Examples for program extraction in Higher-Order Logic

Stefan Berghofer
May 25, 2015

Contents
1 Auxiliary lemmas used in program extraction examples 1
2 Quotient and remainder 3
3 Greatest common divisor 4
4 Warshall’s algorithm 6
5 Higman’s lemma 11
  5.1 Extracting the program . . . . . . . . . . . . . . . . . . . . . 17
  5.2 Some examples . . . . . . . . . . . . . . . . . . . . . . . . . . 19
6 The pigeonhole principle 20
7 Euclid’s theorem 26

1 Auxiliary lemmas used in program extraction examples

theory Util
imports ~/src/HOL/Library/Old-Datatype
begin

Decidability of equality on natural numbers.

lemma nat-eq-dec: \( \forall n::nat. \ m = n \lor m \neq n \)
  apply (induct m)
  apply (case-tac n)
  apply (case-tac [3] n)
  apply (simp only: nat.simps, iprover?)+
  done
Well-founded induction on natural numbers, derived using the standard structural induction rule.

**lemma nat-wf-ind:**

assumes \( R \colon \forall x \colon \mathbb{N} . (\forall y . y < x \Rightarrow P y) \Rightarrow P x \)

shows \( P z \)

**proof (rule R)**

show \( \forall y . y < z \Rightarrow P y \)

**proof (induct z)**

* case 0
  
  thus ?case by simp

next

* case \((\text{Suc } n \ y)\)

from nat-eq-dec show ?case

**proof**

assume \( ny \colon n = y \)

have \( P n \)
  
  by (rule R) (rule Suc)

with \( ny \) show ?case by simp

next

assume \( n \neq y \)

with \( \text{Suc have } y < n \) by simp

thus ?case by (rule Suc)

qed

qed

Bounded search for a natural number satisfying a decidable predicate.

**lemma search:**

assumes \( \text{dec} \colon \forall x \colon \mathbb{N} . P x \vee \neg P x \)

shows \((\exists x < y. P x) \vee \neg (\exists x < y. P x)\)

**proof (induct y)**

* case 0 show ?case

next

* case \((\text{Suc } z)\)

thus ?case

**proof**

assume \( \exists x < z. P x \)

then obtain \( x \) where \( \text{le: } x < z \) and \( P \colon P x \) by iprover

from \( \text{le have } x < \text{Suc } z \) by simp

with \( P \) show ?case by iprover

next

assume \( \neg (\exists x < z. P x) \)

from \( \text{dec show } \) ?case

**proof**

assume \( P \colon P z \)

have \( z < \text{Suc } z \) by simp

with \( P \) show ?thesis by iprover

next

assume \( \neg P z \)
have \( \neg (\exists x < \text{Suc } z. \ P \ x) \)

proof

assume \( \exists x < \text{Suc } z. \ P \ x \)

then obtain \( x \) where \( \text{le}: x < \text{Suc } z \) and \( P \ x \) by \text{iiprover}

have \( x < z \)

proof (cases \( x = z \))

  case \text{True}
   with \( nP \) and \( P \) show \( ?\text{thesis} \) by \text{simp}

next

  case \text{False}
   with \( \text{le} \) show \( ?\text{thesis} \) by \text{simp}

qed

with \( P \) have \( \exists x < z. \ P \ x \) by \text{iiprover}

with \( \text{nex} \) show \( \text{False} \ldots \)

qed

thus \( ?\text{case} \) by \text{iiprover}

qed

qed

end

2 Quotient and remainder

theory QuotRem

imports Util

begin

Derivation of quotient and remainder using program extraction.

\textbf{theorem} division: \( \exists \ q \ r. \ a = \text{Suc } b \ast q + r \wedge r \leq b \)

\textbf{proof} (induct \( a \))

  case 0
   have \( 0 = \text{Suc } b \ast 0 + 0 \wedge 0 \leq b \) by \text{simp}

  thus \( ?\text{case} \) by \text{iiprover}

next

  case (\text{Suc } a)

then obtain \( r \ q \) where \( \text{I}: a = \text{Suc } b \ast q + r \) and \( r \leq b \) by \text{iiprover}

from \text{nat-eq-dec} show \( ?\text{case} \)

proof

  assume \( r = b \)

  with \( \text{I} \) have \( \text{Suc } a = \text{Suc } b \ast (\text{Suc } q) + 0 \wedge 0 \leq b \) by \text{simp}

  thus \( ?\text{case} \) by \text{iiprover}

next

  assume \( r \neq b \)

  with \( r \leq b \) have \( r < b \) by \text{(simp add: order-less-le)}

  with \( \text{I} \) have \( \text{Suc } a = \text{Suc } b \ast q + (\text{Suc } r) \wedge (\text{Suc } r) \leq b \) by \text{simp}

  thus \( ?\text{case} \) by \text{iiprover}

qed

qed
The program extracted from the above proof looks as follows

\[
\text{division} \equiv \lambda x \ x a.
\]

\[
\text{nat-induct-P x (0, 0)}
\]

(\lambda a \ H. \ \text{let } (x, y) = H
\]

in case nat-eq-dec x xa of Left \Rightarrow (0, Suc y)

| Right \Rightarrow (Suc x, y)

The corresponding correctness theorem is

\[a = \text{Suc b} \ast \text{snd } \text{division a b} + \text{fst } \text{division a b} \land \text{fst } \text{division a b} \leq b\]

lemma division 9 2 = (0, 3) by eval

end

3 Greatest common divisor

theory Greatest-Common-Divisor
imports QuotRem
begin

theorem greatest-common-divisor:
\(\forall n::\text{nat}. \ \text{Suc m} < n \Rightarrow \exists k \ n1 \ m1. \ k \ast n1 = n \land k \ast m1 = \text{Suc m} \land\)

(\(\forall l \ l1 \ l2. \ l \ast l1 = n \rightarrow l \ast l2 = \text{Suc m} \rightarrow l \leq k\))

proof (induct m rule: nat-wf-ind)

case (1 m n)

from division obtain \(r \ q\) where \(h1: n = \text{Suc m} \ast q + r\) and \(h2: r \leq m\)

by iprover

show \(?case\)

proof (cases \(r\))

case 0

with \(h1\) have \(\text{Suc m} \ast q = n\) by simp

moreover have \(\text{Suc m} \ast 1 = \text{Suc m}\) by simp

moreover {\n
fix \(l2\) have \(\forall l1. \ l \ast l1 = n \Rightarrow l \ast l2 = \text{Suc m} \Rightarrow l \leq \text{Suc m}\)

by (cases \(l2\) simp-all )

ultimately show \(?thesis\) by iprover

next

case (Suc nat)

with \(h2\) have \(\text{nat} < m\) by simp

moreover from \(h\) have \(\text{Suc nat} < \text{Suc m}\) by simp

ultimately have \(\exists k \ m1 r1. \ k \ast m1 = \text{Suc m} \land k \ast r1 = \text{Suc nat} \land\)

(\(\forall l1 \ l2. \ l \ast l1 = \text{Suc m} \rightarrow l \ast l2 = \text{Suc nat} \rightarrow l \leq k\))

by (rule 1)}
then obtain \( k \) \( m1 \) \( r1 \) where

\[ h1': k * m1 = \text{Suc} \ m \]

and \( h2': k * r1 = \text{Suc} \ \text{nat} \)

and \( h3': \forall l \ l1 \ l2. \ l * l1 = \text{Suc} \ m \implies l * l2 = \text{Suc} \ \text{nat} \implies l \leq k \)

by \( \text{iprover} \)

have \( mn: \text{Suc} \ m < n \) by (rule 1)

from \( h1 \ h1' \ h2' \ Suc \) have \( k * (m1 * q + r1) = n \)

by (simp add: \( \text{add-mult-distrib2} \ \text{mult.assoc} \ [\text{symmetric}] \))

moreover have \( \forall l \ l1 \ l2. \ l * l1 = n \implies l * l2 = \text{Suc} \ m \implies l \leq k \)

proof –

fix \( l \ l1 \ l2 \)

assume \( ll1n: l * l1 = n \)

assume \( ll2m: l * l2 = \text{Suc} \ m \)

moreover have \( l * (l1 - l2 * q) = \text{Suc} \ \text{nat} \)

by (simp add: \( \text{diff-mult-distrib2} \ h1 \ \text{Suc} \ [\text{symmetric}] \ mn \ ll1n \ ll2m \ [\text{symmetric}] \))

ultimately show \( l \leq k \) by (rule \( h3' \))

qed

ultimately show \( \exists \text{thesis using} \ h1' \) by \( \text{iprover} \)

qed

qed

\textit{extract} \textit{greatest-common-divisor}

The extracted program for computing the greatest common divisor is

\textit{greatest-common-divisor} \equiv

\( \lambda x. \ \text{nat-wf-ind-P} \ x \)

