  with ∗ show A ..
qed

theorem Cantor': ∄f :: 'a ⇒ 'a ⇒ bool. ∀ A. ∃ x. A = f x
proof
  assume ∃f :: 'a ⇒ 'a ⇒ bool. ∀ A. ∃ x. A = f x
  then obtain f :: 'a ⇒ 'a ⇒ bool where ∗: ∀ A. ∃ x. A = f x ..
  let ?D = λx. ¬ f x x
  from ∗ have ∃ x. ?D = f x ..
  then obtain a where ?D = f a ..
  then have ?D a ⟷ f a a by (rule arg_cong)
  then have ¬ f a a ⟷ f a a .
  then show False by (rule iff_contradiction)
qed
```

## 4.4 Classic Isabelle/HOL example

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

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

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

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

```
theorem ∃ S. S ∉ range (f :: 'a ⇒ 'a set)
proof
  let ?S = {x. x ∉ f x}
  show ?S ∉ range f
  proof
    assume ?S ∈ range f
    then obtain y where ?S = f y ..
    then show False
    proof (rule equalityCE)
      assume y ∈ f y
```

```
    assume y ∈ ?S
    then have y ∉ f y ..
    with ‹y ∈ f y› show ?thesis by contradiction
  next
    assume y ∉ ?S
    assume y ∉ f y
    then have y ∈ ?S ..
    with ‹y ∉ ?S› show ?thesis by contradiction
  qed
 qed
qed
```

How much creativity is required? As it happens, Isabelle can prove this theorem automatically using best-first search. Depth-first search would diverge, but best-first search successfully navigates through the large search space. The context of Isabelle's classical prover contains rules for the relevant constructs of HOL's set theory.

```
theorem ∃S. S ∉ range (f :: 'a ⇒ 'a set)
 by best
```

```
end
```

# 5 Coherent Logic Problems

```
theory Coherent
imports Main
begin
```

## 5.1 Equivalence of two versions of Pappus' Axiom

```
no_notation comp  (infixl ‹o› 55)
unbundle no relcomp_syntax

lemma p1p2:
 assumes col a b c l ∧ col d e f m
   and col b f g n ∧ col c e g o
   and col b d h p ∧ col a e h q
   and col c d i r ∧ col a f i s
   and el n o ⟹ goal
   and el p q ⟹ goal
   and el s r ⟹ goal
   and ⋀A. el A A ⟹ pl g A ⟹ pl h A ⟹ pl i A ⟹ goal
   and ⋀A B C D. col A B C D ⟹ pl A D
   and ⋀A B C D. col A B C D ⟹ pl B D
   and ⋀A B C D. col A B C D ⟹ pl C D
   and ⋀A B. pl A B ⟹ ep A A
   and ⋀A B. ep A B ⟹ ep B A
   and ⋀A B C. ep A B ⟹ ep B C ⟹ 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* **using** *assms*
**by** *coherent*

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

### 5.2 Preservation of the Diamond Property under reflexive closure

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

**end**

## 6 Some Isar command definitions

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

### 6.1 Diagnostic command: no state change

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

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

### 6.2 Old-style global theory declaration

**ML** ‹

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

global_test a
global_test b
print_test a
```

## 6.3 Local theory specification

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

local_test true = True
print_test true
thm true_def

local_test identity = λx. x
print_test identity x
thm identity_def

context fixes x y :: nat
begin

local_test test = x + y
print_test test
thm test_def

end

print_test test 0 1
thm test_def

end
```

# 7 The Drinker's Principle

**theory** *Drinker*
**imports** *Main*
**begin**

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

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

**lemma** *de_Morgan*:
**assumes** $\neg\ (\forall x.\ P\ x)$
**shows** $\exists x.\ \neg\ P\ x$
**proof** (*rule classical*)
**assume** $\nexists x.\ \neg\ P\ x$
**have** $\forall x.\ P\ x$
**proof**
**fix** $x$ **show** $P\ x$
**proof** (*rule classical*)
**assume** $\neg\ P\ x$
**then have** $\exists x.\ \neg\ P\ x$ **..**
**with** $\langle\nexists x.\ \neg\ P\ x\rangle$ **show** *?thesis* **by** *contradiction*
**qed**
**qed**
**with** $\langle\neg\ (\forall x.\ P\ x)\rangle$ **show** *?thesis* **by** *contradiction*
**qed**

**theorem** *Drinker's_Principle*: $\exists x.\ drunk\ x \longrightarrow (\forall x.\ drunk\ x)$
**proof** *cases*
**assume** $\forall x.\ drunk\ x$
**then have** $drunk\ a \longrightarrow (\forall x.\ drunk\ x)$ **for** $a$ **..**
**then show** *?thesis* **..**
**next**
**assume** $\neg\ (\forall x.\ drunk\ x)$
**then have** $\exists x.\ \neg\ drunk\ x$ **by** (*rule de_Morgan*)
**then obtain** $a$ **where** $\neg\ drunk\ a$ **..**
**have** $drunk\ a \longrightarrow (\forall x.\ drunk\ x)$
**proof**
**assume** $drunk\ a$
**with** $\langle\neg\ drunk\ a\rangle$ **show** $\forall x.\ drunk\ x$ **by** *contradiction*
**qed**
**then show** *?thesis* **..**
**qed**

**end**

# 8 Examples of function definitions

**theory** *Functions*

**imports** *Main HOL−Library.Monad_Syntax*
**begin**

## 8.1 Very basic

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

Partial simp and induction rules:

**thm** *fib.psimps*
**thm** *fib.pinduct*

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

**thm** *fib.cases*

Total simp and induction rules:

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

Elimination rules:

**thm** *fib.elims*

## 8.2 Currying

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

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

## 8.3 Nested recursion

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

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

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

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

### 8.3.1 Here comes McCarthy's 91-function

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

Prove a lemma before attempting a termination proof:

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

**termination**
**proof**
  **let** *?R* = *measure* (λ*x*. *101* − *x*)
  **show** *wf ?R* **..**

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

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

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

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

### 8.3.2 Here comes Takeuchi's function

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

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

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

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

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

## 8.4 More general patterns

### 8.4.1 Overlapping patterns

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

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

**thm** *gcd2.simps*
**thm** *gcd2.induct*

### 8.4.2 Guards

We can reformulate the above example using guarded patterns:

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

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

General patterns allow even strange definitions:

**function** *ev* :: *nat* ⇒ *bool*

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

thm ev.simps
thm ev.induct
thm ev.cases
```

## 8.5 Mutual Recursion

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

thm evn.simps
thm od.simps

thm evn_od.induct
thm evn_od.termination

thm evn.elims
thm od.elims
```

## 8.6 Definitions in local contexts

```
locale my_monoid =
  fixes opr :: 'a ⇒ 'a ⇒ 'a
    and un :: 'a
```

```
  assumes assoc: opr (opr x y) z = opr x (opr y z)
    and lunit: opr un x = x
    and runit: opr x un = x
begin

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

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

thm foldL.simps

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

thm foldR_foldL

end

thm my_monoid.foldL.simps
thm my_monoid.foldR_foldL
```

## 8.7 *fun_cases*

### 8.7.1 Predecessor

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

thm pred.elims

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

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

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

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

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

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

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

### 8.7.2 List to option

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

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

**lemma** *list_to_option xs = Some y ⟹ xs = [y]*
  **by** (*erule list_to_option_SomeE*)

### 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*
  **by** (*erule sum40E*)

## 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)
by pat_completeness auto
```

### 8.9.3 Context, but no recursive call

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

### 8.9.4 Tupled nested recursion

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

### 8.9.5 Let

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

There were some problems with fresh names ...

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

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

### 8.9.6 Abbreviations

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

### 8.9.7 Simple Higher-Order Recursion

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

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

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

**fun** *testcase* :: *'a tree* ⇒ *'a list*
**where**
  *testcase* (*Leaf a*) = [*a*]
| *testcase* (*Branch x*) =

```
  (let xs = concat (map testcase x);
       ys = concat (map testcase x) in
   xs @ ys)
```

### 8.9.8 Pattern matching on records

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

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

### 8.9.9 The diagonal function

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

### 8.9.10 Many equations (quadratic blowup)

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

fun big :: DT ⇒ nat
where
  big A = 0
| big B = 0
| big C = 0
| big D = 0
| big E = 0
| big F = 0
| big G = 0
| big H = 0
| big I = 0
| big J = 0
| big K = 0
| big L = 0
```

```
| big M = 0
| big N = 0
| big P = 0
| big Q = 0
| big R = 0
| big S = 0
| big T = 0
| big U = 0
| big V = 0
```

### 8.9.11 Automatic pattern splitting

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

### 8.9.12 Polymorphic partial-function

```
partial_function (option) f5 :: 'a list ⇒ 'a option
where
  f5 x = f5 x

end
```

# 9 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?]:
  ‹x = y› if ‹Re x = Re y› ‹Im x = Im y›
  by (rule gauss.expand) (use that in simp)

lemma gauss_eq_iff:
  ‹x = y ⟷ Re x = Re y ∧ Im x = Im y›
  by (auto intro: gauss_eqI)
```

## 9.1 Basic arithmetic

