Examples for program extraction in Higher-Order Logic

Stefan Berghofer
August 27, 2014

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 Main
begin

Decidability of equality on natural numbers.

lemma nat-eq-dec: \(\forall n::\text{nat}. \ m = n \lor m \neq n\)
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 \( R: \forall x::\text{nat}. \ (\forall y. \ y < x \implies P y) \implies P x \)

shows \( P z \)

**proof** (rule \( R \))

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

**proof** (induct \( z \))

case 0

thus ?case by simp

next

case (Suc \( n \) \( y \))

from nat-eq-dec show ?case

**proof**

assume ny: \( n = y \)

have \( P n \)

by (rule \( R \)) (rule Suc)

with ny show ?case by simp

next

assume \( n \neq y \)

with 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 \( \text{dec}: \forall x::\text{nat}. \ P x \lor \neg P x \)

shows \( (\exists x<z. \ P x) \lor \neg (\exists x<z. \ P x) \)

**proof** (induct \( y \))

case 0 show ?case by simp

next

case (Suc \( z \))

thus ?case

**proof**

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

then obtain \( x \) where le: \( x < z \) and \( P: P x \) by iprover

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

with \( P \) show ?case by iprover

next

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

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

**proof**

assume \( P: P z \)

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

with \( P \) show ?thesis by iprover

next

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

proof
 assume \exists x < \text{Suc } z. \ P x
 then obtain x where le: x < \text{Suc } z and \ P x by iprover
 have x < z
 proof (cases x = z)
   case True
   with nP and \ P show ?thesis by simp
 next
   case False
   with le show ?thesis by simp
 qed
 with \ P have \exists x < z. \ P x by iprover
 with nex show False ..
 qed
 thus ?case by iprover
 qed
 qed

end

2 Quotient and remainder

theory QuotRem
 imports Util
 begin

Derivation of quotient and remainder using program extraction.

theorem division: \exists r q. a = \text{Suc } b * q + r \land r \leq b
 proof (induct a)
 case 0
 have 0 = \text{Suc } b * 0 + 0 \land 0 \leq b by simp
 thus ?case by iprover
 next
 case (Suc a)
 then obtain r q where I: a = \text{Suc } b * q + r and r \leq b by iprover
 from nat-eq-dec show ?case
 proof
   assume r = b
   with I have Suc a = Suc b * (Suc q) + 0 \land 0 \leq b by simp
   thus ?case by iprover
 next
 assume r \neq b
 with r \leq b have r < b by (simp add: order-less-le)
 with I have Suc a = Suc b * q + (Suc r) \land (Suc r) \leq b by simp
 thus ?case by iprover
 qed
 qed

3
extract division

The program extracted from the above proof looks as follows

\[
\begin{align*}
\text{division} & \equiv \\
\text{\lambda x. } & (\text{nat-induct-P x (0, 0)} \\
& (\lambda a H. \text{let } (x, y) = H \\
& \text{in case } \text{nat-eq-dec x xa of Left } \Rightarrow (0, \text{Suc y)} \\
& \text{| Right } \Rightarrow (\text{Suc x, y))}
\end{align*}
\]

The corresponding correctness theorem is

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

\[\text{lemma division 9 2 } = (0, 3) \text{ by eval}\]

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: \text{n = Suc m} \ast q + r and h2: \text{r} \leq m
 by iprover
 show \text{?case}
 proof (cases r)
  case 0
  with h1 have \text{Suc m} \ast q = n by simp
  moreover have \text{Suc m} \ast l = \text{Suc m} by simp
  moreover {
    fix l2 have \text{l \ast l1 = n \Rightarrow l \ast l2 = Suc m \Rightarrow l \leq Suc m}
    by (cases l2) simp-all }
  ultimately show \text{?thesis} by iprover
 next
  case (Suc nat)
  with h2 have \text{nat < m} by simp
  moreover from h have \text{Suc nat < Suc m} by simp
  ultimately have \exists k m1 r1. k \ast m1 = Suc m \land k \ast r1 = Suc nat \land
  (\forall l l1 l2. l \ast l1 = \text{Suc m} \Rightarrow l \ast l2 = \text{Suc nat} \Rightarrow l \leq k)
  by (rule 1)
\end{proof}
then obtain $k \; m_1 \; r_1$ where
\[ h_1': k \cdot m_1 = \text{Suc} \; m \]
and $h_2': k \cdot r_1 = \text{Suc} \; \text{nat}$
and $h_3': \forall l \; l_1 \; l_2. \; l \cdot l_1 = m \implies l \cdot l_2 = \text{Suc} \; \text{nat} \implies l \leq k$
by iprove
have $m_n: \text{Suc} \; m < n$ by (rule 1)
from $h_1 \; h_1' \; h_2' \; \text{Suc}$ have $k \cdot (m_1 \cdot q + r_1) = n$
moreover have $\forall l \; l_1 \; l_2. \; l \cdot l_1 = n \implies l \cdot l_2 = \text{Suc} \; m \implies l \leq k$
proof -
fix $l \; l_1 \; l_2$
assume $l1n: l \cdot l_1 = n$
assume $l2m: l \cdot l_2 = \text{Suc} \; m$
moreover have $l \cdot (l_1 - l_2 \cdot q) = \text{Suc} \; \text{nat}$
by (simp add: \text{diff-mult-distrib}2 \; h1 \; \text{Suc} \; \text{symmetric} \; mn \; l1n \; l2m \; \text{symmetric})
ultimately show $l \leq k$ by (rule $h3'$)
qed
ultimately show ?thesis using $h1'$ by iprove
qed
qed