(\( \lambda x \ H2 \ xa. \)

\( \text{let} \ (xa, y) = \text{division} \ xa \ x \)

in \( \text{nat-exhaust-P} \ xa \ (\text{Suc} \ x, y, 1) \)

(\( \lambda \text{nat}. \ \text{let} \ (x, ya) = H2 \ \text{nat} \ (\text{Suc} \ x); \ (xa, ya) = ya \)

in \( (x, xa * y + ya, xa)) \))

\textit{instantiation} \textit{nat :: default}

begin

\textit{definition} \textit{default} = (0::nat)

instance ..

end

\textit{instantiation} \textit{prod :: (default, default) default}

begin

\textit{definition} \textit{default} = (default, default)

instance ..

end
instantiation fun :: (type, default) default begin

definition default = (λx. default)

instance ..
end

lemma greatest-common-divisor 7 12 = (4, 3, 2) by eval
end

4 Warshall’s algorithm

theory Warshall
imports Old-Datatype
begin

Derivation of Warshall’s algorithm using program extraction, based on Berger, Schwichtenberg and Seisenberger [1].

datatype b = T | F

primrec is-path' :: ('a ⇒ 'a ⇒ b) ⇒ 'a ⇒ 'a list ⇒ 'a ⇒ bool
where
  is-path' r x [] z = (r x z = T)
| is-path' r x (y # ys) z = (r x y = T ∧ is-path' r y ys z)

definition is-path :: (nat ⇒ nat ⇒ b) ⇒ (nat * nat list * nat) ⇒
  nat ⇒ nat ⇒ nat ⇒ bool
where
  is-path r p i j k ←→
  fst p = j ∧ snd (snd p) = k ∧
  list-all (λx. x < i) (fst (snd p)) ∧
  is-path' r (fst p) (fst (snd p)) (snd (snd p))

definition conc :: ('a * 'a list * 'a) ⇒ ('a * 'a list * 'a) ⇒ ('a * 'a list * 'a)
where
  conc p q = (fst p, fst (snd p) @ fst q #fst (snd q), snd (snd q))

theorem is-path'-snoc [simp]:
  ∀x. is-path' r x (ys @ [y]) z = (is-path' r x ys y ∧ r y z = T)
by (induct ys) simp+ 

theorem list-all-scoc [simp]: list-all P (xs @ [x]) ←→ P x ∧ list-all P xs
by (induct xs, simp+, iprover)
theorem list-all-lemma:
list-all P xs ⇒ (∀x. P x ⇒ Q x) ⇒ list-all Q xs
proof −
assume PQ: ∀x. P x ⇒ Q x
show list-all P xs ⇒ list-all Q xs
proof (induct xs)
case Nil
show ?case by simp
next
case (Cons y ys)
hence Py: P y by simp
from Cons have Pys: list-all P ys by simp
show ?case
  by simp (rule conjI PQ Py Cons Pys)+
qed
qed

theorem lemma1: ∀p. is-path r p i j k ⇒ is-path r p (Suc i) j k
apply (unfold is-path-def)
apply (simp cong add: conj-cong add: split-paired-all)
apply (erule conjE)+
apply (erule list-all-lemma)
apply simp
done

theorem lemma2: ∀p. is-path r p 0 j k ⇒ r j k = T
apply (unfold is-path-def)
apply (simp cong add: conj-cong add: split-paired-all)
apply (case-tac aa)
apply simp+
done

theorem is-path'-conc: is-path' r j xs i ⇒ is-path' r i ys k ⇒
is-path' r j (xs @ i # ys) k
proof −
assume pys: is-path' r i ys k
show ∀j. is-path' r j xs i ⇒ is-path' r j (xs @ i # ys) k
proof (induct xs)
case (Nil j)
hence r j i = T by simp
with pys show ?case by simp
next
case (Cons z zs j)
hence jzr: r j z = T by simp
from Cons have pzs: is-path' r z zs i by simp
show ?case
  by simp (rule conjI jzr Cons pzs)+
qed
theorem lemma3:
\( \forall p \ q. \text{is-path } r \ p \ i \ j \ i \implies \text{is-path } r \ q \ i \ k \implies \text{is-path } r \ \text{conc } p \ q \ (\text{Suc} \ i) \ j \ k \)
apply (unfold is-path-def conc-def)
apply (simp cong add: conj-cong add: split-paired-all)
apply (erule conjE)+
apply (rule conjI)
apply (rule list-all-lemma)
apply simp
apply (erule list-all-lemma)
apply simp
apply (rule conjI)
apply (erule list-all-lemma)
apply simp
apply (rule is-path′-conc)
apply assumption+
done

theorem lemma5:
\( \forall p. \text{is-path } r \ p \ (\text{Suc} \ i) \ j \ k \implies \neg \text{is-path } r \ p \ i \ j \ k \implies
(\exists q. \text{is-path } r \ q \ i \ j \ i) \land (\exists q′. \text{is-path } r \ q′ \ i \ i \ k) \)
proof (simp cong add: conj-cong add: split-paired-all is-path-def, (erule conjE)+)
fix xs
assume asms:
list-all (\( \lambda x. \ x < \text{Suc} \ i \)) xs
is-path′ r j xs k
\( \neg \text{list-all } (\lambda x. \ x < i) \) xs
show (\( \exists ys. \text{list-all } (\lambda x. \ x < i) \) ys \( \land \text{is-path'} r j ys i \)) \( \land 
(\exists ys. \text{list-all } (\lambda x. \ x < i) \) ys \( \land \text{is-path'} r i ys k \)
proof
show \( \forall j. \text{list-all } (\lambda x. \ x < \text{Suc} \ i) \) xs \( \implies \text{is-path'} r j xs k \)
\( \neg \text{list-all } (\lambda x. \ x < i) \) xs \( \implies 
\exists ys. \text{list-all } (\lambda x. \ x < i) \) ys \( \land \text{is-path'} r j ys i \) (is PROP ?ih xs)
proof (induct xs)
case Nil
thus \( \text{PROP ?case by simp} \)
next
case (Cons a as j)
show ?case
proof (cases a=a)
case True
show ?thesis
proof
from True and Cons have r j i = T by simp
thus list-all (\( \lambda x. \ x < i \)) [] \( \land \text{is-path'} r j [] i \) by simp
qed
next
case False
have PROP ?ih as by (rule Cons)
then obtain \( ys \) where \( ys :: \text{list-all} (\lambda x. x < i) ys \land \text{is-path}' r a ys i \)

proof from Cons show \( \text{list-all} (\lambda x. x < \text{Suc} i) as \) by simp
from Cons show \( \text{is-path}' r a as k \) by simp
from Cons and False show \( \neg \text{list-all} (\lambda x. x < i) as \) by (simp)
qed
show ?thesis
proof
from Cons False ys show \( \text{list-all} (\lambda x. x < \text{Suc} i) as \) by simp
qed
qed

(\text{case} \ A \ \text{rule: rev-induct})
case Nil
thus \ A \ \text{by simp}
next
case (\text{snoc} a as k)
show ?case
proof (cases \( a = i \))
case True
show ?thesis
proof
from True and snoc have \( r i k = T \) by simp
thus \( \text{list-all} (\lambda x. x < i) :: k \land \text{is-path}' r i :: k \) by simp
qed
next
case False
have \( \text{PROP} \ ?ih \ as \) by (rule snoc)
then obtain \( ys \) where \( ys :: \text{list-all} (\lambda x. x < i) ys \land \text{is-path}' r i ys a \)
proof
from snoc show \( \text{list-all} (\lambda x. x < \text{Suc} i) as \) by simp
from snoc show \( \text{is-path}' r j as a \) by simp
from snoc and False show \( \neg \text{list-all} (\lambda x. x < i) as \) by simp
qed
show ?thesis
proof
from snoc False ys show \( \text{list-all} (\lambda x. x < i) (ys @ [a]) \land \text{is-path}' r i (ys @ [a]) k \) by simp
qed
qed
qed