```
instantiation gauss :: comm_ring_1
begin

primcorec zero_gauss :: ‹gauss›
  where
    ‹Re 0 = 0›
```

```
  | ‹Im 0 = 0›

primcorec one_gauss :: ‹gauss›
  where
    ‹Re 1 = 1›
  | ‹Im 1 = 0›

primcorec plus_gauss :: ‹gauss ⇒ gauss ⇒ gauss›
  where
    ‹Re (x + y) = Re x + Re y›
  | ‹Im (x + y) = Im x + Im y›

primcorec uminus_gauss :: ‹gauss ⇒ gauss›
  where
    ‹Re (− x) = − Re x›
  | ‹Im (− x) = − Im x›

primcorec minus_gauss :: ‹gauss ⇒ gauss ⇒ gauss›
  where
    ‹Re (x − y) = Re x − Re y›
  | ‹Im (x − y) = Im x − Im y›

primcorec times_gauss :: ‹gauss ⇒ gauss ⇒ gauss›
  where
    ‹Re (x * y) = Re x * Re y − Im x * Im y›
  | ‹Im (x * y) = Re x * Im y + Im x * Re y›

instance
  by standard (simp_all add: gauss_eq_iff algebra_simps)

end

lemma of_nat_gauss:
  ‹of_nat n = Gauss (int n) 0›
  by (induction n) (simp_all add: gauss_eq_iff)

lemma numeral_gauss:
  ‹numeral n = Gauss (numeral n) 0›
proof −
  have ‹numeral n = (of_nat (numeral n) :: gauss)›
    by simp
  also have ‹... = Gauss (of_nat (numeral n)) 0›
    by (simp add: of_nat_gauss)
  finally show ?thesis
    by simp
qed

lemma of_int_gauss:
  ‹of_int k = Gauss k 0›
```

**by** (*simp add*: *gauss_eq_iff of_int_of_nat of_nat_gauss*)

**lemma** *conversion_simps* [*simp*]:
‹*Re* (*numeral m*) = *numeral m*›
‹*Im* (*numeral m*) = *0*›
‹*Re* (*of_nat n*) = *int n*›
‹*Im* (*of_nat n*) = *0*›
‹*Re* (*of_int k*) = *k*›
‹*Im* (*of_int k*) = *0*›
**by** (*simp_all add*: *numeral_gauss of_nat_gauss of_int_gauss*)

**lemma** *gauss_eq_0*:
‹*z* = *0* ⟷ (*Re z*)² + (*Im z*)² = *0*›
**by** (*simp add*: *gauss_eq_iff sum_power2_eq_zero_iff*)

**lemma** *gauss_neq_0*:
‹*z* ≠ *0* ⟷ (*Re z*)² + (*Im z*)² > *0*›
**by** (*simp add*: *gauss_eq_0 sum_power2_ge_zero less_le*)

**lemma** *Re_sum* [*simp*]:
‹*Re* (*sum f s*) = (∑ *x*∈*s*. *Re* (*f x*))›
**by** (*induct s rule*: *infinite_finite_induct*) *auto*

**lemma** *Im_sum* [*simp*]:
‹*Im* (*sum f s*) = (∑ *x*∈*s*. *Im* (*f x*))›
**by** (*induct s rule*: *infinite_finite_induct*) *auto*

**instance** *gauss* :: *idom*
**proof**
**fix** *x y* :: *gauss*
**assume** ‹*x* ≠ *0*› ‹*y* ≠ *0*›
**then show** ‹*x* * *y* ≠ *0*›
**by** (*simp_all add*: *gauss_eq_iff*)
(*smt* (*verit*, *best*) *mult_eq_0_iff mult_neg_neg mult_neg_pos mult_pos_neg mult_pos_pos*)
**qed**

## 9.2 The Gauss Number *i*

**primcorec** *imaginary_unit* :: *gauss* (‹i›)
**where**
‹*Re* i = *0*›
| ‹*Im* i = *1*›

**lemma** *Gauss_eq*:
‹*Gauss a b* = *of_int a* + i * *of_int b*›
**by** (*simp add*: *gauss_eq_iff*)

**lemma** *gauss_eq*:

‹$a = of_int\ (Re\ a) + \mathrm{i} * of_int\ (Im\ a)$›
**by** (*simp add*: *gauss_eq_iff*)

**lemma** *gauss_i_not_zero* [*simp*]:
‹$\mathrm{i} \neq 0$›
**by** (*simp add*: *gauss_eq_iff*)

**lemma** *gauss_i_not_one* [*simp*]:
‹$\mathrm{i} \neq 1$›
**by** (*simp add*: *gauss_eq_iff*)

**lemma** *gauss_i_not_numeral* [*simp*]:
‹$\mathrm{i} \neq numeral\ n$›
**by** (*simp add*: *gauss_eq_iff*)

**lemma** *gauss_i_not_neg_numeral* [*simp*]:
‹$\mathrm{i} \neq - numeral\ n$›
**by** (*simp add*: *gauss_eq_iff*)

**lemma** *i_mult_i_eq* [*simp*]:
‹$\mathrm{i} * \mathrm{i} = - 1$›
**by** (*simp add*: *gauss_eq_iff*)

**lemma** *gauss_i_mult_minus* [*simp*]:
‹$\mathrm{i} * (\mathrm{i} * x) = - x$›
**by** (*simp flip*: *mult.assoc*)

**lemma** *i_squared* [*simp*]:
‹$\mathrm{i}^2 = - 1$›
**by** (*simp add*: *power2_eq_square*)

**lemma** *i_even_power* [*simp*]:
‹$\mathrm{i} \mathbin{\hat{}} (n * 2) = (- 1) \mathbin{\hat{}} n$›
**unfolding** *mult.commute* [*of n*] *power_mult* **by** *simp*

**lemma** *Re_i_times* [*simp*]:
‹$Re\ (\mathrm{i} * z) = - Im\ z$›
**by** *simp*

**lemma** *Im_i_times* [*simp*]:
‹$Im\ (\mathrm{i} * z) = Re\ z$›
**by** *simp*

**lemma** *i_times_eq_iff*:
‹$\mathrm{i} * w = z \longleftrightarrow w = - (\mathrm{i} * z)$›
**by** *auto*

**lemma** *is_unit_i* [*simp*]:
‹$\mathrm{i}\ dvd\ 1$›

**by** (*rule dvdI* [*of* _ _ ‹− i›]) *simp*

**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›
**by** (*simp_all add*: *gauss_eq_iff*)

## 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*›
**by** (*simp add*: *gauss_eq_iff*)

**lemma** *gauss_cnj_cnj* [*simp*]:
‹*cnj (cnj z) = z*›
**by** (*simp add*: *gauss_eq_iff*)

**lemma** *gauss_cnj_zero* [*simp*]:
‹*cnj 0 = 0*›
**by** (*simp add*: *gauss_eq_iff*)

**lemma** *gauss_cnj_zero_iff* [*iff*]:
‹*cnj z = 0 ⟷ z = 0*›
**by** (*simp add*: *gauss_eq_iff*)

**lemma** *gauss_cnj_one_iff* [*simp*]:
‹*cnj z = 1 ⟷ z = 1*›
**by** (*simp add*: *gauss_eq_iff*)

**lemma** *gauss_cnj_add* [*simp*]:
‹*cnj (x + y) = cnj x + cnj y*›
**by** (*simp add*: *gauss_eq_iff*)

**lemma** *cnj_sum* [*simp*]:
‹*cnj (sum f s) = (∑ x∈s. cnj (f x))*›
**by** (*induct s rule*: *infinite_finite_induct*) *auto*

**lemma** *gauss_cnj_diff* [*simp*]:
‹*cnj (x − y) = cnj x − cnj y*›
**by** (*simp add*: *gauss_eq_iff*)

**lemma** *gauss_cnj_minus* [*simp*]:
‹*cnj (− x) = − cnj x*›
**by** (*simp add*: *gauss_eq_iff*)

**lemma** *gauss_cnj_one* [*simp*]:
‹*cnj 1 = 1*›
**by** (*simp add*: *gauss_eq_iff*)

**lemma** *gauss_cnj_mult* [*simp*]:
‹*cnj (x * y) = cnj x * cnj y*›
**by** (*simp add*: *gauss_eq_iff*)

**lemma** *cnj_prod* [*simp*]:
‹*cnj (prod f s) = (∏ x∈s. cnj (f x))*›
**by** (*induct s rule*: *infinite_finite_induct*) *auto*

**lemma** *gauss_cnj_power* [*simp*]:
‹*cnj (x ^ n) = cnj x ^ n*›
**by** (*induct n*) *simp_all*

**lemma** *gauss_cnj_numeral* [*simp*]:
‹*cnj (numeral w) = numeral w*›
**by** (*simp add*: *gauss_eq_iff*)

**lemma** *gauss_cnj_of_nat* [*simp*]:
‹*cnj (of_nat n) = of_nat n*›
**by** (*simp add*: *gauss_eq_iff*)

**lemma** *gauss_cnj_of_int* [*simp*]:

‹*cnj* (*of_int z*) = *of_int z*›
**by** (*simp add*: *gauss_eq_iff*)

**lemma** *gauss_cnj_i* [*simp*]:
‹*cnj* i = − i›
**by** (*simp add*: *gauss_eq_iff*)

**lemma** *gauss_add_cnj*:
‹*z* + *cnj z* = *of_int* (2 ∗ *Re z*)›
**by** (*simp add*: *gauss_eq_iff*)

**lemma** *gauss_diff_cnj*:
‹*z* − *cnj z* = *of_int* (2 ∗ *Im z*) ∗ i›
**by** (*simp add*: *gauss_eq_iff*)

**lemma** *gauss_mult_cnj*:
‹*z* ∗ *cnj z* = *of_int* ((*Re z*)$^2$ + (*Im z*)$^2$)›
**by** (*simp add*: *gauss_eq_iff power2_eq_square*)

**lemma** *cnj_add_mult_eq_Re*:
‹*z* ∗ *cnj w* + *cnj z* ∗ *w* = *of_int* (2 ∗ *Re* (*z* ∗ *cnj w*))›
**by** (*simp add*: *gauss_eq_iff*)

**lemma** *gauss_In_mult_cnj_zero* [*simp*]:
‹*Im* (*z* ∗ *cnj z*) = 0›
**by** *simp*

## 9.4 Algebraic division

**instantiation** *gauss* :: *idom_modulo*
**begin**

**primcorec** *divide_gauss* :: ‹*gauss* ⇒ *gauss* ⇒ *gauss*›
**where**
‹*Re* (*x div y*) = (*Re x* ∗ *Re y* + *Im x* ∗ *Im y*) *cdiv* ((*Re y*)$^2$ + (*Im y*)$^2$)›
| ‹*Im* (*x div y*) = (*Im x* ∗ *Re y* − *Re x* ∗ *Im y*) *cdiv* ((*Re y*)$^2$ + (*Im y*)$^2$)›

**primcorec** *modulo_gauss* :: ‹*gauss* ⇒ *gauss* ⇒ *gauss*›
**where**
‹*Re* (*x mod y*) = *Re x* −
((*Re x* ∗ *Re y* + *Im x* ∗ *Im y*) *cdiv* ((*Re y*)$^2$ + (*Im y*)$^2$) ∗ *Re y* −
(*Im x* ∗ *Re y* − *Re x* ∗ *Im y*) *cdiv* ((*Re y*)$^2$ + (*Im y*)$^2$) ∗ *Im y*)›
| ‹*Im* (*x mod y*) = *Im x* −
((*Re x* ∗ *Re y* + *Im x* ∗ *Im y*) *cdiv* ((*Re y*)$^2$ + (*Im y*)$^2$) ∗ *Im y* +
(*Im x* ∗ *Re y* − *Re x* ∗ *Im y*) *cdiv* ((*Re y*)$^2$ + (*Im y*)$^2$) ∗ *Re y*)›

**instance proof**
**fix** *x y* :: *gauss*
**show** ‹*x div* 0 = 0›