extract greatest-common-divisor

The extracted program for computing the greatest common divisor is

\begin{align*}
greatest-common-divisor & \equiv \\
& \lambda x. \text{nat-wf-ind-P} \; x \\
& \quad (\lambda x. H2 \; xa. \\
& \quad \quad \text{let} \; (xa, y) = \text{division} \; xa \; x \\
& \quad \quad \text{in} \; \text{nat-exhaust-P} \; xa \; (\text{Suc} \; x, \; y, \; 1) \\
& \quad \quad (\lambda \text{nat}. \; \text{let} \; (x, \; ya) = H2 \; \text{nat} \; (\text{Suc} \; x); \; (xa, \; ya) = ya \\
& \quad \quad \text{in} \; (x, \; xa \; * \; y \; + \; ya, \; xa))}
\end{align*}

instantiation $\text{nat} :: \text{default}$
begin

definition default = $0::\text{nat}$

instance ..

end

instantiation $\text{prod} :: (default, \; default)$ default
begin

definition default = $(\text{default}, \; \text{default})$

instance ..

end

5
instantiation fun :: (type, default) default
begin

definition default = (\lambda x. default)

instance ..
end

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

4 Warshall’s algorithm

theory Warshall
imports Main
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 ∧
    list-all (\x. 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]:
  \forall 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-snoc [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 \Rightarrow (\forall x. P x \Rightarrow Q x) \Rightarrow list-all Q xs
proof –
  assume PQ: \forall x. P x \Rightarrow Q x
  show list-all P xs \Rightarrow 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: \forall p. is-path r p i j k \Rightarrow 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: \forall p. is-path r p 0 j k \Rightarrow 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 \Rightarrow is-path' r i ys k \Rightarrow
  is-path' r j (xs @ i # ys) k
proof –
  assume pys: is-path' r i ys k
  show \forall j. is-path' r j xs i \Rightarrow 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
qed
theorem lemma3:
\[ \forall p q. \text{is-path } r p i j i \implies \text{is-path } r q i k \]
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)
simp
apply (erule conjE)+
apply (rule list-all-lemma)
simp
apply (rule is-path' conc)
apply assumption+
done