(\text{rule asms})+
theorem lemma5':
\( \forall p. \text{is-path } r \ p \ (\text{Suc } i) \ j \ k \implies \neg \text{is-path } r \ p \ j \ k \implies \\
\neg (\forall q. \neg \text{is-path } r \ q \ i \ j \ i) \land \neg (\forall q'. \neg \text{is-path } r \ q' \ i \ i \ k) \)
by (iprover dest: lemma5)

theorem warshall:
\( \forall j \ k. \neg (\exists p. \text{is-path } r \ p \ i \ j \ k) \lor (\exists p. \text{is-path } r \ p \ i \ j \ k) \)
proof (induct i)
case (0 j k)
show ?case
proof (cases r j k)
assume r j k = T
hence is-path r (j, [], k) 0 j k
by (simp add: is-path-def)
hence \( \exists p. \text{is-path } r \ p \ 0 \ j \ k \) ..
thus ?thesis ..
next
assume r j k = F
hence r j k ~ = T by simp
hence \( \neg (\exists p. \text{is-path } r \ p \ 0 \ j \ k) \)
by (iprover dest: lemma2)
thus ?thesis ..
qed
next
case (Suc i j k)
thus ?case
proof
assume h1: \( \neg (\exists p. \text{is-path } r \ p \ i \ j \ k) \)
from Suc show ?case
proof
assume \( \neg (\exists p. \text{is-path } r \ p \ i \ j \ i) \)
with h1 have \( \neg (\exists p. \text{is-path } r \ p \ (\text{Suc } i) \ j \ k) \)
by (iprover dest: lemma5')
thus ?case ..
next
assume \( \exists p. \text{is-path } r \ p \ i \ j \ i \)
then obtain p where h2: is-path r p i j i ..
from Suc show ?case
proof
assume h1: \( \neg (\exists p. \text{is-path } r \ p \ i \ i \ k) \)
with h1 have \( \neg (\exists p. \text{is-path } r \ p \ (\text{Suc } i) \ j \ k) \)
by (iprover dest: lemma5')
thus ?case ..
next
assume \( \exists q. \text{is-path } r \ q \ i \ i \ k \)
then obtain q where is-path r q i i k ..
with h2 have is-path r (conc p q) (Suc i) j k
by (rule lemma3)
hence \( \exists pq. \text{is-path } r \ pq \ (\text{Suc } i) \ j \ k \) ..
thus \( ?\text{case} \ldots \)
qed

next

assume \( \exists p. \text{is-path} r p i j k \)
hence \( \exists p. \text{is-path} r p (\text{Suc} i) j k \)
by (iprover intro: lemma1)
thus \( ?\text{case} \ldots \)
qed

extract warshall

The program extracted from the above proof looks as follows

\[
\text{warshall} \equiv \\
\lambda x x a x a a x a a a . \\
\text{nat-induct-P} x u \\
(\lambda x a a x a a a . \text{case} x x a x a a o f \ T \Rightarrow \text{Some} (x a, [], x a a) \mid F \Rightarrow \text{None}) \\
(\lambda x H 2 x a x a a . \\
\text{case} H 2 x x a x a a of \\
\text{None} \Rightarrow \\
\text{case} H 2 x x a x a a of \text{None} \Rightarrow \text{None} \\
\mid \text{Some} q \Rightarrow \\
\text{case} H 2 x x a x a a of \text{None} \Rightarrow \text{None} \mid \text{Some} q a \Rightarrow \text{Some} (\text{conc} q q a a) \\
\mid \text{Some} q \Rightarrow \text{Some} q)
\]

\( x a a x a a a \)

The corresponding correctness theorem is

\[
\text{case warshall} r i j k \text{ of None} \Rightarrow \forall x. \neg \text{is-path} r x i j k \\
\mid \text{Some} q \Rightarrow \text{is-path} r q i j k
\]

ML-val @{\{code warshall\}}

end

5 Higman’s lemma

theory Higman
imports Old-Datatype
begin

Formalization by Stefan Berghofer and Monika Seisenberger, based on Coquand and Fridlender [2].

datatype letter = A | B

inductive emb :: letter list \Rightarrow letter list \Rightarrow bool
where
\[\text{emb0} \ [\text{Pure.intro}]: \text{emb} \ [\ ] \ bs \]
| \text{emb1} \ [\text{Pure.intro}]: \text{emb} \ as \ bs \implies \text{emb} \ (b \ # \ bs) \]
| \text{emb2} \ [\text{Pure.intro}]: \text{emb} \ as \ bs \implies \text{emb} \ (a \ # \ as) \ (a \ # \ bs) \]

\textbf{inductive} \ L :: \text{letter list} \Rightarrow \text{letter list list} \Rightarrow \text{bool} \\
\text{for} \ v :: \text{letter list} \\
\text{where} \\
L0 \ [\text{Pure.intro}]: \text{emb} \ w \ v \implies L \ v \ (w \ # \ ws) \\
| L1 \ [\text{Pure.intro}]: L \ v \ ws \implies L \ v \ (w \ # \ ws) \\

\textbf{inductive} \ \text{good} :: \text{letter list list} \Rightarrow \text{bool} \\
\text{where} \\
good0 \ [\text{Pure.intro}]: L \ w \ ws \implies \text{good} \ (w \ # \ ws) \\
good1 \ [\text{Pure.intro}]: \text{good} \ ws \implies \text{good} \ (w \ # \ ws) \\

\textbf{inductive} \ R :: \text{letter} \Rightarrow \text{letter list list} \Rightarrow \text{letter list list} \Rightarrow \text{bool} \\
\text{for} \ a :: \text{letter} \\
\text{where} \\
R0 \ [\text{Pure.intro}]: R \ a \ [\ ] \ [\ ] \\
| R1 \ [\text{Pure.intro}]: R \ a \ vs \ ws \implies R \ a \ (w \ # \ vs) \ ((a \ # \ w) \ # \ ws) \\

\textbf{inductive} \ T :: \text{letter} \Rightarrow \text{letter list list} \Rightarrow \text{letter list list} \Rightarrow \text{bool} \\
\text{for} \ a :: \text{letter} \\
\text{where} \\
T0 \ [\text{Pure.intro}]: a \neq b \implies R \ b \ ws \ zs \implies T \ a \ (w \ # \ zs) \ ((a \ # \ w) \ # \ zs) \\
| T1 \ [\text{Pure.intro}]: T \ a \ ws \ zs \implies T \ a \ (w \ # \ ws) \ ((a \ # \ w) \ # \ zs) \\
| T2 \ [\text{Pure.intro}]: a \neq b \implies T \ a \ ws \ zs \implies T \ a \ ws \ ((b \ # \ w) \ # \ zs) \\

\textbf{inductive} \ \text{bar} :: \text{letter list list} \Rightarrow \text{bool} \\
\text{where} \\
bar1 \ [\text{Pure.intro}]: \text{good} \ ws \implies \text{bar} \ ws \\
| bar2 \ [\text{Pure.intro}]: (\forall w. \text{bar} \ (w \ # \ ws)) \implies \text{bar} \ ws \\

\textbf{theorem} \ \text{prop1}: \text{bar} \ ([\ ] \ # \ ws) \text{ by iprover} \\

\textbf{theorem} \ \text{lemma1}: L \ as \ ws \implies L \ (a \ # \ as) \ ws \\
\text{by} \ (\text{erule } L\text{-induct, iprover+}) \\

\textbf{lemma} \ \text{lemma2'}: R \ a \ vs \ ws \implies L \ as \ vs \implies L \ (a \ # \ as) \ ws \\
\text{apply} \ (\text{induct set}: \ R) \\
\text{apply} \ (\text{erule } L\text{-cases}) \\
\text{apply} \ \text{simp+} \\
\text{apply} \ (\text{erule } L\text{-cases}) \\
\text{apply} \ \text{simp-all} \\
\text{apply} \ (\text{rule } L0) \\
\text{apply} \ (\text{erule } \text{emb2}) \\
\text{apply} \ (\text{erule } L1) \\
\text{done} \\