```
    by (simp add: gauss_eq_iff)
  show ‹x * y div y = x› if ‹y ≠ 0›
  proof −
    define Y where ‹Y = (Re y)^2 + (Im y)^2›
    moreover have ‹Y > 0›
      using that by (simp add: gauss_eq_0 less_le Y_def)
    have *: ‹Im y * (Im y * Re x) + Re x * (Re y * Re y) = Re x * Y›
      ‹Im x * (Im y * Im y) + Im x * (Re y * Re y) = Im x * Y›
      ‹(Im y)^2 + (Re y)^2 = Y›
      by (simp_all add: power2_eq_square algebra_simps Y_def)
    from ‹Y > 0› show ?thesis
    by (simp add: gauss_eq_iff algebra_simps) (simp add: * nonzero_mult_cdiv_cancel_right)
  qed
  show ‹x div y * y + x mod y = x›
    by (simp add: gauss_eq_iff)
qed

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
  show ‹euclidean_size (0::gauss) = 0›
    by (simp add: euclidean_size_gauss_def)
  show ‹euclidean_size (x mod y) < euclidean_size y› if ‹y ≠ 0› for x y :: gauss
  proof−
    define X and Y and R and I
      where ‹X = (Re x)^2 + (Im x)^2› and ‹Y = (Re y)^2 + (Im y)^2›
        and ‹R = Re x * Re y + Im x * Im y› and ‹I = Im x * Re y − Re x * Im
y›
    with that have ‹0 < Y› and rhs: ‹int (euclidean_size y) = Y›
      by (simp_all add: gauss_neq_0 euclidean_size_gauss_def)
    have ‹X * Y = R^2 + I^2›
      by (simp add: R_def I_def X_def Y_def power2_eq_square algebra_simps)
    let ?lhs = ‹X − I * (I cdiv Y) − R * (R cdiv Y)
        − I cdiv Y * (I cmod Y) − R cdiv Y * (R cmod Y)›
    have ‹?lhs = X + Y * (R cdiv Y) * (R cdiv Y) + Y * (I cdiv Y) * (I cdiv Y)
        − 2 * (R cdiv Y * R + I cdiv Y * I)›
      by (simp flip: minus_cmod_eq_mult_cdiv add: algebra_simps)
    also have ‹... = (Re (x mod y))^2 + (Im (x mod y))^2›
      by (simp add: X_def Y_def R_def I_def algebra_simps power2_eq_square)
    finally have lhs: ‹int (euclidean_size (x mod y)) = ?lhs›
      by (simp add: euclidean_size_gauss_def)
    have ‹?lhs * Y = (I cmod Y)^2 + (R cmod Y)^2›
      apply (simp add: algebra_simps power2_eq_square ‹X * Y = R^2 + I^2›)
```

**apply** (*simp flip*: *mult.assoc add.assoc minus_cmod_eq_mult_cdiv*)
**apply** (*simp add*: *algebra_simps*)
**done**
**also have** ‹$\ldots \leq (Y\ div\ 2)^2 + (Y\ div\ 2)^2$›
**by** (*rule add_mono*) (*use* ‹$Y > 0$› *abs_cmod_less_equal* [*of Y*] **in** ‹*simp_all add*: *power2_le_iff_abs_le*›)
**also have** ‹$\ldots < Y^2$›
**using** ‹$Y > 0$› **by** (*cases* ‹$Y = 1$›) (*simp_all add*: *power2_eq_square mult_le_less_imp_less flip*: *mult.assoc*)
**finally have** ‹*?lhs* $*$ $Y < Y^2$› **.**
**with** ‹$Y > 0$› **have** ‹*?lhs* $< Y$›
**by** (*simp add*: *power2_eq_square*)
**then have** ‹*int* (*euclidean_size* (*x mod y*)) $<$ *int* (*euclidean_size y*)›
**by** (*simp only*: *lhs rhs*)
**then show** *?thesis*
**by** *simp*
**qed**
**show** ‹*euclidean_size x* $\leq$ *euclidean_size* (*x* $*$ *y*)› **if** ‹$y \neq 0$› **for** *x y* :: *gauss*
**proof** −
**from** *that* **have** ‹*euclidean_size y* $> 0$›
**by** (*simp add*: *euclidean_size_gauss_def gauss_neq_0*)
**then have** ‹*euclidean_size x* $\leq$ *euclidean_size x* $*$ *euclidean_size y*›
**by** *simp*
**also have** ‹$\ldots$ = *nat* ($(Re\ x)^2 + (Im\ x)^2$) $*$ ($(Re\ y)^2 + (Im\ y)^2$))›
**by** (*simp add*: *euclidean_size_gauss_def nat_mult_distrib*)
**also have** ‹$\ldots$ = *euclidean_size* (*x* $*$ *y*)›
**by** (*simp add*: *euclidean_size_gauss_def eq_nat_nat_iff*) (*simp add*: *algebra_simps power2_eq_square*)
**finally show** *?thesis* **.**
**qed**
**qed**

**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*
**apply** (*tactic* ‹*ALLGOALS* (*CONVERSION*
(*Conv.arg_conv* (*Conv.arg1_conv* (*Semiring_Normalizer.semiring_normalize_conv*

***context*** )))))›)
**by** (*rule refl*)

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

**schematic_goal**
**fixes** $x$ :: *int*
**shows** $(x - (-2))\hat{}5 * (y - 78) \hat{} 8 = ?X$
**apply** (*tactic* ‹*ALLGOALS* (*CONVERSION*
(*Conv.arg_conv* (*Conv.arg1_conv* (*Semiring_Normalizer.semiring_normalize_conv* ***context*** )))))›)
**by** (*rule refl*)

**lemma** $((-3) \hat{} (Suc\ (Suc\ (Suc\ 0)))) == (X::'a::\{comm_ring_1\})$
**apply** (*simp only*: *power_Suc power_0*)
**apply** (*simp only*: *semiring_norm*)
**oops**

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

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

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

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

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

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

**lemma**
**fixes** $x::'a::idom$

**shows** $x^2*y = x^2$ & $x*y^2 = y^2 \longleftrightarrow x = 1$ & $y = 1 \mid x = 0$ & $y = 0$
**by** *algebra*

## 10.2 Lemmas for Lagrange's theorem

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

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

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

## 10.3 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* $(Ax * c - Ay * s, Ay * c + Ax * s)$
$(Bx * c - By * s, By * c + Bx * s)$ $(Cx * c - Cy * s, Cy * c + Cx * s)$
**using** *assms*
**by** (*algebra add*: *collinear_def split_def fst_conv snd_conv*)

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

**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.

**oracle** *iff_oracle* = ‹
*let*
*fun mk_iff 1 = Var ((P, 0), **typ** ‹bool›)*
*| mk_iff n = HOLogic.mk_eq (Var ((P, 0), **typ** ‹bool›), mk_iff (n − 1));*
*in*
*fn (thy, n) =>*
*if n > 0 andalso n mod 2 = 0*
*then Thm.global_cterm_of thy (HOLogic.mk_Trueprop (mk_iff n))*
*else raise Fail (iff_oracle: ^ string_of_int n)*
*end*
›

## 11.2 Oracle as low-level rule

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

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

These oracle calls had better fail.

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

›

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

### 11.3 Oracle as proof method

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

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

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

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

**lemma** *A*
**apply** (*iff 1*)*?*
**oops**

**end**

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

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

**by** *induction_schema* (*pat_completeness*, *lexicographic_order*)

**lemma** *minus_one_induct*:
⟦⋀n::nat. (n ≠ 0 ⟹ P (n − 1)) ⟹ P n⟧ ⟹ P x
**by** *induction_schema* (*pat_completeness*, *lexicographic_order*)

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

**lemma** *list_induct2'*:
⟦ P [] [];
⋀x xs. P (x#xs) [];
⋀y ys. P [] (y#ys);
⋀x xs y ys. P xs ys ⟹ P (x#xs) (y#ys) ⟧
⟹ P xs ys
**by** *induction_schema* (*pat_completeness*, *lexicographic_order*)

**theorem** *even_odd_induct*:
**assumes** R 0
**assumes** Q 0
**assumes** ⋀n. Q n ⟹ R (Suc n)
**assumes** ⋀n. R n ⟹ Q (Suc n)
**shows** R n Q n
**using** *assms*
**by** *induction_schema* (*pat_completeness*+, *lexicographic_order*)

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

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

**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 $a \leq f(a)$.

## 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.

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

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

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

language.

**theorem** *Knaster_Tarski'*:
  **fixes** $f :: 'a::complete_lattice \Rightarrow 'a$
  **assumes** *mono f*
  **shows** $\exists a.\ f\ a = a$
**proof**
  **let** *?H* $= \{u.\ f\ u \leq u\}$
  **let** *?a* $= \bigsqcap$ *?H*
  **show** *f ?a = ?a*
  **proof** (*rule order_antisym*)
    **show** *f ?a* $\leq$ *?a*
    **proof** (*rule Inf_greatest*)
      **fix** *x*
      **assume** $x \in$ *?H*
      **then have** *?a* $\leq x$ **by** (*rule Inf_lower*)
      **with** ‹*mono f*› **have** *f ?a* $\leq f\ x$ **..**
      **also from** ‹$x \in$ *?H*› **have** $\ldots \leq x$ **..**
      **finally show** *f ?a* $\leq x$ **.**
    **qed**
    **show** *?a* $\leq f$ *?a*
    **proof** (*rule Inf_lower*)
      **from** ‹*mono f*› **and** ‹*f ?a* $\leq$ *?a*› **have** *f* (*f ?a*) $\leq f$ *?a* **..**
      **then show** *f ?a* $\in$ *?H* **..**
    **qed**
  **qed**
**qed**

**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.

**ML** ‹*1 + 1*›

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

## 14.2 Antiquotations

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

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

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

**term** *1 + 1* — term within theory source

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

**ML** ‹***term*** ‹*1 + (1::int)*››

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

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

## 14.3 Recursive ML evaluation

**ML** ‹
*ML* ‹*ML* ‹*val a* = @{*thm refl*}››;
*ML* ‹*val b* = @{*thm sym*}›;
*val c* = @{*thm trans*}
*val thms* = [*a*, *b*, *c*];
›

## 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

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

## 14.5 Example: factorial and ackermann function in Isabelle/ML

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

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

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

**ML** ‹
*fun ackermann 0 n = n + 1*
*| ackermann m 0 = ackermann (m − 1) 1*
*| ackermann m n = ackermann (m − 1) (ackermann m (n − 1))*
›

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

## 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.

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

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

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

## 14.7 Function specifications in Isabelle/HOL

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

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

**declare** [[*ML_source_trace*]]
**ML** ‹***term*** ‹*factorial 4*›› — symbolic term in ML

**ML** ‹@{*code factorial*}› — ML code from function specification

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

**value** *ackermann 3 5*

**end**

# 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* ⟶ *B*) ⟶ *A*) ⟶ *A*
**proof**
  **assume** (*A* ⟶ *B*) ⟶ *A*
  **show** *A*
  **proof** (*rule classical*)
    **assume** ¬ *A*
    **have** *A* ⟶ *B*
    **proof**
      **assume** *A*
      **with** ‹¬ *A*› **show** *B* **by** *contradiction*
    **qed**
    **with** ‹(*A* ⟶ *B*) ⟶ *A*› **show** *A* **..**
  **qed**
**qed**

In the subsequent version the reasoning is rearranged by means of "weak assumptions" (as introduced by **presume**). Before assuming the negated goal ¬ *A*, its intended consequence *A* ⟶ *B* is put into place in order to solve the main problem. Nevertheless, we do not get anything for free, but

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

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 ⟶ B) ⟶ A) ⟶ A
proof
  assume (A ⟶ B) ⟶ A
  show A
  proof (rule classical)
    presume A ⟶ B
    with ‹(A ⟶ B) ⟶ A› show A ..
  next
    assume ¬ A
    show A ⟶ B
    proof
      assume A
      with ‹¬ A› show B by contradiction
    qed
  qed
qed
```

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

# 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* = (|*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*|)

### 16.1.1 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*|)

### 16.1.2 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*|)

### 16.1.3 Some lemmas about records

Basic simplifications.