theorem lemma5:
\[ \forall p. \text{is-path } r p (\text{Suc } i) j k \implies \sim \text{is-path } r p i j k \]
(\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 list-all (\lambda x. x < i) xs
show (\exists ys. list-all (\lambda x. x < i) ys \land is-path' r j ys i) \land
(\exists ys. list-all (\lambda x. x < i) ys \land is-path' r i ys k)
proof
show \[ \forall j. \text{list-all } (\lambda x. x < \text{Suc } i) j \implies \text{is-path' } r j k \]
\neg list-all (\lambda x. x < i) j \implies
\exists ys. list-all (\lambda x. x < i) ys \land is-path' r j ys i (is PROP ?ih xs)
proof (induct xs)
case Nil
thus ?case by simp
next
case (Cons a as j)
show ?case
proof (cases a=i)
case True
show ?thesis
proof
from True and Cons have r j i = T by simp
thus list-all (\lambda x. x < i) [] \land 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 \( \text{thesis} \)
  proof
    from Cons False ys
    show \( \text{list-all} (\lambda x. x < i) (a\#ys) \land \text{is-path'} r j (a\#ys) \) \( i \) by simp
  qed
  qed
  qed
  show \( \forall k. \text{list-all} (\lambda x. x < \text{Suc} i) \) \( xs \Rightarrow \text{is-path'} r j \) \( xs \) \( k \Rightarrow \)
  \( \neg \text{list-all} (\lambda x. x < i) \) \( xs \Rightarrow \)
  \( \exists ys. \text{list-all} (\lambda x. x < i) \) \( ys \land \text{is-path'} r i \) \( ys k \) (is PROP \( ?\text{ih} xs \))
  proof (induct \( xs \) rule: rev-induct)
  case Nil
  thus \( ?\text{case} \) by simp
  next
  case (snoc \( a \) as \( k \))
  show \( ?\text{case} \)
  proof (cases \( a = i \))
    case True
    show \( ?\text{thesis} \)
    proof
      from True and snoc have \( r i k = T \) by simp
      thus \( \text{list-all} (\lambda x. x < i) (\) \( \) \( ) \land \text{is-path'} r i (\) \( \) \( ) k \) by simp
    qed
    next
    case False
    have PROP \( ?\text{ih} as \) 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 \( ?\text{thesis} \)
    proof
      from snoc False ys
      show \( \text{list-all} (\lambda x. x < i) (ys @ [a]) \land \text{is-path'} r j (ys @ [a]) k \) by simp
    qed
  qed
  qed
qed
theorem lemma5':
\( \forall p. \isPath r p (\text{Suc} i) j k \implies \neg \isPath r p i j k \implies \neg (\forall q. \neg \isPath r q i j i) \land \neg (\forall q'. \neg \isPath r q' i i k) \)

by (iprover dest: lemma5)

theorem warshall:
\( \forall j k. \neg (\exists p. \isPath r p i j k) \lor (\exists p. \isPath r p i j k) \)

proof (induct i)
  case (0 j k)
  show ?case
  proof (cases r j k)
    assume r j k = T
    hence \( \isPath r (j, [], k) 0 j k \)
    by (simp add: is-path-def)
    hence \( \exists p. \isPath r p 0 j k \) ..
    thus ?thesis ..
  next
    assume r j k = F
    hence \( r j k \sim = T \) by simp
    hence \( \neg (\exists p. \isPath r p 0 j k) \)
    by (iprover dest: lemma2)
    thus ?thesis ..
  qed
  next
  case (Suc i j k)
  thus ?case
  proof
    assume h1: \( \neg (\exists p. \isPath r p i j k) \)
    from Suc show ?case
    proof
      assume \( \neg (\exists p. \isPath r p i j i) \)
      with h1 have \( \neg (\exists p. \isPath r p (\text{Suc} i) j k) \)
      by (iprover dest: lemma5')
      thus ?case ..
    next
      assume \( \exists p. \isPath r p i j i \)
      then obtain p where h2: \( \isPath r p i j i \) ..
      from Suc show ?case
      proof
        assume \( \neg (\exists p. \isPath r p i i k) \)
        with h1 have \( \neg (\exists p. \isPath r p (\text{Suc} i) j k) \)
        by (iprover dest: lemma5')
        thus ?case ..
      next
        assume \( \exists q. \isPath r q i i k \)
        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 \( \exists pq. \isPath r pq (\text{Suc} i) j k \) ..
thus \textit{case ..}
\textbf{qed}
\textbf{qed}
next
\textbf{assume} \exists p. \textit{is-path} r p i j k
\textbf{hence} \exists p. \textit{is-path} r p (Suc i) j k
\textbf{by} (iprover intro: lemma1)
\textbf{thus} \textit{case ..}
\textbf{qed}
\textbf{qed}

\textbf{extract} \textit{warshall}

The program extracted from the above proof looks as follows

\textit{warshall} \equiv
\lambda x xax xaaa.
\textit{nat-induct-P} xu
(\lambda xaxaxa of T \Rightarrow \textit{Some} (xa, [], xaa) | F \Rightarrow \textit{None})
(\lambda H2 xaxaa.
\textit{case} H2 xaa of
None \Rightarrow
\textit{case} H2 xa x of \textit{None} \Rightarrow \textit{None}
| \textit{Some} q \Rightarrow
\textit{case} H2 x axaa of \textit{None} \Rightarrow \textit{None} | \textit{Some} qa \Rightarrow \textit{Some} (conc q qa)
| \textit{Some} q \Rightarrow \textit{Some} q)
xaa xaaa

The corresponding correctness theorem is

\textit{case warshall} r i j k of \textit{None} \Rightarrow \forall x. \neg \textit{is-path} r x i j k
| \textit{Some} q \Rightarrow \textit{is-path} r q i j k