\text{12}
lemma lemma2: \( R \ a \ vs \ ws \Rightarrow good \ vs \Rightarrow good \ ws \)
apply (induct set: \( R \))
apply iprover
apply (erule good, cases)
apply simp-all
apply (rule good0)
apply (erule lemma2')
apply assumption
apply (erule good1)
done

lemma lemma3': \( T \ a \ vs \ ws \Rightarrow L \ as \ vs \Rightarrow L (a \# as) \ ws \)
apply (induct set: \( T \))
apply (erule L).
cases
apply simp-all
apply (rule L0)
apply (erule emb2)
apply (rule L1)
apply (erule lemma1)
apply (erule L, cases)
apply simp-all
apply iprover+
done

lemma lemma3: \( T \ a \ ws \ zs \Rightarrow good \ ws \Rightarrow good \ zs \)
apply (induct set: \( T \))
apply (erule good, cases)
apply simp-all
apply (rule good0)
apply (erule lemma1)
apply (erule good1)
apply (erule good, cases)
apply simp-all
apply (rule good0)
apply (erule lemma3')
apply iprover+
done

lemma lemma4: \( R \ a \ ws \ zs \Rightarrow ws \neq [] \Rightarrow T \ a \ ws \ zs \)
apply (induct set: \( R \))
apply iprover
apply (case-tac vs)
apply (erule R, cases)
apply simp
apply (case-tac a)
apply (rule-tac b=B in T0)
apply simp
apply (rule R0)
apply (rule-tac b=A in T0)
apply simp
apply (rule R0)
apply simp
apply (rule T1)
apply simp
done

lemma letter-neq: \( \text{a} :: \text{letter} \neq \text{b} \implies \text{c} \neq \text{a} \implies \text{c} = \text{b} \)
apply (case-tac a)
apply (case-tac b)
apply (case-tac c, simp, simp)
apply (case-tac c, simp, simp)
apply (case-tac b)
apply (case-tac c, simp, simp)
apply (case-tac c, simp, simp)
done

lemma letter-eq-dec: \( \text{a} :: \text{letter} = \text{b} \lor \text{a} \neq \text{b} \)
apply (case-tac a)
apply (case-tac b)
apply simp
apply simp
apply (case-tac b)
apply simp
apply simp
apply simp
done

theorem prop2:
assumes ab: \( \text{a} \neq \text{b} \) and bar: \( \text{bar} \text{xs} \)
shows \( \forall \text{ys} \text{zs}. \text{bar} \text{ys} \implies \text{T} \text{a} \text{xs} \text{zs} \implies \text{T} \text{b} \text{ys} \text{zs} \implies \text{bar} \text{zs} \) using bar
proof induct
fix \( \text{xs} \text{zs} \) assume \( \text{T} \text{a} \text{xs} \text{zs} \) and good \( \text{xs} \)
hence good \( \text{zs} \) by (rule lemma3)
then show \text{bar} \text{zs} by (rule bar1)
next
fix \( \text{ys} \text{zs} \)
assume I: \( \forall \text{ys} \text{zs}. \text{bar} \text{ys} \implies \text{T} \text{a} (\text{w} \# \text{xs}) \text{zs} \implies \text{T} \text{b} \text{ys} \text{zs} \implies \text{bar} \text{zs} \)
assume bar \( \text{ys} \)
thus \( \forall \text{zs}. \text{T} \text{a} \text{xs} \text{zs} \implies \text{T} \text{b} \text{ys} \text{zs} \implies \text{bar} \text{zs} \)
proof induct
fix \( \text{ys} \text{zs} \) assume \( \text{T} \text{b} \text{ys} \text{zs} \) and good \( \text{ys} \)
then have good \( \text{zs} \) by (rule lemma3)
then show \text{bar} \text{zs} by (rule bar1)
next
fix \( \text{ys} \text{zs} \) assume I': \( \forall \text{w} \text{zs}. \text{T} \text{a} \text{xs} \text{zs} \implies \text{T} \text{b} (\text{w} \# \text{ys}) \text{zs} \implies \text{bar} \text{zs} \)
and \( \forall \text{w}. \text{bar} (\text{w} \# \text{ys}) \) and \( \text{Ta}: \text{T} \text{a} \text{xs} \text{zs} \) and \( \text{Tb}: \text{T} \text{b} \text{ys} \text{zs} \)
show \text{bar} \text{zs}
proof (rule bar2)
fix \( \text{w} \)
show \( \text{bar} (w \neq zs) \)
proof (cases \( w \))
  case Nil
  thus ?thesis by simp (rule prop1)
next
  case (\text{Cons} c \text{ cs})
  from letter-eq-dec show ?thesis
  proof
    assume \( \text{ca}: c = a \)
    from \( \text{ab} \) have \( \text{bar} ((a \# \text{cs}) \# zs) \) by (iprover intro: I \( y\ y \) \( a \) \( T \text{a} \text{T} \text{b} \))
    thus ?thesis by (simp add: Cons ca)
  next
    assume \( c \neq a \)
    with \( \text{ab} \) have \( \text{cb}: c = b \) by (rule letter-neq)
    from \( \text{ab} \) have \( \text{bar} ((b \# \text{cs}) \# zs) \) by (iprover intro: I' \( \text{a} \text{T} \text{a} \text{T} \text{b} \))
    thus ?thesis by (simp add: Cons cb)
  qed
qed

\text{qed}

\text{theorem prop3:}
\text{assumes bar: bar xs}
\text{shows } \forall zs. xs \neq [] \Rightarrow R a xs zs \Rightarrow \text{bar zs} \text{ using bar}
\text{proof induct}
  fix xs zs
  assume \( \text{R a xs zs} \) and \( \text{good} \) xs
  then have \( \text{good} \) zs by (rule lemma2)
  then show \( \text{bar} \) zs by (rule bar1)
next
  fix xs zs
  assume \( I : \forall zs. w \neq \# xs \neq [] \Rightarrow R a (w \# zs) \Downarrow \text{bar} \) zs
  and \( \text{zsb: } \forall w. \text{bar} (w \# xs) \) and \( \text{xsn: } \) \( xs \neq [] \) and \( \text{R: } R a \) \( xs \) \( zs \)
  show \( \text{bar} \) zs
  proof (rule bar2)
    fix \( w \)
    show \( \text{bar} (w \# zs) \)
    proof (induct \( w \))
      case Nil
      show ?case by (rule prop1)
    next
      case (\text{Cons} c \text{ cs})
      from letter-eq-dec show ?case
      proof
        assume \( \text{ca}: c = a \)
        thus ?thesis by (iprover intro: I [simplified] \( R \))
      next
      from \( \text{R xsn} \) have \( T: T a xs zs \) by (rule lemma4)
assumed $c \neq a$

thus ?thesis by (iprover intro: prop2 Cons xsb xsn R T)

qed

qed

qed

theorem higman: bar []

proof (rule bar2)

fix w

show bar [w]

proof (induct w)

show bar [[]] by (rule prop1)

next

fix c cs assume bar [cs]

thus bar [c # cs] by (rule prop3) (simp, iprover)

qed

qed

primrec

is-prefix :: 'a list ⇒ (nat ⇒ 'a) ⇒ bool

where

is-prefix [] f = True

| is-prefix (x # xs) f = (x = f (length xs) ∧ is-prefix xs f)

theorem L-idx:

assumes L: L w ws

shows is-prefix ws f ⇒ ∃i. emb (f i) w ∧ i < length ws using L

proof

induct

case (L0 v ws)

hence emb (f ((length ws))) w by simp

moreover have length ws < length (v # ws) by simp

ultimately show ?case by iprover

next

case (L1 ws v)

then obtain i where emb: emb (f i) w and i < length ws

by simp iprover

hence i < length (v # ws) by simp

with emb show ?case by iprover

qed

theorem good-idx:

assumes good: good ws

shows is-prefix ws f ⇒ ∃i j. emb (f i) (f j) ∧ i < j using good

proof

induct

case (good0 w ws)

hence w = f (length ws) and is-prefix ws f by simp-all

with good0 show ?case by (iprover dest: L-idx)

next

16
case (good1 ws w)
thus ?case by simp
qed

theorem bar-idx:
assumes bar: bar ws
shows is-prefix ws f \implies \exists i j. emb (f i) (f j) \land i < j using bar
proof induct
  case (bar1 ws)
  thus ?case by (rule good-idx)
next
case (bar2 ws)
hence is-prefix (f (length ws) \# ws) f by simp
  thus ?case by (rule bar2)
qed