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

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

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

**by** *simp*

Equality of records.

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

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

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

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

Surjective pairing

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

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

Representation of records by cases or (degenerate) induction.

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

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

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

**proof** (*cases r*)
  **fix** *xpos ypos more*
  **assume** *r* = (|*xpos* = *xpos*, *ypos* = *ypos*, ... = *more*|)
  **then show** *?thesis* **by** *simp*
**qed**

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

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

Concrete records are type instances of record schemes.

**definition** *foo5* :: *nat*
  **where** *foo5* = *getX* (|*xpos* = *1*, *ypos* = *0*|)

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

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

**lemma** *incX r* = *setX r* (*Suc* (*getX r*))
  **by** (*simp add*: *getX_def setX_def incX_def*)

An alternative definition.

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

## 16.2 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*|)

### 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*
  **by** *simp*

Only the most recent update to a component survives simplification.

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

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

```
lemma (∀r. P (xpos r)) ⟶ (∀x. P x)
  apply (tactic ‹simp_tac (put_simpset HOL_basic_ss context
    |> Simplifier.add_proc (Record.split_simproc (K ~1))) 1›)
  apply simp
  done

lemma (∀r. P (xpos r)) ⟶ (∀x. P x)
  apply (tactic ‹Record.split_simp_tac context [] (K ~1) 1›)
  apply simp
  done

lemma (∃r. P (xpos r)) ⟶ (∃x. P x)
  apply (tactic ‹simp_tac (put_simpset HOL_basic_ss context
    |> Simplifier.add_proc (Record.split_simproc (K ~1))) 1›)
  apply simp
  done

lemma (∃r. P (xpos r)) ⟶ (∃x. P x)
  apply (tactic ‹Record.split_simp_tac context [] (K ~1) 1›)
  apply simp
  done

lemma ⋀r. P (xpos r) ⟹ (∃x. P x)
  apply (tactic ‹simp_tac (put_simpset HOL_basic_ss context
    |> Simplifier.add_proc (Record.split_simproc (K ~1))) 1›)
  apply auto
  done

lemma ⋀r. P (xpos r) ⟹ (∃x. P x)
  apply (tactic ‹Record.split_simp_tac context [] (K ~1) 1›)
  apply auto
  done

lemma P (xpos r) ⟹ (∃x. P x)
  apply (tactic ‹Record.split_simp_tac context [] (K ~1) 1›)
  apply auto
  done

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

The effect of simproc `Record.ex_sel_eq_simproc` is illustrated by the fol-

lowing lemma.

**lemma** $\exists r.\ xpos\ r = x$
  **supply** [[*simproc add*: *Record.ex_sel_eq*]]
  **apply** (*simp*)
  **done**

## 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** $r(\!|b := x,\ a := y|\!) = ?X$
  **by** *simp*

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

**schematic_goal** $r(\!|b := y,\ a := x|\!) = ?X$
  **supply** [[*record_sort_updates*]]
  **by** *simp*

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** $r(\!|f := x1,\ a := x2|\!) = r(\!|a := x2,\ f := x1|\!)$
  **by** *simp*

**lemma** $r(\!|f := x1,\ a := x2|\!) = r(\!|a := x2,\ f := x1|\!)$
  **supply** [[*simproc del*: *Record.eq*]]

**apply** (*simp?*)
**oops**

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

**lemma** $r(\!|f := x1,\ a{:=}\ x2|\!) = r(\!|a := x2,\ f{:=}\ x1|\!)$
**supply** [[*record_sort_updates*, *simproc del*: *Record.eq*]]
**apply** *simp*
**done**

**schematic_goal** $r(\!|f := x1,\ e := x2,\ d{:=}\ x3,\ c{:=}\ x4,\ b{:=}x5,\ a{:=}\ x6|\!) = ?X$
**supply** [[*record_sort_updates*]]
**by** *simp*

**schematic_goal** $r(\!|f := x1,\ e := x2,\ d{:=}\ x3,\ c{:=}\ x4,\ e{:=}x5,\ a{:=}\ x6|\!) = ?X$
**supply** [[*record_sort_updates*]]
**by** *simp*

**schematic_goal** $r(\!|f := x1,\ e := x2,\ d{:=}\ x3,\ c{:=}\ x4,\ e{:=}x5,\ a{:=}\ x6|\!) = ?X$
**by** *simp*

## 16.5 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** $('a,\ 'b,\ 'c)\ bar$

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

**setup** ‹
*let*
*val N = 300*
*in*
*Record.add_record* {*overloaded* = *false*} ([], ***binding***‹*large_record*›) *NONE*

```
    (map (fn i => (Binding.make (fld_ ^ string_of_int i, here), @{typ nat},
Mixfix.NoSyn))
      (1 upto N))
  end
›

declare [[record_codegen]]

schematic_goal ‹fld_1 (r(|fld_300 := x300, fld_20 := x20, fld_200 := x200|))
= ?X›
  by simp

schematic_goal ‹r(|fld_300 := x300, fld_20 := x20, fld_200 := x200|) = ?X›
  supply [[record_sort_updates]]
  by simp

end
```

# 17 Examples for generic reflection and reification

**theory** *Reflection_Examples*
**imports** *Complex_Main HOL−Library.Reflection*
**begin**

This theory presents two methods: reify and reflection

Consider an HOL type $\sigma$, the structure of which is not recongnisable on the theory level. This is the case of *bool*, arithmetical terms such as *int*, *real* etc ... In order to implement a simplification on terms of type $\sigma$ we often need its structure. Traditionnaly such simplifications are written in ML, proofs are synthesized.

An other strategy is to declare an HOL datatype $\tau$ and an HOL function (the interpretation) that maps elements of $\tau$ to elements of $\sigma$.

The functionality of *reify* then is, given a term $t$ of type $\sigma$, to compute a term $s$ of type $\tau$. For this it needs equations for the interpretation.

N.B: All the interpretations supported by *reify* must have the type *'a list* $\Rightarrow \tau \Rightarrow \sigma$. The method *reify* can also be told which subterm of the current subgoal should be reified. The general call for *reify* is *reify eqs* $(t)$, where *eqs* are the defining equations of the interpretation and $(t)$ is an optional parameter which specifies the subterm to which reification should be applied to. If $(t)$ is abscent, *reify* tries to reify the whole subgoal.

The method *reflection* uses *reify* and has a very similar signature: *reflection corr_thm eqs* $(t)$. Here again *eqs* and $(t)$ are as described above and *corr_thm* is a theorem proving *I vs* $(f\ t)$ = *I vs t*. We assume that $I$ is the interpretation and $f$ is some useful and executable simplification of type $\tau \Rightarrow \tau$. The method *reflection* applies reification and hence the theorem $t =$

*I xs s* and hence using *corr_thm* derives $t = I\ xs\ (f\ s)$. It then uses normalization by equational rewriting to prove $f\ s = s'$ which almost finishes the proof of $t = t'$ where $I\ xs\ s' = t'$.

Example 1 : Propositional formulae and NNF.

The type *fm* represents simple propositional formulae:

```
datatype form = TrueF | FalseF | Less nat nat
  | And form form | Or form form | Neg form | ExQ form

primrec interp :: form ⇒ ('a::ord) list ⇒ bool
where
  interp TrueF vs ⟷ True
| interp FalseF vs ⟷ False
| interp (Less i j) vs ⟷ vs ! i < vs ! j
| interp (And f1 f2) vs ⟷ interp f1 vs ∧ interp f2 vs
| interp (Or f1 f2) vs ⟷ interp f1 vs ∨ interp f2 vs
| interp (Neg f) vs ⟷ ¬ interp f vs
| interp (ExQ f) vs ⟷ (∃ v. interp f (v # vs))

lemmas interp_reify_eqs = interp.simps
declare interp_reify_eqs [reify]

lemma ∃ x. x < y ∧ x < z
  apply reify
  oops

datatype fm = And fm fm | Or fm fm | Imp fm fm | Iff fm fm | Not fm | At nat

primrec Ifm :: fm ⇒ bool list ⇒ bool
where
  Ifm (At n) vs ⟷ vs ! n
| Ifm (And p q) vs ⟷ Ifm p vs ∧ Ifm q vs
| Ifm (Or p q) vs ⟷ Ifm p vs ∨ Ifm q vs
| Ifm (Imp p q) vs ⟷ Ifm p vs ⟶ Ifm q vs
| Ifm (Iff p q) vs ⟷ Ifm p vs = Ifm q vs
| Ifm (Not p) vs ⟷ ¬ Ifm p vs

lemma Q ⟶ (D ∧ F ∧ ((¬ D) ∧ (¬ F)))
  apply (reify Ifm.simps)
oops
```

Method *reify* maps a *bool* to an *fm*. For this it needs the semantics of *fm*, i.e. the rewrite rules in *Ifm.simps*.

You can also just pick up a subterm to reify.