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

\section{5 Higman’s lemma}

\textbf{theory} \textit{Higman}
\textbf{imports} \textit{Main}
\textbf{begin}

Formalization by Stefan Berghofer and Monika Seisenberger, based on Coquand and Fridlender [2].
\textbf{datatype} letter \textit{=} A \textit{\mid} B

\textbf{inductive} emb :: letter list \Rightarrow letter list \Rightarrow bool
\textbf{where}

11
\begin{verbatim}
emb0 [Pure.intro]: emb [] bs
| emb1 [Pure.intro]: emb as bs \implies emb (b # bs)
| emb2 [Pure.intro]: emb as bs \implies emb (a # as) (a # bs)

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

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

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

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

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

theorem prop1: bar ([] # ws) by iprover

theorem lemma1: L as ws \implies L (a # as) ws
by (erule L.induct, iprover+)

lemma lemma2': R a vs ws \implies L as vs \implies L (a # as) ws
apply (induct set: R)
apply (erule L.cases)
apply simp+
apply (erule L.cases)
apply simp-all
apply (rule L0)
apply (erule emb2)
apply (erule L1)
done
\end{verbatim}
lemma lemma2: $R \, a \, vs \, ws \implies \text{good} \, vs \implies \text{good} \, ws$
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?: $T \, a \, vs \, ws \implies \text{L} \, as \, vs \implies \text{L} \, (a \neq as) \, ws$
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: $T \, a \, ws \, zs \implies \text{good} \, ws \implies \text{good} \, zs$
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: $R \, a \, ws \, zs \implies ws \neq [] \implies T \, a \, ws \, zs$
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: (a::letter) ≠ b → c ≠ a → c = 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: (a::letter) = b ∨ a ≠ 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: a ≠ b and bar: bar xs
shows ∀ ys zs. bar ys = T a xs zs = T b ys zs = bar zs using bar

proof induct
  fix xs zs assume T a xs zs and good xs
  hence good zs by (rule lemma3)
  then show bar zs by (rule bar1)
next
  fix xs ys assume I: ∃ w ys zs. bar ys --> T a (w # xs) zs --> T b ys zs --> bar zs
  assume bar ys
  thus ∃ zs. T a xs zs --> T b ys zs --> bar zs

proof induct
  fix ys zs assume T b ys zs and good ys
  then have good zs by (rule lemma3)
  then show bar zs by (rule bar1)
next
  fix ys zs assume I': ∃ w zs. T a xs zs --> T b (w # ys) zs --> bar zs
and ys: ∃ w. bar (w # ys) and T b: T a xs zs and T b: T b ys zs
  show bar zs

proof (rule bar2)
  fix w
show bar \((w \neq zs)\)
proof (cases \(w\))
  case Nil
  thus \(\text{thesis}\) by simp (rule prop1)
next
  case (Cons \(c\) \(cs\))
  from letter-eq-dec show \(\text{thesis}\)
  proof
    assume \(ca\): \(c = a\)
    from \(ab\) have \(\text{bar }((a \# cs) \# zs)\) by (iprover intro: I ys Ta Tb)
    thus \(\text{thesis}\) by (simp add: Cons ca)
  next
    assume \(c \neq a\)
    with \(ab\) have \(\text{cb}: \(c = b\)\) by (rule letter-neq)
    from \(ab\) have \(\text{bar }((b \# cs) \# zs)\) by (iprover intro: I’ Ta Tb)
    thus \(\text{thesis}\) by (simp add: Cons cb)
  qed
  qed
  qed
qed

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

thus \( \text{thesis by (iprover intro: prop2 Cons xsb xsn R T)} \)

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

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
\[
\text{case } (\text{good1 ws w}) \\
\text{thus } ?\text{case by simp} \\
\text{qed}
\]

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

\textbf{Strong version}: yields indices of words that can be embedded into each other.

\textbf{theorem} higman-idx: \exists (i::nat) j. \text{emb (f i) (f j) \land i < j}
\textbf{proof} (\text{rule bar-idx})
\begin{enumerate}
  \item \text{show bar [[] by (rule higman)}
  \item \text{show is-prefix [[] f by simp}
\end{enumerate}
\text{qed}

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

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

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

\section{5.1 Extracting the program}
\textbf{declare} \text{R.induct [ind-realizer]}
\textbf{declare} \text{T.induct [ind-realizer]}
\textbf{declare} \text{L.induct [ind-realizer]}

17
declare good.induct [ind-realizer]
declare bar.induct [ind-realizer]

extract higman-idx

Program extracted from the proof of higman-idx:

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

Corresponding correctness theorem:

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

Program extracted from the proof of higman:

\[ \text{higman} \equiv \]
\[ \text{bar2} \ [] \ (\text{rec-list} (\text{prop1} \ [])) (\lambda a \ w \ H. \ \text{prop3} \ a \ [a \ # \ w] \ H \ (R1 \ [] \ [] \ w \ R0)) \]

Program extracted from the proof of prop1:

\[ \text{prop1} \equiv \]
\[ \lambda x. \text{bar2} \ [] \ (x \ # \ x) \ (\lambda w. \text{bar1} \ (w \ # \ [] \ # \ x) \ (\text{good0} \ w \ ([] \ # \ x) \ (L0 \ [] \ x))) \]

Program extracted from the proof of prop2:

\[ \text{prop2} \equiv \]
\[ \lambda x \ xa \ xaa \ xaaa \ H. \]
\[ \text{rec-barT} \ (\lambda \ ws \ xa \ xaa \ xaaa \ Ha \ Haa. \ \text{bar1} \ xaaa \ (\text{lemma3} \ x \ Ha \ xa)) \]
\[ (\lambda ws \ xa \ r \ xaaa \ xaab \ H. \]
\[ \text{rec-barT} \ (\lambda ws \ x \ xaa \ Ha. \ \text{bar1} \ xaa \ (\text{lemma3} \ xa \ Ha \ x)) \]
\[ (\lambda ws \ xa \ ra \ xaaa \ xaaab \ H. \]
\[ \text{bar2} \ xaaa \]
\[ (\lambda w. \ \text{case} \ w \ \text{of} \ [] \Rightarrow \text{prop1} \ xaaa \]
\[ | \ a \ # \ \text{list} \Rightarrow \]
\[ \text{case} \ \text{letter-eq-dec} \ a \ \text{of} \]
\[ \text{Left} \Rightarrow \]
\[ r \ \text{list} \ ws \ a \ ((x \ # \ \text{list}) \ # \ xaaa) \ (\text{bar2} \ ws \ xaa) \]
\[ (T1 \ ws \ xaaa \ \text{list} \ H) \ (T2 \ x \ ws \ xaa \ \text{list} \ Ha) \]
\[ | \ \text{Right} \Rightarrow \]
\[ ra \ \text{list} \ ((x \ # \ \text{list}) \ # \ xaaa) \]
\[ (T2 \ xa \ ws \ xaaa \ \text{list} \ H) \ (T1 \ ws \ xaaa \ \text{list} \ Ha)) \]
\[ H \ xaab) \]
\[ H \ xaaa \ xaaa \]

Program extracted from the proof of prop3:

\[ \text{prop3} \equiv \]
\[ \lambda x \ xa \ H. \]
\[ \text{rec-barT} \ (\lambda ws \ xa \ xaa \ H. \ \text{bar1} \ xaa \ (\text{lemma2} \ x \ Ha)) \]
\((\lambda w x a r x a a H)\) bar2 xaa

(rec-list (prop1 xaa)
  (\lambda a w H a. case letter-eq-dec a x of
  Left \Rightarrow r w ((x \# w) \# xaa) (R1 ws xaa w H)
  | Right \Rightarrow prop2 a x ws ((a \# w) \# xaa) Ha (bar2 ws xa)
  (T0 x ws xaa w H) (T2 a ws xaa w (lemma4 x H))))

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 \Rightarrow Random.seed \Rightarrow letter list \times Random.seed where

mk-word-aux k = exec {
  i \leftarrow Random.range 10;
  (if i > 7 \land k > 2 \lor k > 1000 then Pair []
  else exec {
    let l = (if i mod 2 = 0 then A else B);
    ls \leftarrow mk-word-aux (Suc k);
    Pair (l \# ls)
  })
} by pat-completeness auto

termination by (relation measure ((\text{op} -) 1001)) auto

definition mk-word :: Random.seed \Rightarrow letter list \times Random.seed where

mk-word = mk-word-aux 0

primrec mk-word-s :: nat \Rightarrow Random.seed \Rightarrow letter list \times Random.seed where

mk-word-s 0 = mk-word
| mk-word-s (Suc n) = exec {
  - \leftarrow mk-word;
  mk-word-s n
}