Strong version: yields indices of words that can be embedded into each other.

theorem higman-idx: \exists (i::nat) j. emb (f i) (f j) \land i < j
proof (rule bar-idx)
  show bar [] by (rule higman)
  show is-prefix [] f by simp
qed

Weak version: only yield sequence containing words that can be embedded into each other.

theorem good-prefix-lemma:
assumes bar: bar ws
shows is-prefix ws f \implies \exists vs. is-prefix vs f \land good vs using bar
proof induct
  case bar1
  thus ?case by iprover
next
case (bar2 ws)
  from bar2.prems have is-prefix (f (length ws) \# ws) f by simp
  thus ?case by (iprover intro: bar2)
qed

theorem good-prefix: \exists vs. is-prefix vs f \land good vs
using higman
by (rule good-prefix-lemma) simp+

5.1 Extracting the program

declare R.induct [ind-realizer]
declare T.induct [ind-realizer]
declare L.induct [ind-realizer]
declare good.induct [ind-realizer]
declare bar.induct [ind-realizer]

extract higman-idx

Program extracted from the proof of higman-idx:

\[ higman-idx \equiv \lambda x. \text{bar-idx } x \ higman \]

Corresponding correctness theorem:

\[ \text{emb } (f \ (\text{fst } (higman-idx \ f))) \ (f \ (\text{snd } (higman-idx \ f))) \land \text{fst } (higman-idx \ f) < \text{snd } (higman-idx \ f) \]

Program extracted from the proof of higman:

\[ higman \equiv \bar{2} \ [] \ (\text{rec-list } (\text{prop1 } [])) \ (\lambda a \ w \ H. \ \text{prop3 } a [a \neq w] \ H \ (R1 \ [] \ [] \ w \ R0)) \]

Program extracted from the proof of prop1:

\[ \text{prop1 } \equiv \lambda x. \ \bar{2} \ [] \ x \ (\lambda w. \ \bar{1} \ w \ (w \neq [] \ # \ x) \ (\text{good0 } w \ ([] \ # \ x) \ (L0 \ [] \ x))) \]

Program extracted from the proof of prop2:

\[ \text{prop2 } \equiv \lambda x \ xa \ xaa \ xaaa \ H. \ \text{rec-barT} \ (\lambda ws \ xa \ xaa \ xaaa \ Ha \ Ha. \ \text{bar1} \ xaa \ (\text{lemma3 } x \ Ha \ x)) \]

Program extracted from the proof of prop3:

\[ \text{prop3 } \equiv \lambda x \ xa \ H. \ \text{rec-barT} \ (\lambda ws \ xa \ xaa \ H. \ \text{bar1} \ xaa \ (\text{lemma2 } x \ H \ xa)) \]
\[ (\lambda w x a r x a a H. \\
\quad \text{bar2 xaa} \\
\quad (\text{rec-list} \ (\text{prop1 xaa}) \\
\quad (\lambda a w H a. \\
\quad \text{case letter-eq-dec} a x \text{ of} \\
\quad \text{Left } \Rightarrow \ r \ w \ ((x \ # \ w) \ # \ xaa) \ (R1 \ ws \ xaa \ w H) \\
\quad \ | \ \text{Right } \Rightarrow \ \\
\quad \text{prop2} a x ws ((a \ # \ w) \ # \ xaa) \ Ha \ (\text{bar2} ws \ xaa) \\
\quad \ (T0 \ x \ ws \ xaa \ w H) \ (T2 \ a \ ws \ xaa \ w \ (\text{lemma4} x H)))) \)]

\[ H x a \]

### 5.2 Some examples

**instantiation LT and TT :: default**

**begin**

**definition** default = L0 [] []

**definition** default = T0 A [] [] R0

**instance ..**

**end**

**function** mk-word-aux :: nat ⇒ Random.seed ⇒ letter list × Random.seed where

mk-word-aux k = exec {
  i ← Random.range 10;
  if i > 7 ∧ k > 2 ∨ k > 1000 then Pair []
  else exec {
    let l = (if i mod 2 = 0 then A else B);
    ls ← mk-word-aux (Suc k);
    Pair (l # ls)
  }
}

**by** pat-completeness auto

**termination by** (relation measure ((op −) 1001)) auto

**definition** mk-word :: Random.seed ⇒ letter list × Random.seed where

mk-word = mk-word-aux 0

**primrec** mk-word-s :: nat ⇒ Random.seed ⇒ letter list × Random.seed where

mk-word-s 0 = mk-word

| mk-word-s (Suc n) = \[ \text{exec} \{ \\
\quad - \ ← \ mk-word; \\
\quad \text{mk-word-s} \ n \\
\}\] 

**definition** g1 :: nat ⇒ letter list where

\[ g1 s = \text{fst} (\text{mk-word-s} \ s \ (20000, 1)) \]
\textbf{definition} \( g2 :: \text{nat} \Rightarrow \text{letter list} \) where
\( g2 \ s = \text{fst} \ (\text{mk-word-s} \ s \ (50000, 1)) \)

\textbf{fun} \( f1 :: \text{nat} \Rightarrow \text{letter list} \) where
\( f1 \ 0 = \ [A, A] \)
\( |f1 \ (\text{Suc} \ 0) = [B] \)
\( |f1 \ (\text{Suc} \ (\text{Suc} \ 0)) = [A, B] \)
\( |f1 \ - = [] \)

\textbf{fun} \( f2 :: \text{nat} \Rightarrow \text{letter list} \) where
\( f2 \ 0 = \ [A, A] \)
\( |f2 \ (\text{Suc} \ 0) = [B] \)
\( |f2 \ (\text{Suc} \ (\text{Suc} \ 0)) = [B, A] \)
\( |f2 \ - = [] \)

\textbf{ML-val} \( \langle\langle \)
local
val higman-idx = @{code higman-idx};
val \( g1 = \@\{\text{code} \ g1\}; \)
val \( g2 = \@\{\text{code} \ g2\}; \)
val \( f1 = \@\{\text{code} \ f1\}; \)
val \( f2 = \@\{\text{code} \ f2\}; \)
in
val \( (i1, j1) = \text{higman-idx} \ g1; \)
val \( (v1, w1) = (g1 \ i1, g1 \ j1); \)
val \( (i2, j2) = \text{higman-idx} \ g2; \)
val \( (v2, w2) = (g2 \ i2, g2 \ j2); \)
val \( (i3, j3) = \text{higman-idx} \ f1; \)
val \( (v3, w3) = (f1 \ i3, f1 \ j3); \)
val \( (i4, j4) = \text{higman-idx} \ f2; \)
val \( (v4, w4) = (f2 \ i4, f2 \ j4); \)
end;
\rangle\rangle \)

\textbf{6 The pigeonhole principle}

\textbf{theory} \( \text{Pigeonhole} \)
\textbf{imports} \( \text{Util} \sim\sim/\text{src/HOL/Library/Code-Target-Numeral} \)
\textbf{begin}

We formalize two proofs of the pigeonhole principle, which lead to extracted programs of quite different complexity. The original formalization of these proofs in NUPRL is due to Aleksey Nogin [3].

This proof yields a polynomial program.

