

Examples for program extraction in Higher-Order Logic

Stefan Berghofer

May 23, 2024

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	18
5.2	Some examples	19
6	The pigeonhole principle	21
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:  $\bigwedge n::nat. m = n \vee m \neq n$ 
  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 $R: \bigwedge x::nat. (\bigwedge y. y < x \implies P y) \implies P x$
shows $P z$

proof (*rule R*)

show $\bigwedge y. y < z \implies P y$

proof (*induct z*)

case 0

then show *?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*

then show *?case* **by** (*rule Suc*)

qed

qed

qed

Bounded search for a natural number satisfying a decidable predicate.

lemma *search*:

assumes *dec*: $\bigwedge x::nat. P x \vee \neg P x$

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

proof (*induct y*)

case 0

show *?case* **by** *simp*

next

case (*Suc z*)

then show *?case*

proof

assume $\exists x < z. P x$

then obtain x **where** $le: x < z$ **and** $P: P x$ **by** *iprover*

from *le* **have** $x < Suc z$ **by** *simp*

with P **show** *?case* **by** *iprover*

next

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

from *dec* **show** *?case*

proof

assume $P: P z$

have $z < Suc z$ **by** *simp*

with P **show** *?thesis* **by** *iprover*

next

```

assume  $nP: \neg P z$ 
have  $\neg (\exists x < Suc z. P x)$ 
proof
  assume  $\exists x < Suc z. P x$ 
  then obtain  $x$  where  $le: x < Suc z$  and  $P: P x$  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
then show ?case by iprover
qed
qed
qed
end

```

2 Quotient and remainder

```

theory QuotRem
imports Util HOL-Library.Realizers
begin

```

Derivation of quotient and remainder using program extraction.

```

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

```

qed

extract *division*

The program extracted from the above proof looks as follows

```
division ≡
λx xa.
  nat-induct-P x (0, 0)
  (λa H. let (x, y) = H
        in case nat-eq-dec x xa of Left ⇒ (0, Suc y)
         | Right ⇒ (Suc x, y))
```

The corresponding correctness theorem is

$$a = \text{Suc } b * \text{snd } (\text{division } a \ b) + \text{fst } (\text{division } a \ b) \wedge \text{fst } (\text{division } a \ b) \leq b$$

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

end

3 Greatest common divisor

theory *Greatest-Common-Divisor*

imports *QuotRem*

begin

theorem *greatest-common-divisor*:

$$\bigwedge n::\text{nat}. \text{Suc } m < n \implies \\ \exists k \ n1 \ m1. k * n1 = n \wedge k * m1 = \text{Suc } m \wedge \\ (\forall l \ l1 \ l2. l * l1 = n \longrightarrow l * l2 = \text{Suc } m \longrightarrow l \leq k)$$

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

case (1 m n)

from *division* **obtain** r q **where** h1: n = Suc m * q + r **and** h2: r ≤ m

by *iprover*

show ?case

proof (*cases r*)

case 0

with h1 **have** Suc m * q = n **by** *simp*

moreover **have** Suc m * 1 = Suc m **by** *simp*

moreover **have** l * l1 = n \implies l * l2 = Suc m \implies l ≤ Suc m **for** l l1 l2

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

ultimately **show** ?thesis **by** *iprover*

next

case (Suc nat)

with h2 **have** h: nat < m **by** *simp*

moreover **from** h **have** Suc nat < Suc m **by** *simp*

ultimately **have** $\exists k \ m1 \ r1. k * m1 = \text{Suc } m \wedge k * r1 = \text{Suc } \text{nat} \wedge$

$(\forall l \ l1 \ l2. l * l1 = \text{Suc } m \longrightarrow l * l2 = \text{Suc } \text{nat} \longrightarrow l \leq k)$

```

    by (rule 1)
  then obtain  $k\ m1\ r1$  where  $h1': k * m1 = Suc\ m$ 
    and  $h2': k * r1 = Suc\ nat$ 
    and  $h3': \bigwedge l\ l1\ l2. l * l1 = Suc\ m \implies l * l2 = Suc\ nat \implies l \leq k$ 
    by iprover
  have  $mn: Suc\ m < n$  by (rule 1)
  from  $h1\ h1'\ h2'\ Suc$  have  $k * (m1 * q + r1) = n$ 
    by (simp add: add-mult-distrib2 mult.assoc [symmetric])
  moreover have  $l \leq k$  if  $ll1n: l * l1 = n$  and  $ll2m: l * l2 = Suc\ m$  for  $l\ l1\ l2$ 
  proof -
    have  $l * (l1 - l2 * q) = Suc\ nat$ 
    by (simp add: diff-mult-distrib2 h1 Suc [symmetric] mn ll1n ll2m [symmetric])
    with ll2m show  $l \leq k$  by (rule h3')
  qed
  ultimately show ?thesis using h1' by iprover
qed
qed