definition g1 :: nat \Rightarrow letter list where

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

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

\textbf{end}

\section{The pigeonhole principle}

\textbf{theory} Pigeonhole
\textbf{imports} Util \simie//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} pigeonhole:
\[ \forall f. \left( \forall i. \ i \leq \text{Suc} \, n \implies f \, i \leq n \right) \implies \exists j. \ i \leq \text{Suc} \, n \land j < i \land f \, i = f \, j \]
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) ⇒
  (∀ i j. Suc k ≤ i ⇒ i ≤ Suc (Suc n) ⇒ j < i ⇒ f i ≠ f j) ⇒
  (∃ i j. i ≤ k ∧ j < i ∧ f i = f j)
  proof (induct k)
  case 0
  let ?f = λ i. if f i = Suc n then f (Suc (Suc n)) else f i
  have ¬ (∃ i j. i ≤ Suc n ∧ j < i ∧ ?f i = ?f j)
  proof
    assume ∃ 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
  qed
}
with $f_i$ 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
qed
moreover have $\forall i\ .\ i \leq \text{Suc}\ n \Rightarrow ?f_i \leq n$
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 $\exists i$ thesis $i$
  by simp
qed
hence $\exists i\ j\ .\ i \leq \text{Suc}\ n \land j < i \land ?f_i = ?f_j$
  by (rule Suc)
ultimately show $\exists$case ..
next
case $(\text{Suc}\ k)$
from search [OF nat-eq-dec] show $\exists$case
proof
assume $\exists j < \text{Suc}\ k\ .\ f\ (\text{Suc}\ k) = f_j$
thus $\exists$case by (iprover intro: le-refl)
next
assume nex: $\neg\ (\exists j < \text{Suc}\ k\ .\ f\ (\text{Suc}\ k) = f_j)$
have $\exists i\ j\ .\ i \leq k \land j < i \land f_i = f_j$
proof (rule Suc)
from 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 $\exists$thesis
proof
assume $f_i = f_j$
hence $f\ (\text{Suc}\ k) = f_j$ by (simp add: eq)
with nex and $j$ and eq show False by iprover
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
   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:

\textbf{theorem} pigeonhole-slow:
\(\forall f. (\forall i. \ i \leq \text{Suc } n \Rightarrow f i \leq n) \Rightarrow \exists i. j. \ i \leq \text{Suc } n \land j < i \land f i = f j\)
\textbf{proof} (induct \(n\))
   case \(0\)
   have \(\text{Suc } 0 \leq \text{Suc } 0\) ..
   moreover have \(0 < \text{Suc } 0\) ..
   moreover from \(0\) have \(f (\text{Suc } 0) = f 0\) by simp
   ultimately show ?case by iprover
   next
   case \((\text{Suc } n)\)
   from search [OF nat-eq-dec] show ?case
   proof
      assume \(\exists j < \text{Suc } (\text{Suc } n). f (\text{Suc } (\text{Suc } n)) = f j\)
      thus ?case by (iprover intro: le-refl)
   next
      assume \(\neg (\exists j < \text{Suc } (\text{Suc } n). f (\text{Suc } (\text{Suc } n)) = f j)\)
      hence \(\text{nex}: (\forall j < \text{Suc } (\text{Suc } n). f (\text{Suc } (\text{Suc } n)) \neq f j)\) by iprover
      let \(\text{if } = \lambda i. \text{if } f i = \text{Suc } n \text{ then } f (\text{Suc } (\text{Suc } n)) \text{ else } f i\)
      have \(\forall i. \ i \leq \text{Suc } n \Rightarrow \text{if } i \leq n\)
      proof -
         fix \(i\) assume \(i: \ i \leq \text{Suc } n\)
         show ?thesis \(i\)
         proof (cases \(f i = \text{Suc } n\))
            case \(\text{True}\)
            from \(i\) and \(\text{nex}\) have \(f (\text{Suc } (\text{Suc } n)) \neq f i\) by simp
            with \(\text{True}\) have \(f (\text{Suc } (\text{Suc } n)) \neq \text{Suc } n\) by simp
            moreover from \(\text{Suc}\) have \(f (\text{Suc } (\text{Suc } n)) \leq \text{Suc } n\) by simp
            ultimately have \(f (\text{Suc } (\text{Suc } n)) \leq n\) by simp
            with \(\text{True}\) show ?thesis by simp
         next
            case \(\text{False}\)
            from \(\text{Suc}\) and \(i\) have \(f i \leq \text{Suc } n\) by simp
            with \(\text{False}\) show ?thesis by simp
         qed
      qed
qed
qed
hence \( \exists j \cdot i \leq \text{Succ } n \land j < i \land \text{f } i = \text{f } j \) by (rule Suc)
then obtain \( i, j \) where \( i \leq \text{Succ } n \) and \( j < i \) and \( \text{f } i = \text{f } j \)
by iprover
have \( \text{f } i = \text{f } j \)
proof (cases \( \text{f } i = \text{Succ } n \))
case True
show \( \text{thesis} \)
proof (cases \( \text{f } j = \text{Succ } n \))
assume \( \text{f } j = \text{Succ } n \)
with True show \( \text{thesis} \) by simp
next
assume \( \text{f } j \neq \text{Succ } n \)
moreover from \( \text{ji} \) have \( \text{f } (\text{Succ } n) \neq \text{f } j \) by simp
ultimately show \( \text{thesis} \) using True \( \text{f} \) by simp
qed
next
case False
show \( \text{thesis} \)
proof (cases \( \text{f } j = \text{Succ } n \))
assume \( \text{f } j = \text{Succ } n \)
moreover from \( \text{i} \) have \( \text{f } (\text{Succ } n) \neq \text{f } i \) by simp
ultimately show \( \text{thesis} \) using False \( \text{f} \) by simp
qed
qed
moreover from \( \text{i} \) have \( \text{i} \leq \text{Succ } (\text{Succ } n) \) by simp
ultimately show \( \text{thesis} \) using \( \text{ji} \) by iprover
qed
qed