```
lemma Q ⟶ (D ∧ F ∧ ((¬ D) ∧ (¬ F)))
  apply (reify Ifm.simps (((¬ D) ∧ (¬ F))))
oops
```

Let's perform NNF. This is a version that tends to generate disjunctions

**primrec** *fmsize* :: *fm* ⇒ *nat*
**where**
  *fmsize* (*At n*) = *1*
| *fmsize* (*Not p*) = *1* + *fmsize p*
| *fmsize* (*And p q*) = *1* + *fmsize p* + *fmsize q*
| *fmsize* (*Or p q*) = *1* + *fmsize p* + *fmsize q*
| *fmsize* (*Imp p q*) = *2* + *fmsize p* + *fmsize q*
| *fmsize* (*Iff p q*) = *2* + *2*∗ *fmsize p* + *2*∗ *fmsize q*

**lemma** [*measure_function*]: *is_measure fmsize* **..**

**fun** *nnf* :: *fm* ⇒ *fm*
**where**
  *nnf* (*At n*) = *At n*
| *nnf* (*And p q*) = *And* (*nnf p*) (*nnf q*)
| *nnf* (*Or p q*) = *Or* (*nnf p*) (*nnf q*)
| *nnf* (*Imp p q*) = *Or* (*nnf* (*Not p*)) (*nnf q*)
| *nnf* (*Iff p q*) = *Or* (*And* (*nnf p*) (*nnf q*)) (*And* (*nnf* (*Not p*)) (*nnf* (*Not q*)))
| *nnf* (*Not* (*And p q*)) = *Or* (*nnf* (*Not p*)) (*nnf* (*Not q*))
| *nnf* (*Not* (*Or p q*)) = *And* (*nnf* (*Not p*)) (*nnf* (*Not q*))
| *nnf* (*Not* (*Imp p q*)) = *And* (*nnf p*) (*nnf* (*Not q*))
| *nnf* (*Not* (*Iff p q*)) = *Or* (*And* (*nnf p*) (*nnf* (*Not q*))) (*And* (*nnf* (*Not p*)) (*nnf q*))
| *nnf* (*Not* (*Not p*)) = *nnf p*
| *nnf* (*Not p*) = *Not p*

The correctness theorem of *nnf*: it preserves the semantics of *fm*

**lemma** *nnf* [*reflection*]:
  *Ifm* (*nnf p*) *vs* = *Ifm p vs*
  **by** (*induct p rule*: *nnf.induct*) *auto*

Now let's perform NNF using our *nnf* function defined above. First to the whole subgoal.

**lemma** *A* ≠ *B* ∧ (*B* ⟶ *A* ≠ (*B* ∨ *C* ∧ (*B* ⟶ *A* ∨ *D*))) ⟶ *A* ∨ *B* ∧ *D*
  **apply** (*reflection Ifm.simps*)
**oops**

Now we specify on which subterm it should be applied

**lemma** *A* ≠ *B* ∧ (*B* ⟶ *A* ≠ (*B* ∨ *C* ∧ (*B* ⟶ *A* ∨ *D*))) ⟶ *A* ∨ *B* ∧ *D*
  **apply** (*reflection Ifm.simps only*: *B* ∨ *C* ∧ (*B* ⟶ *A* ∨ *D*))
**oops**

Example 2: Simple arithmetic formulae

The type *num* reflects linear expressions over natural number

**datatype** *num* = *C nat* | *Add num num* | *Mul nat num* | *Var nat* | *CN nat nat num*

This is just technical to make recursive definitions easier.

**primrec** *num_size* :: *num* ⇒ *nat*
**where**
  *num_size* (*C c*) = *1*
| *num_size* (*Var n*) = *1*
| *num_size* (*Add a b*) = *1* + *num_size a* + *num_size b*
| *num_size* (*Mul c a*) = *1* + *num_size a*
| *num_size* (*CN n c a*) = *4* + *num_size a*

**lemma** [*measure_function*]: *is_measure num_size* **..**

The semantics of num

**primrec** *Inum*:: *num* ⇒ *nat list* ⇒ *nat*
**where**
  *Inum_C* : *Inum* (*C i*) *vs* = *i*
| *Inum_Var*: *Inum* (*Var n*) *vs* = *vs*!*n*
| *Inum_Add*: *Inum* (*Add s t*) *vs* = *Inum s vs* + *Inum t vs*
| *Inum_Mul*: *Inum* (*Mul c t*) *vs* = *c* ∗ *Inum t vs*
| *Inum_CN* : *Inum* (*CN n c t*) *vs* = *c*∗(*vs*!*n*) + *Inum t vs*

Let's reify some nat expressions ...

**lemma** *4* ∗ (*2* ∗ *x* + (*y*::*nat*)) + *f a* ≠ *0*
  **apply** (*reify Inum.simps* (*4* ∗ (*2* ∗ *x* + (*y*::*nat*)) + *f a*))
**oops**

We're in a bad situation! $x$, $y$ and $f$ have been recongnized as constants, which is correct but does not correspond to our intuition of the constructor C. It should encapsulate constants, i.e. numbers, i.e. numerals.

So let's leave the *Inum_C* equation at the end and see what happens ...

**lemma** *4* ∗ (*2* ∗ *x* + (*y*::*nat*)) ≠ *0*
  **apply** (*reify Inum_Var Inum_Add Inum_Mul Inum_CN Inum_C* (*4* ∗ (*2* ∗ *x* + (*y*::*nat*))))
**oops**

Hm, let's specialize *Inum_C* with numerals.

**lemma** *Inum_number*: *Inum* (*C* (*numeral t*)) *vs* = *numeral t* **by** *simp*
**lemmas** *Inum_eqs* = *Inum_Var Inum_Add Inum_Mul Inum_CN Inum_number*

Second attempt

**lemma** *1* ∗ (*2* ∗ *x* + (*y*::*nat*)) ≠ *0*
  **apply** (*reify Inum_eqs* (*1* ∗ (*2* ∗ *x* + (*y*::*nat*))))
**oops**

That was fine, so let's try another one ...

**lemma** *1* ∗ (*2* ∗ *x* + (*y*::*nat*) + *0* + *1*) ≠ *0*
  **apply** (*reify Inum_eqs* (*1* ∗ (*2* ∗ *x* + (*y*::*nat*) + *0* + *1*)))

**oops**

Oh!! 0 is not a variable ... Oh! 0 is not a *numeral* ... thing. The same for 1. So let's add those equations, too.

**lemma** *Inum_01*: *Inum (C 0) vs = 0 Inum (C 1) vs = 1 Inum (C(Suc n)) vs = Suc n*
**by** *simp_all*

**lemmas** *Inum_eqs'= Inum_eqs Inum_01*

Third attempt:

**lemma** *1 * (2 * x + (y::nat) + 0 + 1) ≠ 0*
**apply** (*reify Inum_eqs' (1 * (2 * x + (y::nat) + 0 + 1))*)
**oops**

Okay, let's try reflection. Some simplifications on *Reflection_Examples.num* follow. You can skim until the main theorem *linum*.

**fun** *lin_add* :: *num ⇒ num ⇒ num*
**where**
*lin_add (CN n1 c1 r1) (CN n2 c2 r2) =*
*(if n1 = n2 then*
*(let c = c1 + c2*
*in (if c = 0 then lin_add r1 r2 else CN n1 c (lin_add r1 r2)))*
*else if n1 ≤ n2 then (CN n1 c1 (lin_add r1 (CN n2 c2 r2)))*
*else (CN n2 c2 (lin_add (CN n1 c1 r1) r2)))*
| *lin_add (CN n1 c1 r1) t = CN n1 c1 (lin_add r1 t)*
| *lin_add t (CN n2 c2 r2) = CN n2 c2 (lin_add t r2)*
| *lin_add (C b1) (C b2) = C (b1 + b2)*
| *lin_add a b = Add a b*

**lemma** *lin_add*:
*Inum (lin_add t s) bs = Inum (Add t s) bs*
**apply** (*induct t s rule*: *lin_add.induct*, *simp_all add*: *Let_def*)
**apply** (*case_tac c1+c2 = 0*,*case_tac n1 ≤ n2*, *simp_all*)
**apply** (*case_tac n1 = n2*, *simp_all add*: *algebra_simps*)
**done**

**fun** *lin_mul* :: *num ⇒ nat ⇒ num*
**where**
*lin_mul (C j) i = C (i * j)*
| *lin_mul (CN n c a) i = (if i=0 then (C 0) else CN n (i * c) (lin_mul a i))*
| *lin_mul t i = (Mul i t)*

**lemma** *lin_mul*:
*Inum (lin_mul t i) bs = Inum (Mul i t) bs*
**by** (*induct t i rule*: *lin_mul.induct*) (*auto simp add*: *algebra_simps*)

**fun** *linum*:: *num ⇒ num*

**where**
  *linum* (*C b*) = *C b*
| *linum* (*Var n*) = *CN n 1* (*C 0*)
| *linum* (*Add t s*) = *lin_add* (*linum t*) (*linum s*)
| *linum* (*Mul c t*) = *lin_mul* (*linum t*) *c*
| *linum* (*CN n c t*) = *lin_add* (*linum* (*Mul c* (*Var n*))) (*linum t*)

**lemma** *linum* [*reflection*]:
  *Inum* (*linum t*) *bs* = *Inum t bs*
  **by** (*induct t rule*: *linum.induct*) (*simp_all add*: *lin_mul lin_add*)

Now we can use linum to simplify nat terms using reflection

**lemma** *Suc* (*Suc 1*) ∗ (*x* + *Suc 1* ∗ *y*) = *3* ∗ *x* + *6* ∗ *y*
  **apply** (*reflection Inum_eqs′ only*: *Suc* (*Suc 1*) ∗ (*x* + *Suc 1* ∗ *y*))
**oops**

Let's lift this to formulae and see what happens

**datatype** *aform* = *Lt num num* | *Eq num num* | *Ge num num* | *NEq num num*
|
  *Conj aform aform* | *Disj aform aform* | *NEG aform* | *T* | *F*

**primrec** *linaformsize*:: *aform* ⇒ *nat*
**where**
  *linaformsize T* = *1*
| *linaformsize F* = *1*
| *linaformsize* (*Lt a b*) = *1*
| *linaformsize* (*Ge a b*) = *1*
| *linaformsize* (*Eq a b*) = *1*
| *linaformsize* (*NEq a b*) = *1*
| *linaformsize* (*NEG p*) = *2* + *linaformsize p*
| *linaformsize* (*Conj p q*) = *1* + *linaformsize p* + *linaformsize q*
| *linaformsize* (*Disj p q*) = *1* + *linaformsize p* + *linaformsize q*

**lemma** [*measure_function*]: *is_measure linaformsize* **..**

**primrec** *is_aform* :: *aform* => *nat list* => *bool*
**where**
  *is_aform T vs* = *True*
| *is_aform F vs* = *False*
| *is_aform* (*Lt a b*) *vs* = (*Inum a vs* < *Inum b vs*)
| *is_aform* (*Eq a b*) *vs* = (*Inum a vs* = *Inum b vs*)
| *is_aform* (*Ge a b*) *vs* = (*Inum a vs* ≥ *Inum b vs*)
| *is_aform* (*NEq a b*) *vs* = (*Inum a vs* ≠ *Inum b vs*)
| *is_aform* (*NEG p*) *vs* = (¬ (*is_aform p vs*))
| *is_aform* (*Conj p q*) *vs* = (*is_aform p vs* ∧ *is_aform q vs*)
| *is_aform* (*Disj p q*) *vs* = (*is_aform p vs* ∨ *is_aform q vs*)

Let's reify and do reflection

**lemma** (*3*::*nat*) ∗ *x* + *t* < *0* ∧ (*2* ∗ *x* + *y* ≠ *17*)

**apply** (*reify Inum_eqs′ is_aform.simps*)
**oops**

Note that reification handles several interpretations at the same time

**lemma** (*3*::*nat*) ∗ *x* + *t* < *0* ∧ *x* ∗ *x* + *t* ∗ *x* + *3* + *1* = *z* ∗ *t* ∗ *4* ∗ *z* ∨ *x* + *x* + *1* < *0*
**apply** (*reflection Inum_eqs′ is_aform.simps only*: *x* + *x* + *1*)
**oops**

For reflection we now define a simple transformation on aform: NNF + linum on atoms