```

extract *greatest-common-divisor*

The extracted program for computing the greatest common divisor is

```

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

```

instantiation *nat* :: *default*
begin

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

instance ..

end

instantiation *prod* :: $(default, default)\ default$
begin

definition *default* = $(default, default)$

instance ..

end

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 *HOL-Library.Realizers*

begin

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

datatype $b = T \mid F$

primrec $is-path' :: ('a \Rightarrow 'a \Rightarrow b) \Rightarrow 'a \Rightarrow 'a\ list \Rightarrow 'a \Rightarrow bool$

where

$is-path' r x [] z \longleftrightarrow r x z = T$
 $| is-path' r x (y \# ys) z \longleftrightarrow r x y = T \wedge is-path' r y ys z$

definition $is-path :: (nat \Rightarrow nat \Rightarrow b) \Rightarrow (nat * nat\ list * nat) \Rightarrow nat \Rightarrow nat \Rightarrow nat \Rightarrow bool$

where $is-path r p i j k \longleftrightarrow$

$fst\ p = j \wedge snd\ (snd\ p) = k \wedge$

$list-all\ (\lambda x. x < i)\ (fst\ (snd\ p)) \wedge$

$is-path' r (fst\ p) (fst\ (snd\ p)) (snd\ (snd\ p))$

definition $conc :: 'a \times 'a\ list \times 'a \Rightarrow 'a \times 'a\ list \times 'a \Rightarrow 'a \times 'a\ list * 'a$

where $conc\ p\ q = (fst\ p, fst\ (snd\ p) @ fst\ q \# fst\ (snd\ q), snd\ (snd\ q))$

theorem $is-path'-snoc$ [*simp*]: $\bigwedge x. is-path' r x (ys @ [y]) z = (is-path' r x ys y \wedge r y z = T)$

by (*induct ys simp+*)

theorem $list-all-scoc$ [*simp*]: $list-all\ P\ (xs @ [x]) \longleftrightarrow P\ x \wedge list-all\ P\ xs$

by (*induct xs (simp+, iprover)*)

theorem $list-all-lemma$: $list-all\ P\ xs \Longrightarrow (\bigwedge x. P\ x \Longrightarrow Q\ x) \Longrightarrow list-all\ Q\ xs$

proof –

assume $PQ: \bigwedge x. P\ x \Longrightarrow Q\ x$

show $list-all\ P\ xs \Longrightarrow list-all\ Q\ xs$

proof (*induct xs*)

case *Nil*

```

    show ?case by simp
  next
    case (Cons y ys)
    then have 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:  $\bigwedge p. \text{is-path } r \ p \ i \ j \ k \implies \text{is-path } r \ p \ (\text{Suc } i) \ j \ k$ 
  unfolding 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:  $\bigwedge p. \text{is-path } r \ p \ 0 \ j \ k \implies r \ j \ k = T$ 
  unfolding is-path-def
  apply (simp cong add: conj-cong add: split-paired-all)
  apply (case-tac a)
  apply simp-all
  done

theorem is-path'-conc:  $\text{is-path}' \ r \ j \ xs \ i \implies \text{is-path}' \ r \ i \ ys \ k \implies$ 
 $\text{is-path}' \ r \ j \ (xs \ @ \ i \ \# \ ys) \ k$ 
  proof -
    assume pys: is-path' r i ys k
    show  $\bigwedge j. \text{is-path}' \ r \ j \ xs \ i \implies \text{is-path}' \ r \ j \ (xs \ @ \ i \ \# \ ys) \ k$ 
    proof (induct xs)
      case (Nil j)
      then have r j i = T by simp
      with pys show ?case by simp
    next
      case (Cons z zs j)
      then have 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:
 $\bigwedge p \ q. \text{is-path } r \ p \ i \ j \ i \implies \text{is-path } r \ q \ i \ i \ k \implies$ 
 $\text{is-path } r \ (\text{conc } p \ q) \ (\text{Suc } i) \ j \ k$ 
  apply (unfold is-path-def conc-def)
  apply (simp cong add: conj-cong add: split-paired-all)
  apply (erule conjE)+

```

```

apply (rule conjI)
apply (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:

```

 $\bigwedge p. \text{is-path } r \ p \ (\text{Suc } i) \ j \ k \implies \neg \text{is-path } r \ p \ i \ j \ k \implies$ 
 $(\exists q. \text{is-path } r \ q \ i \ j \ i) \wedge (\exists q'. \text{is-path } r \ q' \ i \ i \ k)$ 

```

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*

$\neg \text{list-all } (\lambda x. x < i) \text{ } xs$

show $(\exists ys. \text{list-all } (\lambda x. x < i) \text{ } ys \wedge \text{is-path}' \ r \ j \ ys \ i) \wedge$

$(\exists ys. \text{list-all } (\lambda x. x < i) \text{ } ys \wedge \text{is-path}' \ r \ i \ ys \ k)$

proof

have $\bigwedge j. \text{list-all } (\lambda x. x < \text{Suc } i) \text{ } xs \implies \text{is-path}' \ r \ j \ xs \ k \implies$

$\neg \text{list-all } (\lambda x. x < i) \text{ } xs \implies$

$\exists ys. \text{list-all } (\lambda x. x < i) \text{ } ys \wedge \text{is-path}' \ r \ j \ ys \ i$ (**is PROP ?ih xs**)

proof (induct *xs*)

case *Nil*

then show ?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

then show *list-all* ($\lambda x. x < i$) [] \wedge *is-path'* *r j* [] *i* **by** simp

qed

next

case *False*

have PROP ?ih *as* **by** (rule *Cons*)

then obtain *ys* **where** *ys*: *list-all* ($\lambda x. x < i$) *ys* \wedge *is-path'* *r a* *ys i*

proof

from *Cons* **show** *list-all* ($\lambda x. x < \text{Suc } i$) *as* **by** simp

from *Cons* **show** *is-path'* *r a* *as k* **by** simp

from *Cons* **and** *False* **show** $\neg \text{list-all } (\lambda x. x < i) \text{ } as$ **by** (simp)

qed

show ?thesis

proof

from *Cons False ys*
show $list\text{-}all (\lambda x. x < i) (a \# ys) \wedge is\text{-}path' r j (a \# ys) i$ **by** *simp*
qed
qed
qed
from *this asms* **show** $\exists ys. list\text{-}all (\lambda x. x < i) ys \wedge is\text{-}path' r j ys i$.
have $\bigwedge k. list\text{-}all (\lambda x. x < Suc\ i) xs \implies is\text{-}path' r j xs k \implies$
 $\neg list\text{-}all (\lambda x. x < i) xs \implies$
 $\exists ys. list\text{-}all (\lambda x. x < i) ys \wedge is\text{-}path' r i ys k$ (**is** *PROP ?ih xs*)
proof (*induct xs rule: rev-induct*)
case *Nil*
then show *?case* **by** *simp*
next
case (*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*
then show $list\text{-}all (\lambda x. x < i) [] \wedge is\text{-}path' r i [] k$ **by** *simp*
qed
next
case *False*
have *PROP ?ih as* **by** (*rule snoc*)
then obtain *ys* **where** $ys: list\text{-}all (\lambda x. x < i) ys \wedge is\text{-}path' r i ys a$
proof
from *snoc* **show** $list\text{-}all (\lambda x. x < Suc\ i) as$ **by** *simp*
from *snoc* **show** $is\text{-}path' r j as\ a$ **by** *simp*
from *snoc and False* **show** $\neg list\text{-}all (\lambda x. x < i) as$ **by** *simp*
qed
show *?thesis*
proof
from *snoc False ys*
show $list\text{-}all (\lambda x. x < i) (ys @ [a]) \wedge is\text{-}path' r i (ys @ [a]) k$
by *simp*
qed
qed
qed
from *this asms* **show** $\exists ys. list\text{-}all (\lambda x. x < i) ys \wedge is\text{-}path' r i ys k$.
qed
qed

theorem *lemma5'*:

$\bigwedge p. is\text{-}path\ r\ p\ (Suc\ i)\ j\ k \implies \neg is\text{-}path\ r\ p\ i\ j\ k \implies$
 $\neg (\forall q. \neg is\text{-}path\ r\ q\ i\ j\ i) \wedge \neg (\forall q'. \neg is\text{-}path\ r\ q'\ i\ i\ k)$
by (*iprover dest: lemma5*)

theorem *warshall*: $\bigwedge j\ k. \neg (\exists p. is\text{-}path\ r\ p\ i\ j\ k) \vee (\exists p. is\text{-}path\ r\ p\ i\ j\ k)$

```

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

```

```

    then show ?case ..
  qed
qed

```

```

extract warshall

```

The program extracted from the above proof looks as follows

```

warshall ≡
λx xa xb xc.
  nat-induct-P xa
  (λxa xb. case xa xb of T ⇒ Some (xa, [], xb) | F ⇒ None)
  (λx H2 xa xb.
    case H2 xa xb of
      None ⇒
        case H2 xa x of None ⇒ None
        | Some q ⇒
          case H2 x xb of None ⇒ None | Some qa ⇒ Some (conc q qa)
        | Some q ⇒ Some q)
  xb xc

```

The corresponding correctness theorem is

```

case warshall r i j k of None ⇒ ∀x. ¬ is-path r x i j k
| Some q ⇒ is-path r q i j k

```