extract pigeonhole pigeonhole-slow

The programs extracted from the above proofs look as follows:

\[
pigeonhole \equiv \\
\lambda x. \text{nat-induct-P } x \ (\lambda x. (\text{Succ } 0, 0)) \\
\ (\lambda x \text{H2 } xa. \\
\ (\lambda x. \text{nat-induct-P } (\text{Succ } x)) \ \text{default} \\
\ (\lambda x \text{H2}. \\
\ (\lambda x \text{xa. case search } (\text{Succ } x) \\
\ (\lambda xaa. \text{nat-eq-dec } (xa \ (\text{Succ } x)) \ (xa \ xaa)) \ \text{of} \\
None \Rightarrow \text{let } (x, y) = \text{H2 } in \ (x, y) \ | \ \text{Some } p \Rightarrow (\text{Succ } x, p))\)
\]

\[
pigeonhole-slow \equiv \\
\lambda x. \text{nat-induct-P } x \ (\lambda x. (\text{Succ } 0, 0)) \\
\ (\lambda x \text{H2 } xa.
\]

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

\[
\text{search} \equiv \\
\lambda x. \text{H}.
\text{nati} \cdot \text{P} x \text{ None} \\
(\lambda y. \text{H} y \text{ of Left } \Rightarrow \text{Some } y \mid \text{Right } \Rightarrow \text{None})
\]

The correctness statement for \textit{pigeonhole} is

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

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

\textbf{instantiation} \ \textbf{nat} :: \textit{default} \\
\textbf{begin} \\
\textbf{definition} \textit{default} = (0::nat) \\
\textbf{instance} .. \\
\textbf{end} \\
\textbf{instantiation} \ \textbf{prod} :: (\textit{default}, \textit{default}) \textit{default} \\
\textbf{begin} \\
\textbf{definition} \textit{default} = (\textit{default}, \textit{default}) \\
\textbf{instance} .. \\
\textbf{end} \\
\textbf{definition} \textit{test } n u = \text{pigeonhole (nat-of-integer } n) (\lambda m. m - 1) \\
\textbf{definition} \textit{test’ } n u = \text{pigeonhole-slow (nat-of-integer } n) (\lambda m. m - 1) \\
\textbf{definition} \textit{test” } u = \text{pigeonhole } 8 (\textbf{List}.\textit{nth } [0, 1, 2, 3, 4, 5, 6, 3, 7, 8])
ML-val timeit (@{code test} 10)
ML-val timeit (@{code test'} 10)
ML-val timeit (@{code test} 20)
ML-val timeit (@{code test'} 20)
ML-val timeit (@{code test} 25)
ML-val timeit (@{code test'} 25)
ML-val timeit (@{code test} 500)
ML-val timeit (@{code test'})

end

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: \( n = m \times k \implies m < n \implies k < n \implies Suc 0 < m \)
  by (induct m) auto

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

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

lemma prime-eq: prime \((p::nat) = (1 < p \land (\forall m. \text{m dvd } p \implies 1 < m \implies m = p))\)
  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=Suc 0)
  apply simp
apply simp
done

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 = m * k) ∨ ¬ (∃ m. n = m * k)
  by (rule search)
}
hence (∃ m. n = m * k) ∨ ¬ (∃ m. n = m * k)
  by (rule search)
thus ?thesis
proof
  assume ∃ m. n = m * k
  then obtain k m where k: k < n and m: m < n and nmk: n = m * k
  by iprover
  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 ¬ (∃ m. n = m * k)
  hence A: ∀ m. n = m * k by iprover
  have ∀ m. m < n → 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-mn-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
    next
  next