\textbf{theorem} \( \text{pigeonhole}: \)
\( \forall \ f. \ (\forall i. \ i \leq \text{Suc} \ n \Rightarrow f \ i \leq n) \Rightarrow \exists \ i \ j. \ i \leq \text{Suc} \ n \land j < i \land f i = f j \)
proof (induct n)
case 0
hence Suc 0 ≤ Suc 0 ∧ 0 < Suc 0 ∧ f (Suc 0) = f 0 by simp
thus ?case by iprover
next
case (Suc n)
{ 
  fix k
  have
  k ≤ Suc (Suc n) \implies
  (\(i, j\). Suc k ≤ i \implies i ≤ Suc (Suc n) \implies j < i \implies f i ≠ f j) \implies
  (\(i, j\). i ≤ k ∧ j < i ∧ f i = f j)
proof (induct k)
case 0
let \(?f = \lambda i. if f i = Suc n then f (Suc (Suc n)) else f i\)
have \(\neg (\exists i j. i ≤ Suc n ∧ j < i ∧ ?f i = ?f j)\)
proof
  assume \(\exists i j. i ≤ Suc n ∧ j < i ∧ ?f i = ?f j\)
  then obtain i j where i: i ≤ Suc n and j: j < i
  and f: \(?f i = ?f j\) by iprover
  from j have i-nz: Suc 0 ≤ i by simp
  from i have iSSn: i ≤ Suc (Suc n) by simp
  have S0SSn: Suc 0 ≤ Suc (Suc n) by simp
  show False
  proof cases
    assume fi: f i = Suc n
    show False
    proof cases
      assume fj: f j = Suc n
      from i-nz and iSSn and j have f i ≠ f j by (rule 0)
      moreover from fi have f i = f j
      by (simp add: fj [symmetric])
      ultimately show \(?thesis ..\)
    next
    from i and j have j < Suc (Suc n) by simp
    with S0SSn and le-refl have f (Suc (Suc n)) ≠ f j
      by (rule 0)
    moreover assume f j ≠ Suc n
    with fi and f have f (Suc (Suc n)) = f j by simp
    ultimately show False ..
  qed
next
assume fi: f i ≠ Suc n
show False
proof cases
  from i have i < Suc (Suc n) by simp
  with S0SSn and le-refl have f (Suc (Suc n)) ≠ f i
    by (rule 0)
  moreover assume f j = Suc n
with \( f \) and \( f \) have \( f (\text{Suc} (\text{Suc} n)) = f i \) by simp
ultimately show False ..

next
from i-nz and iSSn and \( j \)
have \( f i \neq f j \) by (rule 0)
moreover assume \( f j \neq \text{Suc} n \)
with \( f i \) and \( f \) have \( f i = f j \) by simp
ultimately show False ..

qed
qed

moreover have \( \forall i. i \leq \text{Suc} n \implies \exists f i \leq n \)

proof –
  fix \( i \) assume \( i \leq \text{Suc} n \)
  hence \( i : i < \text{Suc} (\text{Suc} n) \) by simp
  have \( f (\text{Suc} (\text{Suc} n)) \neq f i \)
    by (rule 0) (simp-all add: \( i \))
  moreover have \( f (\text{Suc} (\text{Suc} n)) \leq \text{Suc} n \)
    by (rule Suc) simp
  moreover from \( i \) have \( i \leq \text{Suc} (\text{Suc} n) \) by simp
  hence \( f i \leq \text{Suc} n \) by (rule Suc)
  ultimately show \( \exists \text{thesis} i \)
    by simp

  hence \( \exists i. j. i \leq \text{Suc} n \land j < i \land \exists f i = ?f j \)
    by (rule Suc)
  ultimately show \( \exists \text{case} .. \)

next
  case \( \text{Suc} k \)
  from search [OF nat-eq-dec] show \( \exists \text{case} \)
  proof
    assume \( \exists j < \text{Suc} k. f (\text{Suc} k) = f j \)
    thus \( \exists \text{case} \) by (iprover intro: le-refl)

next
  assume nex: \( \neg (\exists j < \text{Suc} k. f (\text{Suc} k) = f j) \)
  have \( \exists i. j. i \leq k \land j < i \land \exists f i = ?f j \)
  proof (rule Suc)
    from \( \text{Suc} \) show \( k \leq \text{Suc} (\text{Suc} n) \) by simp
    fix \( i. j \) assume \( k. \text{Suc} k \leq i \) and \( i : i \leq \text{Suc} (\text{Suc} n) \)
    and \( j. j < i \)
    show \( f i \neq f j \)
    proof cases
      assume eq: \( i = \text{Suc} k \)
      show \( \exists \text{thesis} \)
      proof
        assume \( f i = f j \)
        hence \( f (\text{Suc} k) = f j \) by (simp add: eq)
        with \( \text{nex} \) and \( j \) and \( \text{eq} \) show False by iprover
      qed
    qed
  qed

...
next
  assume \( i \neq \text{Suc} \, k \)
  with \( k \) have \( \text{Suc} \, (\text{Suc} \, k) \leq i \) by simp
  thus ?thesis using \( i \) and \( j \) by (rule Suc)
  qed
  qed
  thus ?thesis by (iprover intro: le-SucI)
  qed

\}

note \( r = \text{this} \)

show ?case by (rule \( r \)) simp-all

qed

The following proof, although quite elegant from a mathematical point of view, leads to an exponential program:

\[\text{theorem pigeonhole-slow:}\]
\[\forall f. (\forall i. i \leq \text{Suc} \, n \Rightarrow f \, i \leq n) \Rightarrow \exists i \, j. i \leq \text{Suc} \, n \land j < i \land f \, i = f \, j\]

\[\text{proof (induct } n)\]

\[\text{case 0}\]

\[\text{have } \text{Suc} \, 0 \leq \text{Suc} \, 0 \text{ ..}\]

\[\text{moreover have 0 < Suc 0 ..}\]

\[\text{moreover from 0 have } f \, (\text{Suc} \, 0) = f \, 0 \text{ by simp}\]

\[\text{ultimately show } ?\text{case by iprover}\]

next

\[\text{case } (\text{Suc} \, n)\]

\[\text{from search } [OF \text{ nat-eq-dec}] \text{ show } ?\text{case}\]

\[\text{proof}\]

\[\text{assume } \exists j < \text{Suc} \, (\text{Suc} \, n), f \, (\text{Suc} \, (\text{Suc} \, n)) = f \, j\]

\[\text{thus } ?\text{case by (iprover intro: le-refl)}\]

next

\[\text{assume } \neg (\exists j < \text{Suc} \, (\text{Suc} \, n), f \, (\text{Suc} \, (\text{Suc} \, n)) = f \, j)\]

\[\text{hence nex: } \forall j < \text{Suc} \, (\text{Suc} \, n), f \, (\text{Suc} \, (\text{Suc} \, n)) \neq f \, j \text{ by iprover}\]

\[\text{let } \text{if } = \lambda i. \text{if } f \, i = \text{Suc} \, n \text{ then } f \, (\text{Suc} \, (\text{Suc} \, n)) \text{ else } f \, i\]

\[\text{have } \forall i. i \leq \text{Suc} \, n \Rightarrow \text{if } i \leq n\]

\[\text{proof ~}\]

\[\text{fix } i \text{ assume } i: i \leq \text{Suc} \, n\]

\[\text{show } ?\text{thesis } i\]

\[\text{proof } (\text{cases } f \, i = \text{Suc} \, n)\]

\[\text{case True}\]

\[\text{from } i \text{ and } \text{nex} \text{ have } f \, (\text{Suc} \, (\text{Suc} \, n)) \neq f \, i \text{ by simp}\]

\[\text{with True have } f \, (\text{Suc} \, (\text{Suc} \, n)) \neq \text{Suc} \, n \text{ by simp}\]

\[\text{moreover from } \text{Suc} \text{ have } f \, (\text{Suc} \, (\text{Suc} \, n)) \leq \text{Suc} \, n \text{ by simp}\]

\[\text{ultimately have } f \, (\text{Suc} \, (\text{Suc} \, n)) \leq n \text{ by simp}\]

\[\text{with True show } ?\text{thesis by simp}\]

next

\[\text{case False}\]

\[\text{from } \text{Suc} \text{ and } i \text{ have } f \, i \leq \text{Suc} \, n \text{ by simp}\]

\[\text{with False show } ?\text{thesis by simp}\]
qed
qed
hence \( \exists i, j\) \( i \leq Suc n \land j < i \land f i = f j \) by (rule Suc)
then obtain \( i, j \) where \( i \leq Suc n \) and \( j < i \) and \( f i = f j \)
by \( \text{iprover} \)
have \( f i = f j \)
proof \( (\text{cases} f i = Suc n) \)
case True
show \( \text{?thesis} \)
proof \( (\text{cases} f j = Suc n) \)
assume \( f j = Suc n \)
with True show \( \text{?thesis} \) by simp
next
assume \( f j \neq Suc n \)
moreover from \( i \) \( ji \) \( nex \) have \( f (Suc (Suc n)) \neq f j \) by simp
ultimately show \( \text{?thesis} \) using True \( f \) by simp
qed
next
case False
show \( \text{?thesis} \)
proof \( (\text{cases} f j = Suc n) \)
assume \( f j = Suc n \)
moreover from \( i \) \( nex \) have \( f (Suc (Suc n)) \neq f i \) by simp
ultimately show \( \text{?thesis} \) using False \( f \) by simp
next
assume \( f j \neq Suc n \)
with False \( f \) show \( \text{?thesis} \) by simp
qed
qed
moreover from \( i \) have \( i \leq Suc (Suc n) \) by simp
ultimately show \( \text{?thesis} \) using \( ji \) by \( \text{iprover} \)
qed
qed