```

ML-val @{code warshall}

```

```

end

```

5 Higman's lemma

```

theory Higman
imports Main
begin

```

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

```

datatype letter = A | B

```

```

inductive emb :: letter list ⇒ letter list ⇒ bool

```

```

where

```

```

  emb0 [Pure.intro]: emb [] bs
| emb1 [Pure.intro]: emb as bs ⇒ emb as (b # bs)
| emb2 [Pure.intro]: emb as bs ⇒ emb (a # as) (a # bs)

```

```

inductive L :: letter list ⇒ letter list list ⇒ 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]: $T\ a\ ws\ zs \implies T\ a\ (w\ \# \ ws)\ ((a\ \# \ w)\ \# \ zs)$
| *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]: $(\bigwedge w. bar\ (w\ \# \ ws)) \implies bar\ ws$

theorem *prop1*: $bar\ ([]\ \# \ ws)$

by *iprover*

theorem *lemma1*: $L\ as\ ws \implies L\ (a\ \# \ as)\ ws$

by (erule *L.induct*) *iprover*+

lemma *lemma2'*: $R\ a\ vs\ ws \implies L\ as\ vs \implies L\ (a\ \# \ as)\ ws$

supply [[*simproc del: defined-all*]]

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

lemma *lemma2*: $R\ a\ vs\ ws \implies good\ vs \implies good\ ws$

supply [[*simproc del: defined-all*]]

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':  $T\ a\ vs\ ws \implies L\ as\ vs \implies L\ (a\ \# \ as)\ ws$ 
supply [[simproc del: defined-all]]
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:  $T\ a\ ws\ zs \implies good\ ws \implies good\ zs$ 
supply [[simproc del: defined-all]]
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:  $R\ a\ ws\ zs \implies ws \neq [] \implies T\ a\ ws\ zs$ 
supply [[simproc del: defined-all]]
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-neg:  $a \neq b \implies c \neq a \implies c = b$  for  $a\ b\ c :: \text{letter}$ 
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:  $a = b \vee a \neq b$  for  $a\ b :: \text{letter}$ 
apply (case-tac a)
apply (case-tac b)
apply simp
apply simp
apply (case-tac b)
apply simp
apply simp
done

```

```

theorem prop2:
  assumes  $ab: a \neq b$  and  $bar: bar\ xs$ 
  shows  $\bigwedge ys\ zs. bar\ ys \implies T\ a\ xs\ zs \implies T\ b\ ys\ zs \implies bar\ zs$ 
  using bar
proof induct
  fix  $xs\ zs$ 
  assume  $T\ a\ xs\ zs$  and  $good\ xs$ 
  then have  $good\ zs$  by (rule lemma3)
  then show  $bar\ zs$  by (rule bar1)
next
  fix  $xs\ ys$ 
  assume  $I: \bigwedge w\ ys\ zs. bar\ ys \implies T\ a\ (w\ \# \ xs)\ zs \implies T\ b\ ys\ zs \implies bar\ zs$ 
  assume  $bar\ ys$ 
  then show  $\bigwedge zs. T\ a\ xs\ zs \implies T\ b\ ys\ zs \implies bar\ zs$ 
  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  $I': \bigwedge w\ zs. T\ a\ xs\ zs \implies T\ b\ (w\ \# \ ys)\ zs \implies bar\ zs$ 
  and  $ys: \bigwedge w. bar\ (w\ \# \ ys)$  and  $Ta: T\ a\ xs\ zs$  and  $Tb: T\ b\ ys\ zs$ 

```

```

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

```

```

theorem prop3:
  assumes bar: bar xs
  shows  $\bigwedge zs. xs \neq [] \implies R\ a\ xs\ zs \implies bar\ zs$ 
  using bar
proof induct
  fix xs zs
  assume R a xs zs and good xs
  then have good zs by (rule lemma2)
  then show bar zs by (rule bar1)
next
  fix xs zs
  assume I:  $\bigwedge w zs. w \# xs \neq [] \implies R\ a\ (w \# xs)\ zs \implies bar\ zs$ 
  and xsb:  $\bigwedge w. bar\ (w \# xs)$  and xsn:  $xs \neq []$  and R:  $R\ a\ xs\ zs$ 
  show bar zs
  proof (rule bar2)
    fix w
    show bar (w # zs)
    proof (induct w)
      case Nil
      show ?case by (rule prop1)
    next
    case (Cons c cs)
    from letter-eq-dec show ?case
  proof

```

```

    assume c = a
    then show ?thesis by (iprover intro: I [simplified] R)
  next
    from R xsn have T: T a xs zs by (rule lemma4)
    assume c ≠ a
    then show ?thesis by (iprover intro: prop2 Cons xsb xsn R T)
  qed
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]
    then show 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)
  then have 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
  then have 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)
  then have  $w = f (\text{length } ws)$  and is-prefix ws f by simp-all
  with good0 show ?case by (iprover dest: L-idx)
next
  case (good1 ws w)
  then show ?case by simp
qed

```