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

\textbf{lemma} dvd-factorial: \( 0 < m \implies m \leq n \implies m \text{ dvd} \text{ fact} (n::nat) \)
\textbf{proof} (\text{induct} \ n \ \text{rule: nat-induct})
\hspace{1em} \text{case} \ 0
\then \text{ show} \ ?\text{case} \ \text{by} \ \text{simp} \)
next
\hspace{1em} \text{case} \ (Suc \ n)
\from \ (m \leq Suc \ n) \ \text{show} \ ?\text{case} 
\hspace{1em} \text{proof} (\text{rule le-SucE})
\hspace{2em} \text{assume} \ m \leq n
\hspace{2em} \text{with} \ (0 < m) \ \text{have} \ m \text{ dvd} \text{ fact} \ n \ \text{by} \ (\text{rule Suc})
\hspace{2em} \text{then have} \ m \text{ dvd} \text{ fact} \ n \ast Suc \ n \ \text{by} \ (\text{rule dvd-mult2})
\hspace{2em} \text{then show} \ ?\text{thesis} \ \text{by} \ (\text{simp add: mult.commute}) \)
next
\hspace{2em} \text{assume} \ m = Suc \ n
\hspace{2em} \text{then have} \ m \text{ dvd} \text{ fact} \ n \ast Suc \ n
\hspace{3em} \text{by} \ (\text{auto intro: dvdI simp: ac-simps})
\hspace{2em} \text{then show} \ ?\text{thesis} \ \text{by} \ (\text{simp add: mult.commute}) \)
qed
qed

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

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

\textbf{lemma} all-prime-simps:
\( \text{all-prime} \ [] \)
\( \text{all-prime} \ (p \ # \ ps) \leftarrow \text{prime} \ p \land \text{all-prime} \ ps \)
\by \ ((\text{simp-all add: all-prime-def}) \)

\textbf{lemma} all-prime-append:
\( \text{all-prime} \ (ps \ @ \ qs) \leftarrow \text{all-prime} \ ps \land \text{all-prime} \ qs \)
\by \ ((\text{simp add: all-prime-def ball-Un}) \)

\textbf{lemma} split-all-prime:
assumes all-prime ms and all-prime ns
shows \( \exists qs. \) all-prime qs \& (PROD m::nat:#multiset-of qs. m) =
\( \exists qs. \) ?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) =
(\( \forall \) n. \( \forall \) k. Suc 0 < m \& Suc 0 < k \& m < n \& k < n \& n = m * k) \& prime n
ultimately have ?thesis ..
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-nat-def [symmetric] by (induct ps rule: list-nenmpty-induct)
  (simp-all add: all-prime-simps msetprod-singleton msetprod-Un prime-gt-1-nat less-1-mult del: One-nat-def)

lemma factor-exists: Suc 0 < n \Longrightarrow (\exists 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 ?thesis ..
then show ?thesis ..
proof
  assume 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 \( \exists ps. \) 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: (\( \forall \) n. \( \forall \) m. \& all-prime ps1. m) = m
  by iprover
  from kn and k have \( \exists ps. \) all-prime ps \& \& (\( \forall \) m::nat:#multiset-of ps. m)
  = k by (rule 1)
  then obtain ps2 where all-prime ps2 and prod-ps2-k: (\( \forall \) m::nat:#multiset-of ps2. m) = k
  by iprover
  from all-prime ps1 \&\&\& (all-prime ps2)
  have \( \exists ps. \) all-prime ps \& \& (\( \forall \) m::nat:#multiset-of ps. m) =
    (\( \forall \) m::nat:#multiset-of ps1. m) * (\( \forall \) 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

have 1 < fact n + 1 by simp

then obtain p where 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
extract Euclid

The program extracted from the proof of Euclid’s theorem looks as follows.

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

The program corresponding to the proof of the factorization theorem is

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

instantiation nat :: default
begin

definition default = (0::nat)
instance ..
end

instantiation list :: (type) default
begin

definition default = []
instance ..
end

primrec iterate :: nat \Rightarrow \ ('a \Rightarrow 'a) \Rightarrow \ 'a \Rightarrow \ 'a \ list \ where
iterate 0 \ f \ x = []
| iterate (Suc \ n) \ f \ x = (\text{let} \ y = f \ x \text{ in} \ y \# \ iterate \ n \ f \ y)

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

lemma iterate 4 Euclid 0 = [2, 3, 7, 71] by eval
end
References