**fun** *linaform*:: *aform* ⇒ *aform*
**where**
*linaform* (*Lt s t*) = *Lt* (*linum s*) (*linum t*)
| *linaform* (*Eq s t*) = *Eq* (*linum s*) (*linum t*)
| *linaform* (*Ge s t*) = *Ge* (*linum s*) (*linum t*)
| *linaform* (*NEq s t*) = *NEq* (*linum s*) (*linum t*)
| *linaform* (*Conj p q*) = *Conj* (*linaform p*) (*linaform q*)
| *linaform* (*Disj p q*) = *Disj* (*linaform p*) (*linaform q*)
| *linaform* (*NEG T*) = *F*
| *linaform* (*NEG F*) = *T*
| *linaform* (*NEG* (*Lt a b*)) = *Ge a b*
| *linaform* (*NEG* (*Ge a b*)) = *Lt a b*
| *linaform* (*NEG* (*Eq a b*)) = *NEq a b*
| *linaform* (*NEG* (*NEq a b*)) = *Eq a b*
| *linaform* (*NEG* (*NEG p*)) = *linaform p*
| *linaform* (*NEG* (*Conj p q*)) = *Disj* (*linaform* (*NEG p*)) (*linaform* (*NEG q*))
| *linaform* (*NEG* (*Disj p q*)) = *Conj* (*linaform* (*NEG p*)) (*linaform* (*NEG q*))
| *linaform p* = *p*

**lemma** *linaform*: *is_aform* (*linaform p*) *vs* = *is_aform p vs*
**by** (*induct p rule*: *linaform.induct*) (*auto simp add*: *linum*)

**lemma** (*Suc* (*Suc* (*Suc 0*)) ∗ ((*x*::*nat*) + *Suc* (*Suc 0*)) + *Suc* (*Suc* (*Suc 0*)) ∗
(*Suc* (*Suc* (*Suc 0*))) ∗ ((*x*::*nat*) + *Suc* (*Suc 0*))) < *0* ∧ *Suc 0* + *Suc 0* < *0*
**apply** (*reflection Inum_eqs′ is_aform.simps rules*: *linaform*)
**oops**

**declare** *linaform* [*reflection*]

**lemma** (*Suc* (*Suc* (*Suc 0*)) ∗ ((*x*::*nat*) + *Suc* (*Suc 0*)) + *Suc* (*Suc* (*Suc 0*)) ∗
(*Suc* (*Suc* (*Suc 0*))) ∗ ((*x*::*nat*) + *Suc* (*Suc 0*))) < *0* ∧ *Suc 0* + *Suc 0* < *0*
**apply** (*reflection Inum_eqs′ is_aform.simps*)
**oops**

We now give an example where interpretaions have zero or more than only one envornement of different types and show that automatic reification also deals with bindings

**datatype** *rb* = *BC bool* | *BAnd rb rb* | *BOr rb rb*

**primrec** *Irb* :: *rb* ⇒ *bool*
**where**
  *Irb* (*BC p*) ⟷ *p*
| *Irb* (*BAnd s t*) ⟷ *Irb s* ∧ *Irb t*
| *Irb* (*BOr s t*) ⟷ *Irb s* ∨ *Irb t*

**lemma** *A* ∧ (*B* ∨ *D* ∧ *B*) ∧ *A* ∧ (*B* ∨ *D* ∧ *B*) ∨ *A* ∧ (*B* ∨ *D* ∧ *B*) ∨ *A* ∧ (*B* ∨ *D* ∧ *B*)
  **apply** (*reify Irb.simps*)
**oops**

**datatype** *rint* = *IC int* | *IVar nat* | *IAdd rint rint* | *IMult rint rint*
  | *INeg rint* | *ISub rint rint*

**primrec** *Irint* :: *rint* ⇒ *int list* ⇒ *int*
**where**
  *Irint_Var*: *Irint* (*IVar n*) *vs* = *vs* ! *n*
| *Irint_Neg*: *Irint* (*INeg t*) *vs* = − *Irint t vs*
| *Irint_Add*: *Irint* (*IAdd s t*) *vs* = *Irint s vs* + *Irint t vs*
| *Irint_Sub*: *Irint* (*ISub s t*) *vs* = *Irint s vs* − *Irint t vs*
| *Irint_Mult*: *Irint* (*IMult s t*) *vs* = *Irint s vs* ∗ *Irint t vs*
| *Irint_C*: *Irint* (*IC i*) *vs* = *i*

**lemma** *Irint_C0*: *Irint* (*IC 0*) *vs* = *0*
  **by** *simp*

**lemma** *Irint_C1*: *Irint* (*IC 1*) *vs* = *1*
  **by** *simp*

**lemma** *Irint_Cnumeral*: *Irint* (*IC* (*numeral x*)) *vs* = *numeral x*
  **by** *simp*

**lemmas** *Irint_simps* = *Irint_Var Irint_Neg Irint_Add Irint_Sub Irint_Mult Irint_C0 Irint_C1 Irint_Cnumeral*

**lemma** (*3*::*int*) ∗ *x* + *y* ∗ *y* − *9* + (− *z*) = *0*
  **apply** (*reify Irint_simps* ((*3*::*int*) ∗ *x* + *y* ∗ *y* − *9* + (− *z*)))
  **oops**

**datatype** *rlist* = *LVar nat* | *LEmpty* | *LCons rint rlist* | *LAppend rlist rlist*

**primrec** *Irlist* :: *rlist* ⇒ *int list* ⇒ *int list list* ⇒ *int list*
**where**
  *Irlist* (*LEmpty*) *is vs* = []
| *Irlist* (*LVar n*) *is vs* = *vs* ! *n*
| *Irlist* (*LCons i t*) *is vs* = *Irint i is* # *Irlist t is vs*
| *Irlist* (*LAppend s t*) *is vs* = *Irlist s is vs* @ *Irlist t is vs*

**lemma** $[(1::int)] = []$
**apply** (*reify Irlist.simps Irint_simps* ([1] :: *int list*))
**oops**

**lemma** $([(3::int) * x + y * y - 9 + (- z)] @ []) @ xs = [y * y - z - 9 + (3::int) * x]$
**apply** (*reify Irlist.simps Irint_simps* (([(3::*int*) * *x* + *y* * *y* − 9 + (− *z*)] @ []) @ *xs*))
**oops**

**datatype** *rnat* = *NC nat*| *NVar nat*| *NSuc rnat* | *NAdd rnat rnat* | *NMult rnat rnat*
| *NNeg rnat* | *NSub rnat rnat* | *Nlgth rlist*

**primrec** *Irnat* :: *rnat* ⇒ *int list* ⇒ *int list list* ⇒ *nat list* ⇒ *nat*
**where**
*Irnat_Suc*: *Irnat* (*NSuc t*) *is ls vs* = *Suc* (*Irnat t is ls vs*)
| *Irnat_Var*: *Irnat* (*NVar n*) *is ls vs* = *vs* ! *n*
| *Irnat_Neg*: *Irnat* (*NNeg t*) *is ls vs* = *0*
| *Irnat_Add*: *Irnat* (*NAdd s t*) *is ls vs* = *Irnat s is ls vs* + *Irnat t is ls vs*
| *Irnat_Sub*: *Irnat* (*NSub s t*) *is ls vs* = *Irnat s is ls vs* − *Irnat t is ls vs*
| *Irnat_Mult*: *Irnat* (*NMult s t*) *is ls vs* = *Irnat s is ls vs* ∗ *Irnat t is ls vs*
| *Irnat_lgth*: *Irnat* (*Nlgth rxs*) *is ls vs* = *length* (*Irlist rxs is ls*)
| *Irnat_C*: *Irnat* (*NC i*) *is ls vs* = *i*

**lemma** *Irnat_C0*: *Irnat* (*NC 0*) *is ls vs* = *0*
**by** *simp*

**lemma** *Irnat_C1*: *Irnat* (*NC 1*) *is ls vs* = *1*
**by** *simp*

**lemma** *Irnat_Cnumeral*: *Irnat* (*NC* (*numeral x*)) *is ls vs* = *numeral x*
**by** *simp*

**lemmas** *Irnat_simps* = *Irnat_Suc Irnat_Var Irnat_Neg Irnat_Add Irnat_Sub Irnat_Mult Irnat_lgth*
*Irnat_C0 Irnat_C1 Irnat_Cnumeral*

**lemma** *Suc n* ∗ *length* (([(3::*int*) * *x* + *y* * *y* − 9 + (− *z*)] @ []) @ *xs*) = *length xs*
**apply** (*reify Irnat_simps Irlist.simps Irint_simps*
(*Suc n* ∗ *length* (([(3::*int*) * *x* + *y* * *y* − 9 + (− *z*)] @ []) @ *xs*)))
**oops**

**datatype** *rifm* = *RT* | *RF* | *RVar nat*
| *RNLT rnat rnat* | *RNILT rnat rint* | *RNEQ rnat rnat*
| *RAnd rifm rifm* | *ROr rifm rifm* | *RImp rifm rifm*| *RIff rifm rifm*
| *RNEX rifm* | *RIEX rifm* | *RLEX rifm* | *RNALL rifm* | *RIALL rifm* | *RLALL rifm*

| *RBEX rifm* | *RBALL rifm*

**primrec** *Irifm* :: *rifm* ⇒ *bool list* ⇒ *int list* ⇒ (*int list*) *list* ⇒ *nat list* ⇒ *bool*
**where**
*Irifm RT ps is ls ns* ⟷ *True*
| *Irifm RF ps is ls ns* ⟷ *False*
| *Irifm* (*RVar n*) *ps is ls ns* ⟷ *ps* ! *n*
| *Irifm* (*RNLT s t*) *ps is ls ns* ⟷ *Irnat s is ls ns* < *Irnat t is ls ns*
| *Irifm* (*RNILT s t*) *ps is ls ns* ⟷ *int* (*Irnat s is ls ns*) < *Irint t is*
| *Irifm* (*RNEQ s t*) *ps is ls ns* ⟷ *Irnat s is ls ns* = *Irnat t is ls ns*
| *Irifm* (*RAnd p q*) *ps is ls ns* ⟷ *Irifm p ps is ls ns* ∧ *Irifm q ps is ls ns*
| *Irifm* (*ROr p q*) *ps is ls ns* ⟷ *Irifm p ps is ls ns* ∨ *Irifm q ps is ls ns*
| *Irifm* (*RImp p q*) *ps is ls ns* ⟷ *Irifm p ps is ls ns* ⟶ *Irifm q ps is ls ns*
| *Irifm* (*RIff p q*) *ps is ls ns* ⟷ *Irifm p ps is ls ns* = *Irifm q ps is ls ns*
| *Irifm* (*RNEX p*) *ps is ls ns* ⟷ (∃ *x*. *Irifm p ps is ls* (*x* # *ns*))
| *Irifm* (*RIEX p*) *ps is ls ns* ⟷ (∃ *x*. *Irifm p ps* (*x* # *is*) *ls ns*)
| *Irifm* (*RLEX p*) *ps is ls ns* ⟷ (∃ *x*. *Irifm p ps is* (*x* # *ls*) *ns*)
| *Irifm* (*RBEX p*) *ps is ls ns* ⟷ (∃ *x*. *Irifm p* (*x* # *ps*) *is ls ns*)
| *Irifm* (*RNALL p*) *ps is ls ns* ⟷ (∀ *x*. *Irifm p ps is ls* (*x*#*ns*))
| *Irifm* (*RIALL p*) *ps is ls ns* ⟷ (∀ *x*. *Irifm p ps* (*x* # *is*) *ls ns*)
| *Irifm* (*RLALL p*) *ps is ls ns* ⟷ (∀ *x*. *Irifm p ps is* (*x*#*ls*) *ns*)
| *Irifm* (*RBALL p*) *ps is ls ns* ⟷ (∀ *x*. *Irifm p* (*x* # *ps*) *is ls ns*)

**lemma** ∀ *x*. ∃ *n*. ((*Suc n*) ∗ *length* (([(*3*::*int*) ∗ *x* + *f t* ∗ *y* − *9* + (− *z*)] @ []) @ *xs*) = *length xs*) ∧ *m* < *5*∗*n* − *length* (*xs* @ [*2*,*3*,*4*,*x*∗*z* + *8* − *y*]) ⟶ (∃ *p*. ∀ *q*. *p* ∧ *q* ⟶ *r*)
**apply** (*reify Irifm.simps Irnat_simps Irlist.simps Irint_simps*)
**oops**

An example for equations containing type variables

**datatype** *prod* = *Zero* | *One* | *Var nat* | *Mul prod prod*
| *Pw prod nat* | *PNM nat nat prod*

**primrec** *Iprod* :: *prod* ⇒ ('*a*::*linordered_idom*) *list* ⇒'*a*
**where**
*Iprod Zero vs* = *0*
| *Iprod One vs* = *1*
| *Iprod* (*Var n*) *vs* = *vs* ! *n*
| *Iprod* (*Mul a b*) *vs* = *Iprod a vs* ∗ *Iprod b vs*
| *Iprod* (*Pw a n*) *vs* = *Iprod a vs* ^ *n*
| *Iprod* (*PNM n k t*) *vs* = (*vs* ! *n*) ^ *k* ∗ *Iprod t vs*

**datatype** *sgn* = *Pos prod* | *Neg prod* | *ZeroEq prod* | *NZeroEq prod* | *Tr* | *F*
| *Or sgn sgn* | *And sgn sgn*

**primrec** *Isgn* :: *sgn* ⇒ ('*a*::*linordered_idom*) *list* ⇒ *bool*
**where**
*Isgn Tr vs* ⟷ *True*
| *Isgn F vs* ⟷ *False*