```

theorem bar-idx:
  assumes bar: bar ws
  shows is-prefix ws f  $\implies \exists i j. \text{emb } (f i) (f j) \wedge i < j$ 
  using bar
proof induct
  case (bar1 ws)
  then show ?case by (rule good-idx)
next
  case (bar2 ws)
  then have is-prefix (f (length ws) # ws) f by simp
  then show ?case by (rule bar2)
qed

```

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

```

theorem higman-idx:  $\exists (i::\text{nat}) j. \text{emb } (f i) (f j) \wedge i < j$ 
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  $\implies \exists vs. \text{is-prefix } vs f \wedge \text{good } vs$ 
  using bar
proof induct
  case bar1
  then show ?case by iprover
next
  case (bar2 ws)
  from bar2.prem1 have is-prefix (f (length ws) # ws) f by simp
  then show ?case by (iprover intro: bar2)
qed

```

```

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

```

end

5.1 Extracting the program

```

theory Higman-Extraction
imports Higman HOL-Library.Realizers HOL-Library.Open-State-Syntax
begin

```

```

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*:

$$\mathit{higman-idx} \equiv \lambda x. \mathit{bar-idx} \ x \ \mathit{higman}$$

Corresponding correctness theorem:

$$\mathit{emb} \ (f \ (\mathit{fst} \ (\mathit{higman-idx} \ f))) \ (f \ (\mathit{snd} \ (\mathit{higman-idx} \ f))) \wedge$$

$$\mathit{fst} \ (\mathit{higman-idx} \ f) < \mathit{snd} \ (\mathit{higman-idx} \ f)$$

Program extracted from the proof of *higman*:

$$\mathit{higman} \equiv$$

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

Program extracted from the proof of *prop1*:

$$\mathit{prop1} \equiv$$

$$\lambda x. \ \mathit{bar2} \ ([] \ \# \ x) \ (\lambda w. \ \mathit{bar1} \ (w \ \# \ [] \ \# \ x) \ (\mathit{good0} \ w \ ([] \ \# \ x) \ (\mathit{L0} \ [] \ x)))$$

Program extracted from the proof of *prop2*:

$$\mathit{prop2} \equiv$$

$$\lambda x \ x_a \ x_b \ x_c \ H.$$

$$\mathit{compat-barT.rec-split-barT}$$

$$(\lambda w s \ x_a \ x_b \ x_b a \ H \ H_a \ H_a a. \ \mathit{bar1} \ x_b a \ (\mathit{lemma3} \ x \ H_a \ x_a))$$

$$(\lambda w s \ x_b \ r \ x_b a \ x_b b \ H.$$

$$\mathit{compat-barT.rec-split-barT} \ (\lambda w s \ x \ x_b \ H \ H_a. \ \mathit{bar1} \ x_b \ (\mathit{lemma3} \ x_a \ H_a \ x))$$

$$(\lambda w s_a \ x_b \ r_a \ x_c \ H \ H_a.$$

$$\mathit{bar2} \ x_c$$

$$(\lambda w. \ \mathit{case} \ w \ \mathit{of} \ [] \Rightarrow \ \mathit{prop1} \ x_c$$

$$| \ a \ \# \ \mathit{list} \Rightarrow$$

$$\mathit{case} \ \mathit{letter-eq-dec} \ a \ x \ \mathit{of}$$

$$\mathit{Left} \Rightarrow$$

$$r \ \mathit{list} \ w_s a \ ((x \ \# \ \mathit{list}) \ \# \ x_c) \ (\mathit{bar2} \ w_s a \ x_b)$$

$$(\mathit{T1} \ w_s \ x_c \ \mathit{list} \ H) \ (\mathit{T2} \ x \ w_s a \ x_c \ \mathit{list} \ H_a)$$

$$| \ \mathit{Right} \Rightarrow$$

```

      ra list ((xa # list) # xc) (T2 xa ws xc list H)
      (T1 wsa xc list Ha)))
    H xbb)
  H xb xc

```

Program extracted from the proof of *prop3*:

```

prop3 ≡
λx xa H.
  compat-barT.rec-split-barT (λws xa xb H. bar1 xb (lemma2 x H xa))
  (λws xa r xb H.
    bar2 xb
    (rec-list (prop1 xb)
      (λa w Ha.
        case letter-eq-dec a x of
        Left ⇒ r w ((x # w) # xb) (R1 ws xb w H)
        | Right ⇒
          prop2 a x ws ((a # w) # xb) Ha (bar2 ws xa)
          (T0 x ws xb w H) (T2 a ws xb w (lemma4 x H))))))
  H xa