\textbf{extract} pigeonhole pigeonhole-slow

The programs extracted from the above proofs look as follows:

\textit{pigeonhole} \equiv
\begin{align*}
\lambda x. \text{nat-induct-P} & x \ (\lambda x. \ (Suc 0, 0)) \\
(\lambda x \ H2 \ xa. & \text{nat-induct-P} \ (Suc (Suc x)) \ \text{default} \\
(\lambda x \ H2. & \text{case search} \ (Suc x) \\
& \ (\lambda xaa. \ \text{nat-eq-dec} \ (xa \ (Suc x)) \ (xa \ xaa)) \ \text{of} \\
& None \Rightarrow \text{let} \ (x, y) = H2 \ \text{in} \ (x, y) \ | \ \text{Some} \ p \Rightarrow (Suc x, p)))
\end{align*}

\textit{pigeonhole-slow} \equiv
\begin{align*}
\lambda x. \text{nat-induct-P} & x \ (\lambda x. \ (Suc 0, 0)) \\
(\lambda x \ H2 \ xa.
\end{align*}

24
The program for searching for an element in an array is

\[
\text{search} \equiv \lambda x \, H. \ \text{nat-induct-P} \ x \ \text{None}
\]

\[
\begin{align*}
\text{let } (x, y) &= (\lambda i. \ \text{if } xa \ i = \text{Suc} \ x \ \text{then } xa \ (\text{Suc} \ x) \ \text{else } xa \ i) \\
\text{in } (x, y)
\end{align*}
\]

| Some \ p \Rightarrow (\text{Suc} \ (\text{Suc} \ x), \ p) |

The correctness statement for \textit{pigeonhole} is

\[
(\forall i. \ i \leq \text{Suc} \ n \Rightarrow f \ i \leq n) \ \Rightarrow \\
\text{fst} \ (\text{pigeonhole} \ n \ f) \leq \text{Suc} \ n \ \land \\
\text{snd} \ (\text{pigeonhole} \ n \ f) < \text{fst} \ (\text{pigeonhole} \ n \ f) \ \land \\
f \ (\text{fst} \ (\text{pigeonhole} \ n \ f)) = f \ (\text{snd} \ (\text{pigeonhole} \ n \ f))
\]

In order to analyze the speed of the above programs, we generate ML code from them.

\textbf{instantiation} \ \textit{nat} :: \ \textit{default} \\
\begin{verbatim}
begin \\
definition default = (0::nat) \\
instance .. \\
end
\end{verbatim}

\textbf{instantiation} \ \textit{prod} :: (\textit{default}, \textit{default}) \ \textit{default} \\
\begin{verbatim}
begin \\
definition default = (\text{default}, \text{default}) \\
instance .. \\
end
\end{verbatim}

\textbf{definition} \\
\begin{verbatim}
test \ n \ u = \text{pigeonhole} \ (\text{nat-of-integer} \ n) \ (\lambda m. \ m - 1) \\
definition \\
\begin{verbatim}
test’ \ n \ u = \text{pigeonhole-slow} \ (\text{nat-of-integer} \ n) \ (\lambda m. \ m - 1) \\
definition \\
\begin{verbatim}
test’’ \ u = \text{pigeonhole} \ 8 \ (\text{List.nth} \ [0, \ 1, \ 2, \ 3, \ 4, \ 5, \ 6, \ 3, \ 7, \ 8])
\end{verbatim}
\end{verbatim}
\end{verbatim}
7 Euclid’s theorem

theory Euclid
imports
  ~~/src/HOL/Number-Theory/UniqueFactorization
  Util
  ~~/src/HOL/Library/Code-Target-Numeral
begin

A constructive version of the proof of Euclid’s theorem by Markus Wenzel
and Freek Wiedijk [4].

lemma factor-greater-one1: \( n = m \cdot k \implies m < n \implies k < n \implies \text{Suc} 0 < m \)
  by (induct m) auto

lemma factor-greater-one2: \( n = m \cdot k \implies m < n \implies k < n \implies \text{Suc} 0 < k \)
  by (induct k) auto

lemma prod-mn-less-k:
  \( (0::nat) < n \implies 0 < k \implies \text{Suc} 0 < m \implies m \cdot n = k \implies n < k \)
  by (induct m) auto

lemma prime-eq: \( \text{prime} (p::nat) = (1 < p \land (∀m. m dvd p \implies 1 < m \implies m = p)) \)
  apply (simp add: prime-nat-def)
  apply (rule iffI)
  apply blast
  apply (erule conjE)
  apply (rule conjI)
  apply assumption
  apply (rule allI impI)+
  apply (erule allE)
  apply (erule impE)
  apply assumption
  apply (case_tac m=0)
  apply simp
  apply (case_tac m=\text{Suc} 0)
  apply simp
apply simp

lemma prime-eq': prime (p::nat) = (1 < p ∧ (∀ m k. p = m * k → 1 < m → m = p))
  by (simp add: prime-eq dvd-def HOL.all-simps [symmetric] del: HOL.all-simps)

lemma not-prime-ex-mk:
  assumes n: Suc 0 < n
  shows (∃ m k. Suc 0 < m ∧ Suc 0 < k ∧ m < n ∧ k < n ∧ n = m * k) ∨
  prime n
proof -
  { fix k
    from nat-eq-dec
    have (∃ m<n. n = m * k) ∨ ¬ (∃ m<n. n = m * k)
      by (rule search)
  }
  hence (∃ k<n. ∃ m<n. n = m * k) ∨ ¬ (∃ k<n. ∃ m<n. n = m * k)
    by (rule search)
thus ?thesis
proof
  assume ∃ k<n. ∃ m<n. n = m * k
  then obtain k m where k: k<n and m: m<n and nmk: n = m * k
    by iprover
  moreover from nmk m k have Suc 0 < m by (rule factor-greater-one1)
  moreover from nmk m k have Suc 0 < k by (rule factor-greater-one2)
  ultimately show ?thesis using k m nmk by iprover
next
  assume ¬ (∃ k<n. ∃ m<n. n = m * k)
  hence A: ∀ k<n. ∀ m<n. n ≠ m * k by iprover
  have ∀ m k. n = m * k → Suc 0 < m → m = n
  proof (intro allI impI)
    fix m k
    assume nmk: n = m * k
    assume m: Suc 0 < m
    from n m nmk have k: 0 < k
      by (cases k) auto
    moreover from n have n: 0 < n by simp
    moreover note m
    moreover from nmk have m * k = n by simp
    ultimately have kn: k < n by (rule prod-ran-less-k)
    show m = n
    proof (cases k = Suc 0)
      case True
      with nmk show ?thesis by (simp only: mult-Suc-right)
    next
case False
    from m have 0 < m by simp