```
| Isgn (ZeroEq t) vs ⟷ Iprod t vs = 0
| Isgn (NZeroEq t) vs ⟷ Iprod t vs ≠ 0
| Isgn (Pos t) vs ⟷ Iprod t vs > 0
| Isgn (Neg t) vs ⟷ Iprod t vs < 0
| Isgn (And p q) vs ⟷ Isgn p vs ∧ Isgn q vs
| Isgn (Or p q) vs ⟷ Isgn p vs ∨ Isgn q vs

lemmas eqs = Isgn.simps Iprod.simps

lemma (x::'a::{linordered_idom}) ^ 4 * y * z * y ^ 2 * z ^ 23 > 0
  apply (reify eqs)
  oops

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

# 18 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)
by (rewrite at a + b add.commute)
   (rule assms)


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
  by (rewrite in f _ + f ¤ = _ diff_self) fact

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
  by (rewrite at f (_ + ¤) + f _ = _ diff_self) fact

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
  by (rewrite in f (¤ + _) + _ = _ diff_self) fact
```

**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*
**by** (*rewrite* **in** *f (_ + ¤) + _ = _ diff_self*) *fact*

**lemma**
**fixes** *x y* :: *nat*
**shows***x + y > c ⟹ y + x > c*
**by** (*rewrite at ¤ > c add.commute*) *assumption*

**lemma**
**fixes** *x y* :: *nat*
**assumes** *y + x > c ⟹ y + x > c*
**shows** *x + y > c ⟹ y + x > c*
**by** (*rewrite* **in** *asm add.commute*) *fact*

**lemma**
**fixes** *x y* :: *nat*
**assumes** *y + x > c ⟹ y + x > c*
**shows** *x + y > c ⟹ y + x > c*
**by** (*rewrite* **in** *x + y > c at asm add.commute*) *fact*

**lemma**
**fixes** *x y* :: *nat*
**assumes** *y + x > c ⟹ y + x > c*
**shows** *x + y > c ⟹ y + x > c*
**by** (*rewrite at ¤ > c at asm add.commute*) *fact*

**lemma**
**assumes** *P {x::int. y + 1 = 1 + x}*
**shows** *P {x::int. y + 1 = x + 1}*
**by** (*rewrite at x+1* **in** *{x::int. ¤ } add.commute*) *fact*

**lemma**
**assumes** *P {x::int. y + 1 = 1 + x}*
**shows** *P {x::int. y + 1 = x + 1}*
**by** (*rewrite at any_identifier_will_work+1* **in** *{any_identifier_will_work::int. ¤ } add.commute*)
*fact*

**lemma**
**assumes** *P {(x::nat, y::nat, z). x + z * 3 = Q (λs t. s * t + y − 3)}*
**shows** *P {(x::nat, y::nat, z). x + z * 3 = Q (λs t. y + s * t − 3)}*
**by** (*rewrite at b + d * e* **in** *λ(a, b, c). _ = Q (λd e. ¤) add.commute*) *fact*

**lemma**
**assumes** $PROP\ P \equiv PROP\ Q$
**shows** $PROP\ R \Longrightarrow PROP\ P \Longrightarrow PROP\ Q$
**by** (*rewrite at asm assms*)

**lemma**
**assumes** $PROP\ P \equiv PROP\ Q$
**shows** $PROP\ R \Longrightarrow PROP\ R \Longrightarrow PROP\ P \Longrightarrow PROP\ Q$
**by** (*rewrite at asm assms*)

**lemma**
**assumes** $(PROP\ P \Longrightarrow PROP\ Q) \equiv (PROP\ S \Longrightarrow PROP\ R)$
**shows** $PROP\ S \Longrightarrow (PROP\ P \Longrightarrow PROP\ Q) \Longrightarrow PROP\ R$
**apply** (*rewrite at asm assms*)
**apply** *assumption*
**done**

**lemma** *test_theorem*:
**fixes** $x :: nat$
**shows** $x \leq y \Longrightarrow x \geq y \Longrightarrow x = y$
**by** (*rule Orderings.order_antisym*)

**lemma**
**fixes** $f :: nat \Rightarrow nat$
**shows** $f\ x \leq 0 \Longrightarrow f\ x \geq 0 \Longrightarrow f\ x = 0$
**apply** (*rewrite at f x to 0 test_theorem*)
**apply** *assumption*
**apply** *assumption*
**apply** (*rule refl*)
**done**

**lemma**
**assumes** *rewr*: $PROP\ P \Longrightarrow PROP\ Q \Longrightarrow PROP\ R \equiv PROP\ R'$
**assumes** *A1*: $PROP\ S \Longrightarrow PROP\ T \Longrightarrow PROP\ U \Longrightarrow PROP\ P$
**assumes** *A2*: $PROP\ S \Longrightarrow PROP\ T \Longrightarrow PROP\ U \Longrightarrow PROP\ Q$
**assumes** *C*: $PROP\ S \Longrightarrow PROP\ R' \Longrightarrow PROP\ T \Longrightarrow PROP\ U \Longrightarrow PROP\ V$
**shows** $PROP\ S \Longrightarrow PROP\ R \Longrightarrow PROP\ T \Longrightarrow PROP\ U \Longrightarrow PROP\ V$
**apply** (*rewrite at asm rewr*)
**apply** (*fact A1*)
**apply** (*fact A2*)
**apply** (*fact C*)
**done**

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

**lemma** *annotate_f*: $f = f_inv\ I$
**by** (*simp add*: *f_inv_def fun_eq_iff*)

**lemma**
**assumes** $P\ (\lambda n.\ f_inv\ (\lambda_.\ True)\ n + 1) = x$
**shows** $P\ (\lambda n.\ f\ n + 1) = x$
**by** (*rewrite to f_inv* $(\lambda_.\ True)$ *annotate_f*) *fact*

**lemma**
**assumes** $P\ (\lambda n.\ f_inv\ (\lambda x.\ n < x + 1)\ n + 1) = x$
**shows** $P\ (\lambda n.\ f\ n + 1) = x$
**by** (*rewrite* **in** $\lambda n.\ \square$ *to f_inv* $(\lambda x.\ n < x + 1)$ *annotate_f*) *fact*

**lemma**
**assumes** $P\ (\lambda n.\ f_inv\ (\lambda x.\ n < x + 1)\ n + 1) = x$
**shows** $P\ (\lambda n.\ f\ n + 1) = x$
**by** (*rewrite* **in** $\lambda abc.\ \square$ *to f_inv* $(\lambda x.\ abc < x + 1)$ *annotate_f*) *fact*

**lemma**
**assumes** $P\ (2 + 1)$
**shows** $\bigwedge x\ y.\ P\ (1 + 2 :: nat)$
**by** (*rewrite* **in** $P\ (1 + 2)$ *at* **for** $(x)$ *add.commute*) *fact*

**lemma**
**assumes** $\bigwedge x\ y.\ P\ (y + x)$
**shows** $\bigwedge x\ y.\ P\ (x + y :: nat)$
**by** (*rewrite* **in** $P\ (x + _)$ *at* **for** $(x\ y)$ *add.commute*) *fact*

**lemma**
**assumes** $\bigwedge x\ y\ z.\ y + x + z = z + y + (x::int)$
**shows** $\bigwedge x\ y\ z.\ x + y + z = z + y + (x::int)$
**by** (*rewrite at* $x + y$ **in** $x + y + z$ **in for** $(x\ y\ z)$ *add.commute*) *fact*

**lemma**
**assumes** $\bigwedge x\ y\ z.\ z + (x + y) = z + y + (x::int)$
**shows** $\bigwedge x\ y\ z.\ x + y + z = z + y + (x::int)$
**by** (*rewrite at* $(_ + y) + z$ **in for** $(y\ z)$ *add.commute*) *fact*