```

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* ((-) 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) = exec {
  - ← mk-word;
  mk-word-s n
}

definition g1 :: nat ⇒ letter list
where g1 s = fst (mk-word-s s (20000, 1))

definition g2 :: nat ⇒ letter list
where g2 s = fst (mk-word-s s (50000, 1))

fun f1 :: nat ⇒ letter list
where
  f1 0 = [A, A]
| f1 (Suc 0) = [B]
| f1 (Suc (Suc 0)) = [A, B]
| f1 - = []

fun f2 :: nat ⇒ letter list
where
  f2 0 = [A, A]
| f2 (Suc 0) = [B]
| f2 (Suc (Suc 0)) = [B, A]
| f2 - = []

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

```

end

6 The pigeonhole principle

theory *Pigeonhole*

imports *Util HOL-Library.Realizers HOL-Library.Code-Target-Numeral*

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.

theorem *pigeonhole*:

$\bigwedge f. (\bigwedge i. i \leq \text{Suc } n \implies f i \leq n) \implies \exists i j. i \leq \text{Suc } n \wedge j < i \wedge f i = f j$

proof (*induct n*)

case 0

then have $\text{Suc } 0 \leq \text{Suc } 0 \wedge 0 < \text{Suc } 0 \wedge f (\text{Suc } 0) = f 0$ **by** *simp*

then show *?case* **by** *iprover*

next

case (*Suc n*)

have *r*:

$k \leq \text{Suc } (\text{Suc } n) \implies$

$(\bigwedge i j. \text{Suc } k \leq i \implies i \leq \text{Suc } (\text{Suc } n) \implies j < i \implies f i \neq f j) \implies$

$(\exists i j. i \leq k \wedge j < i \wedge f i = f j)$ **for** *k*

proof (*induct k*)

case 0

let $?f = \lambda i. \text{if } f i = \text{Suc } n \text{ then } f (\text{Suc } (\text{Suc } n)) \text{ else } f i$

have $\neg (\exists i j. i \leq \text{Suc } n \wedge j < i \wedge ?f i = ?f j)$

proof

assume $\exists i j. i \leq \text{Suc } n \wedge j < i \wedge ?f i = ?f j$

then obtain *i j* **where** *i*: $i \leq \text{Suc } n$ **and** *j*: $j < i$ **and** *f*: $?f i = ?f j$

by *iprover*

from *j* **have** *i-nz*: $\text{Suc } 0 \leq i$ **by** *simp*

from *i* **have** *iSSn*: $i \leq \text{Suc } (\text{Suc } n)$ **by** *simp*

have *SOSSn*: $\text{Suc } 0 \leq \text{Suc } (\text{Suc } n)$ **by** *simp*

show *False*

proof *cases*

assume *fi*: $f i = \text{Suc } n$

show *False*

proof *cases*

assume *fj*: $f j = \text{Suc } n$

from *i-nz* **and** *iSSn* **and** *j* **have** $f i \neq f j$ **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** $j < \text{Suc } (\text{Suc } n)$ **by** *simp*

with *SOSSn* **and** *le-refl* **have** $f (\text{Suc } (\text{Suc } n)) \neq f j$

```

    by (rule 0)
  moreover assume  $f j \neq \text{Suc } n$ 
  with  $f i$  and  $f$  have  $f (\text{Suc } (\text{Suc } n)) = f j$  by simp
  ultimately show False ..
qed
next
assume  $f i: f i \neq \text{Suc } n$ 
show False
proof cases
  from  $i$  have  $i < \text{Suc } (\text{Suc } n)$  by simp
  with  $S0SSn$  and le-refl have  $f (\text{Suc } (\text{Suc } n)) \neq f i$ 
    by (rule 0)
  moreover assume  $f j = \text{Suc } n$ 
  with  $f i$  and  $f$  have  $f (\text{Suc } (\text{Suc } n)) = f i$  by simp
  ultimately show False ..
next
from  $i\text{-nz}$  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  $?f i \leq n$  if  $i \leq \text{Suc } n$  for  $i$ 
proof -
  from that have  $i: i < \text{Suc } (\text{Suc } n)$  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
  then have  $f i \leq \text{Suc } n$  by (rule Suc)
  ultimately show ?thesis
    by simp
qed
then have  $\exists i j. i \leq \text{Suc } n \wedge j < i \wedge ?f i = ?f j$ 
  by (rule Suc)
ultimately show ?case ..
next
case (Suc k)
from search [OF nat-eq-dec] show ?case
proof
  assume  $\exists j < \text{Suc } k. f (\text{Suc } k) = f j$ 
  then show ?case by (iprover intro: le-refl)
next
assume nex:  $\neg (\exists j < \text{Suc } k. f (\text{Suc } k) = f j)$ 
have  $\exists i j. i \leq k \wedge j < i \wedge f i = f j$ 
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 ?thesis
  proof
    assume  $f i = f j$ 
    then have  $f (\text{Suc } k) = f j$  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
  then show ?thesis using i and j by (rule Suc)
qed
qed
then show ?thesis by (iprover intro: le-SucI)
qed
qed
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:

```

theorem pigeonhole-slow:
   $\bigwedge f. (\bigwedge i. i \leq \text{Suc } n \implies f i \leq n) \implies \exists i j. i \leq \text{Suc } n \wedge j < i \wedge f i = f j$ 
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 (Suc n)
  from search [OF nat-eq-dec] show ?case
  proof
    assume  $\exists j < \text{Suc } (\text{Suc } n). f (\text{Suc } (\text{Suc } n)) = f j$ 
    then show ?case by (iprover intro: le-refl)
  next
    assume  $\neg (\exists j < \text{Suc } (\text{Suc } n). f (\text{Suc } (\text{Suc } n)) = f j)$ 
    then have nex:  $\forall j < \text{Suc } (\text{Suc } n). f (\text{Suc } (\text{Suc } n)) \neq f j$  by iprover
    let ?f =  $\lambda i. \text{if } f i = \text{Suc } n \text{ then } f (\text{Suc } (\text{Suc } n)) \text{ else } f i$ 
    have  $\bigwedge i. i \leq \text{Suc } n \implies ?f i \leq n$ 
    proof -
      fix i assume  $i \leq \text{Suc } n$ 
      show ?thesis i
      proof (cases f i = Suc n)