27
moreover note \( n \)
moreover from \( \text{False} n \) have \( \text{Suc} 0 < k \) by auto
moreover from \( nmk \) have \( k \ast m = n \) by (simp only: ac-simps)
ultimately have \( mn: m < n \) by (rule prod-mn-less-k)
with \( kn A \) have \( \text{thesis} \) by sprover
qed
qed
with \( n \) have \( \text{prime} n \)
  by (simp only: prime-eq One-nat-def simp-thms)
thus \( \text{thesis} \)
qed
qed

lemma dvd-factorial: \( 0 < m \implies m \leq n \implies m \text{ dvd} \text{ fact} (n::nat) \)
proof (induct n rule: nat-induct)
  case 0
  then show \( \text{thesis} \) by simp
next
case (Suc n)
  from \( (m \leq \text{Suc} n) \) show \( \text{thesis} \)
  proof (rule le-SucE)
    assume \( m \leq n \)
    with \( (0 < m) \) have \( m \text{ dvd} \text{ fact} n \) by (rule Suc)
    then have \( m \text{ dvd} (\text{fact} n \ast \text{Suc} n) \) by (rule dvd-mult2)
    then show \( \text{thesis} \) by (simp add: mult.commute)
  next
    assume \( m = \text{Suc} n \)
    then have \( m \text{ dvd} (\text{fact} n \ast \text{Suc} n) \)
      by (auto intro: dvdI simp: ac-simps)
    then show \( \text{thesis} \) by (simp add: mult.commute)
  qed
qed

lemma dvd-prod [iff]: \( n \text{ dvd} (\text{PROD} m::nat:#multiset-of (n # ns). m) \)
  by (simp add: msetprod-Un msetprod-singleton)

definition all-prime :: \( \text{nat} \) \( \text{list} \) \( \Rightarrow \) \( \text{bool} \) where
  all-prime \( ps \) \( \iff \) (\( \forall p \in \text{set} \ ps \). prime \( p \))

lemma all-prime-simps:
  all-prime [\]
  all-prime (\( p \# ps \)) \( \iff \) prime \( p \) \( \land \) all-prime \( ps \)
  by (simp-all add: all-prime-def)

lemma all-prime-append:
  all-prime (\( ps \circ qs \)) \( \iff \) all-prime \( ps \land all-prime \ qs \)
  by (simp add: all-prime-def ball-Un)

lemma split-all-prime:
assumes all-prime ms and all-prime ns
shows ∃ qs. all-prime qs ∧ (PROD m::nat:#multiset-of qs. m) = (PROD m::nat:#multiset-of ms. m) * (PROD m::nat:#multiset-of ns. m) (is ∃ qs. ?P qs ∧ ?Q qs)
proof
  from assms have all-prime (ms @ ns)
  by (simp add: all-prime-append)
moreover from assms have (PROD m::nat:#multiset-of (ms @ ns). m) = (PROD m::nat:#multiset-of ms. m) * (PROD m::nat:#multiset-of ns. m)
by (simp add: msetprod-Un)
ultimately have ?P (ms @ ns) ∧ ?Q (ms @ ns) ..
then show ?thesis ..
qed

lemma all-prime-nempty-g-one:
assumes all-prime ps and ps ≠ []
shows Suc 0 < (PROD m::nat:#multiset-of ps. m)
using (ps ≠ [] : all-prime ps)
unfolding One-def by (induct ps rule: list-nondef)
  (simp-all add: all-prime-simps msetprod-singleton msetprod-Un prime-gt-1-nat less-1-mult del: One-def)

lemma factor-exists: Suc 0 < n =⇒ (∃ ps. all-prime ps ∧ (PROD m::nat:#multiset-of ps. m) = n)
proof (induct n rule: nat-wf-ind)
case (Suc n)
from (Suc 0 < n)
have (∃ m k. Suc 0 < m ∧ Suc 0 < k ∧ m < n ∧ k < n ∧ n = m * k) ∨ prime
n
  by (rule not-prime-ex-mk)
then show ?case
proof
  assume ∃ m k. Suc 0 < m ∧ Suc 0 < k ∧ m < n ∧ k < n ∧ n = m * k
  then obtain m k where: Suc 0 < m and k: Suc 0 < k and mn: m < n
  and kn: k < n and nmk: n = m * k by iprover
  from mn and m have ∃ ps. all-prime ps ∧ (PROD m::nat:#multiset-of ps. m) = m by (rule 1)
  then obtain ps1 where all-prime ps1 and prod-ps1-m: (PROD m::nat:#multiset-of ps1. m) = m
  by iprover
  from kn and k have ∃ ps. all-prime ps ∧ (PROD m::nat:#multiset-of ps. m) = k by (rule 1)
  then obtain ps2 where all-prime ps2 and prod-ps2-k: (PROD m::nat:#multiset-of ps2. m) = k
  by iprover
  from (all-prime ps1) (all-prime ps2)
  have ∃ ps. all-prime ps ∧ (PROD m::nat:#multiset-of ps. m) = (PROD m::nat:#multiset-of ps1. m) * (PROD m::nat:#multiset-of ps2. m)
  by (rule split-all-prime)
with prod-ps1-m prod-ps2-k nmk show ?thesis by simp

next
   assume prime n then have all-prime [n] by (simp add: all-prime-simps)
   moreover have (PROD m::nat:#multiset-of [n]. m) = n by (simp add: msetprod-singleton)
   ultimately have all-prime [n] ∧ (PROD m::nat:#multiset-of [n]. m) = n ..
   then show ?thesis ..
   qed

qed

lemma prime-factor-exists:
   assumes N: (1::nat) < n
   shows ∃ p. prime p ∧ p dvd n
   proof –
      from N obtain ps where all-prime ps
      and prod-ps: n = (PROD m::nat:#multiset-of ps. m) using factor-exists
      by simp iprover
      with N have ps ≠ [] by (auto simp add: all-prime-nempty-g-one msetprod-empty)
      then obtain p qs where ps = p # qs by (cases ps) simp
      with ⟨all-prime ps⟩ have prime p by (simp add: all-prime-simps)
      moreover from ⟨all-prime ps⟩ ps prod-ps
      have p dvd n by (simp only: dvd-prod)
      ultimately show ?thesis by iprover
   qed

Euclid’s theorem: there are infinitely many primes.

lemma Euclid: ∃ p::nat. prime p ∧ n < p
   proof –
      let ?k = fact n + (1::nat)
      have 1 < ?k by simp
      then obtain p where prime: prime p and dvd: p dvd ?k using prime-factor-exists
      by iprover
      have n < p
      proof –
      have ¬ p ≤ n
      proof
         assume pn: p ≤ n
         from prime p have 0 < p by (rule prime-gt-0-nat)
         then have p dvd fact n using pn by (rule dvd-factorial)
         with dvd have p dvd ?k − fact n by (rule dvd-diff-nat)
         then have p dvd 1 by simp
         with prime show False by auto
      qed
      then show ?thesis by simp
      qed
      with prime show ?thesis by iprover
      qed
The program extracted from the proof of Euclid’s theorem looks as follows.

\[ \text{Euclid} \equiv \lambda x. \text{prime-factor-exists} (\text{fact } x + 1) \]

The program corresponding to the proof of the factorization theorem is

\[ \text{factor-exists} \equiv \lambda x. \text{nat-wf-ind-P } x \]
\[
(\lambda x \text{H2}.
\quad \text{case not-prime-ex-mk } x \text{ of None } \Rightarrow [x]
\quad | \text{ Some } p \Rightarrow \text{ let } (x, y) = p \text{ in split-all-prime } (\text{H2 } x) (\text{H2 } y))
\]

**instantiation** nat :: default

**definition** default = (0::nat)

**instance** ..

**end**

**instantiation** list :: (type) default

**definition** default = []

**instance** ..

**end**

**primrec** iterate :: nat \(\Rightarrow\) ('a \(\Rightarrow\) 'a) \(\Rightarrow\) 'a \(\Rightarrow\) 'a list \textbf{where}

iterate 0 f x = []
\[
| \text{iterate } (\text{Suc } n) \ f \ x = (\text{let } y = f \ x \text{ in } y \# \text{iterate } n \ f \ y)
\]

**lemma** factor-exists 1007 = [53, 19] \textbf{by eval}
**lemma** factor-exists 567 = [7, 3, 3, 3] \textbf{by eval}
**lemma** factor-exists 345 = [23, 5, 3] \textbf{by eval}
**lemma** factor-exists 999 = [37, 3, 3, 3] \textbf{by eval}
**lemma** factor-exists 876 = [73, 3, 2, 2] \textbf{by eval}

**lemma** iterate 4 Euclid 0 = [2, 3, 7, 71] \textbf{by eval}

**end**
References