```
lemma
  assumes ⋀x y z. x + y + z = y + z + (x::int)
  shows   ⋀x y z. x + y + z = z + y + (x::int)
by (rewrite at ¤ + _ at _ = ¤ in for () add.commute) fact

lemma
  assumes eq: ⋀x. P x ⟹ g x = x
  assumes f1: ⋀x. Q x ⟹ P x
  assumes f2: ⋀x. Q x ⟹ x
  shows ⋀x. Q x ⟹ g x
  apply (rewrite at g x in for (x) eq)
  apply (fact f1)
  apply (fact f2)
  done


lemma
  assumes (⋀(x::int). x < 1 + x)
  and     (x::int) + 1 > x
  shows   (⋀(x::int). x + 1 > x) ⟹ (x::int) + 1 > x
by (rewrite at x + 1 in for (x) at asm add.commute)
   (rule assms)


lemma
  assumes ⋀a b. P ((a + 1) * (1 + b))
  shows ⋀a b :: nat. P ((a + 1) * (b + 1))
  apply (tactic ‹
    let
      val (x, ctxt) = yield_singleton Variable.add_fixes x context
      (* Note that the pattern order is reversed *)
      val pat = [
        Rewrite.For [(x, SOME Type ‹nat›)],
        Rewrite.In,
         Rewrite.Term (Const ‹plus Type ‹nat› for ‹Free (x, Type ‹nat›)› term ‹1
:: nat››, [])]
      val to = NONE
   in CCONVERSION (Rewrite.rewrite_conv ctxt (pat, to) @{thms add.commute})
1 end
  ›)
  apply (fact assms)
  done

lemma
  assumes Q (λb :: int. P (λa. a + b) (λa. a + b))
  shows Q (λb :: int. P (λa. a + b) (λa. b + a))
  apply (tactic ‹
    let
      val (x, ctxt) = yield_singleton Variable.add_fixes x context
```

```
    val pat = [
      Rewrite.Concl,
      Rewrite.In,
    Rewrite.Term (Free (Q, (Type ‹int› --> TVar (('b,0), [])) --> Type ‹bool›)
        $ Abs (x, Type ‹int›, Rewrite.mk_hole 1 (Type ‹int› --> TVar (('b,0),
[])) $ Bound 0), [(x, Type ‹int›)]),
      Rewrite.In,
      Rewrite.Term (Const ‹plus Type ‹int› for ‹Free (x, Type ‹int›)› ‹Var ((c,
0), Type ‹int›)››, [])
      ]
    val to = NONE
  in CCONVERSION (Rewrite.rewrite_conv ctxt (pat, to) @{thms add.commute})
1 end
›)
apply (fact assms)
done


ML ‹
  val ct = cprop ‹Q (λb :: int. P (λa. a + b) (λa. b + a))›
  val (x, ctxt) = yield_singleton Variable.add_fixes x context
  val pat = [
    Rewrite.Concl,
    Rewrite.In,
    Rewrite.Term (Free (Q, (typ ‹int› --> TVar (('b,0), [])) --> typ ‹bool›)
      $ Abs (x, typ ‹int›, Rewrite.mk_hole 1 (typ ‹int› --> TVar (('b,0), [])) $
Bound 0), [(x, typ ‹int›)]),
    Rewrite.In,
    Rewrite.Term (Const ‹plus Type ‹int› for ‹Free (x, Type ‹int›)› ‹Var ((c, 0),
Type ‹int›)››, [])
    ]
  val to = NONE
  val th = Rewrite.rewrite_conv ctxt (pat, to) @{thms add.commute} ct
›
```

Some regression tests

```
ML ‹
  val ct = cterm ‹(λb :: int. (λa. b + a))›
  val (x, ctxt) = yield_singleton Variable.add_fixes x context
  val pat = [
    Rewrite.In,
    Rewrite.Term (Const ‹plus Type ‹int› for ‹Var ((c, 0), Type ‹int›)› ‹Var ((c,
0), Type ‹int›)››, [])
    ]
  val to = NONE
  val _ =
    case try (Rewrite.rewrite_conv ctxt (pat, to) @{thms add.commute}) ct of
      NONE => ()
    | _ => error should not have matched anything
```

›

**ML** ‹
  *Rewrite.params_pconv* (*Conv.all_conv* |> *K* |> *K*) ***context*** (*Vartab.empty*, [])
***cterm*** ‹⋀*x*. *PROP A*›
›

**lemma**
  **assumes** *eq*: *PROP A* ⟹ *PROP B* ≡ *PROP C*
  **assumes** *f1*: *PROP D* ⟹ *PROP A*
  **assumes** *f2*: *PROP D* ⟹ *PROP C*
  **shows** ⋀*x*. *PROP D* ⟹ *PROP B*
  **apply** (*rewrite eq*)
  **apply** (*fact f1*)
  **apply** (*fact f2*)
  **done**

**end**

# 19 Finite sequences

**theory** *Seq*
  **imports** *Main*
**begin**

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

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

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

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

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

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

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

**end**

# 20 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**
**from** ‹*prime p*› **have** *p*: $p > 1$ **by** (*rule prime_gt_1_nat*)
**assume** $sqrt\ p \in \mathbb{Q}$
**then obtain** $m\ n :: nat$
**where** *n*: $n \neq 0$
**and** *sqrt_rat*: $|sqrt\ p| = m\ /\ n$
**and** *coprime m n* **by** (*rule Rats_abs_nat_div_natE*)
**have** *eq*: $m^2 = p * n^2$
**proof** −
**from** *n* **and** *sqrt_rat* **have** $m = |sqrt\ p| * n$ **by** *simp*
**then have** $m^2 = (sqrt\ p)^2 * n^2$ **by** (*simp add*: *power_mult_distrib*)
**also have** $(sqrt\ p)^2 = p$ **by** *simp*
**also have** $\ldots * n^2 = p * n^2$ **by** *simp*
**finally show** *?thesis* **by** *linarith*
**qed**
**have** *p dvd m* ∧ *p dvd n*
**proof**
**from** *eq* **have** *p dvd* $m^2$ **..**
**with** ‹*prime p*› **show** *p dvd m* **by** (*rule prime_dvd_power*)
**then obtain** *k* **where** $m = p * k$ **..**
**with** *eq* **have** $p * n^2 = p^2 * k^2$ **by** *algebra*
**with** *p* **have** $n^2 = p * k^2$ **by** (*simp add*: *power2_eq_square*)
**then have** *p dvd* $n^2$ **..**
**with** ‹*prime p*› **show** *p dvd n* **by** (*rule prime_dvd_power*)
**qed**
**then have** *p dvd gcd m n* **by** *simp*
**with** ‹*coprime m n*› **have** $p = 1$ **by** *simp*
**with** *p* **show** *False* **by** *simp*
**qed**

**corollary** *sqrt_2_not_rat*: $sqrt\ 2 \notin \mathbb{Q}$
**using** *sqrt_prime_irrational* [*of 2*] **by** *simp*

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

**theorem**
  **fixes** $p :: nat$
  **assumes** *prime p*
  **shows** $sqrt\ p \notin \mathbb{Q}$
**proof**
  **from** ‹*prime p*› **have** *p*: $p > 1$ **by** (*rule prime_gt_1_nat*)
  **assume** $sqrt\ p \in \mathbb{Q}$
  **then obtain** *m n* :: *nat*
    **where** *n*: $n \neq 0$
      **and** *sqrt_rat*: $|sqrt\ p| = m\ /\ n$
      **and** *coprime m n* **by** (*rule Rats_abs_nat_div_natE*)
  **from** *n* **and** *sqrt_rat* **have** $m = |sqrt\ p| * n$ **by** *simp*
  **then have** $m^2 = (sqrt\ p)^2 * n^2$ **by** (*auto simp add*: *power2_eq_square*)
  **also have** $(sqrt\ p)^2 = p$ **by** *simp*
  **also have** $\ldots * n^2 = p * n^2$ **by** *simp*
  **finally have** *eq*: $m^2 = p * n^2$ **by** *linarith*
  **then have** *p dvd* $m^2$ **..**
  **with** ‹*prime p*› **have** *dvd_m*: *p dvd m* **by** (*rule prime_dvd_power*)
  **then obtain** *k* **where** $m = p * k$ **..**
  **with** *eq* **have** $p * n^2 = p^2 * k^2$ **by** *algebra*
  **with** *p* **have** $n^2 = p * k^2$ **by** (*simp add*: *power2_eq_square*)
  **then have** *p dvd* $n^2$ **..**
  **with** ‹*prime p*› **have** *p dvd n* **by** (*rule prime_dvd_power*)
  **with** *dvd_m* **have** *p dvd gcd m n* **by** (*rule gcd_greatest*)
  **with** ‹*coprime m n*› **have** $p = 1$ **by** *simp*
  **with** *p* **show** *False* **by** *simp*
**qed**

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

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

**end**

# References


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

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