```

```

    case True
    from i and nex have f (Suc (Suc n)) ≠ f i by simp
    with True have f (Suc (Suc n)) ≠ Suc n by simp
    moreover from Suc have f (Suc (Suc n)) ≤ Suc n by simp
    ultimately have f (Suc (Suc n)) ≤ n by simp
    with True show ?thesis by simp
  next
    case False
    from Suc and i have f i ≤ Suc n by simp
    with False show ?thesis by simp
  qed
qed
then have ∃ i j. i ≤ Suc n ∧ j < i ∧ ?f i = ?f j by (rule Suc)
then obtain i j where i: i ≤ Suc n and ji: j < i and f: ?f i = ?f j
  by iprover
have f i = f j
proof (cases f i = Suc n)
  case True
  show ?thesis
  proof (cases f j = Suc n)
    assume f j = Suc n
    with True show ?thesis by simp
  next
    assume f j ≠ Suc n
    moreover from i ji nex have f (Suc (Suc n)) ≠ f j by simp
    ultimately show ?thesis using True f by simp
  qed
next
  case False
  show ?thesis
  proof (cases f j = Suc n)
    assume f j = Suc n
    moreover from i nex have f (Suc (Suc n)) ≠ f i by simp
    ultimately show ?thesis using False f by simp
  next
    assume f j ≠ Suc n
    with False f show ?thesis by simp
  qed
qed
moreover from i have i ≤ Suc (Suc n) by simp
ultimately show ?thesis using ji by iprover
qed

```

extract *pigeonhole pigeonhole-slow*

The programs extracted from the above proofs look as follows:

```

pigeonhole ≡
λx. nat-induct-P x (λx. (Suc 0, 0))

```


$$\begin{aligned}
& (\lambda x \ H2 \ xa. \\
& \quad nat-induct-P \ (Suc \ (Suc \ x)) \ default \\
& \quad (\lambda x \ H2. \\
& \quad \quad case \ search \ (Suc \ x) \ (\lambda xb. \ nat-eq-dec \ (xa \ (Suc \ x)) \ (xa \ xb)) \ of \\
& \quad \quad \quad None \Rightarrow \ let \ (x, \ y) = H2 \ in \ (x, \ y) \ | \ Some \ p \Rightarrow \ (Suc \ x, \ p))
\end{aligned}$$

$$\begin{aligned}
pigeonhole-slow & \equiv \\
\lambda x. \ nat-induct-P \ x \ (\lambda x. \ (Suc \ 0, \ 0)) \\
& (\lambda x \ H2 \ xa. \\
& \quad case \ search \ (Suc \ (Suc \ x)) \\
& \quad \quad (\lambda xb. \ nat-eq-dec \ (xa \ (Suc \ (Suc \ x))) \ (xa \ xb)) \ of \\
& \quad \quad \quad None \Rightarrow \\
& \quad \quad \quad \quad let \ (x, \ y) = \\
& \quad \quad \quad \quad \quad H2 \ (\lambda i. \ if \ xa \ i = Suc \ x \ then \ xa \ (Suc \ (Suc \ x)) \ else \ xa \ i) \\
& \quad \quad \quad \quad in \ (x, \ y) \\
& \quad \quad \quad \quad | \ Some \ p \Rightarrow \ (Suc \ (Suc \ x), \ p)
\end{aligned}$$

The program for searching for an element in an array is

$$\begin{aligned}
search & \equiv \\
\lambda x \ H. \ nat-induct-P \ x \ None \\
& (\lambda y \ Ha. \\
& \quad case \ Ha \ of \ None \Rightarrow \ case \ H \ y \ of \ Left \Rightarrow \ Some \ y \ | \ Right \Rightarrow \ None \\
& \quad \quad | \ Some \ p \Rightarrow \ Some \ p)
\end{aligned}$$

The correctness statement for *pigeonhole* is

$$\begin{aligned}
& (\bigwedge i. \ i \leq \ Suc \ n \Longrightarrow \ f \ i \leq \ n) \Longrightarrow \\
& \quad fst \ (pigeonhole \ n \ f) \leq \ Suc \ n \wedge \\
& \quad snd \ (pigeonhole \ n \ f) < \ fst \ (pigeonhole \ n \ f) \wedge \\
& \quad f \ (fst \ (pigeonhole \ n \ f)) = f \ (snd \ (pigeonhole \ n \ f))
\end{aligned}$$

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

```

instantiation nat :: default
begin

definition default = (0::nat)

instance ..

end

instantiation prod :: (default, default) default
begin

definition default = (default, default)

instance ..

```

end

definition $test\ n\ u = pigeonhole\ (nat-of-integer\ n)\ (\lambda m. m - 1)$

definition $test'\ n\ u = pigeonhole-slow\ (nat-of-integer\ n)\ (\lambda m. m - 1)$

definition $test''\ u = pigeonhole\ 8\ (List.nth\ [0, 1, 2, 3, 4, 5, 6, 3, 7, 8])$

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

HOL-Computational-Algebra.Primes

Util

HOL-Library.Code-Target-Numeral

HOL-Library.Realizers

begin

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

lemma *factor-greater-one1*: $n = m * k \implies m < n \implies k < n \implies Suc\ 0 < m$
by (*induct m*) *auto*

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

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

lemma *prime-eq*: $prime\ (p::nat) \iff 1 < p \wedge (\forall m. m\ dvd\ p \longrightarrow 1 < m \longrightarrow m = p)$

apply (*simp add: prime-nat-iff*)

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)  $\longleftrightarrow 1 < p \wedge (\forall m k. p = m * k \longrightarrow 1 < m \longrightarrow m = p)$ 
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 ( $\exists m k. \text{Suc } 0 < m \wedge \text{Suc } 0 < k \wedge m < n \wedge k < n \wedge n = m * k$ )  $\vee$  prime n
proof -
  from nat-eq-dec have ( $\exists m < n. n = m * k$ )  $\vee$   $\neg$  ( $\exists m < n. n = m * k$ ) for k
    by (rule search)
  then have ( $\exists k < n. \exists m < n. n = m * k$ )  $\vee$   $\neg$  ( $\exists k < n. \exists m < n. n = m * k$ )
    by (rule search)
  then show ?thesis
  proof
    assume  $\exists k < n. \exists m < n. n = m * k$ 
    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  $\neg$  ( $\exists k < n. \exists m < n. n = m * k$ )
    then have A:  $\forall k < n. \forall m < n. n \neq m * k$  by iprover
    have  $\forall m k. n = m * k \longrightarrow \text{Suc } 0 < m \longrightarrow m = n$ 
    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
  moreover note n
  moreover from False n nmk k have  $Suc\ 0 < k$  by auto
  moreover from nmk have  $k * m = n$  by (simp only: ac-simps)
  ultimately have mn:  $m < n$  by (rule prod-mn-less-k)
  with kn A nmk show ?thesis by iprover
qed
qed
with n have prime n
  by (simp only: prime-eq' One-nat-def simp-thms)
  then show ?thesis ..
qed
qed

```

```

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

```

```

lemma dvd-prod [iff]:  $n\ dvd\ (\prod m::nat \in\ \# mset\ (n\ \#\ ns). m)$ 
  by (simp add: prod-mset-Un)

```

```

definition all-prime :: nat list  $\Rightarrow$  bool
  where all-prime ps  $\longleftrightarrow (\forall p \in\ set\ ps. prime\ p)$ 

```

```

lemma all-prime-simps:
  all-prime []
  all-prime (p # ps)  $\longleftrightarrow prime\ p \wedge all-prime\ ps$ 
  by (simp-all add: all-prime-def)

```

```

lemma all-prime-append: all-prime (ps @ qs)  $\longleftrightarrow all-prime\ ps \wedge all-prime\ qs$ 
  by (simp add: all-prime-def ball-Un)

```

lemma *split-all-prime*:
assumes *all-prime ms and all-prime ns*
shows $\exists qs. \text{all-prime } qs \wedge$
 $(\prod m::nat \in\# \text{mset } qs. m) = (\prod m::nat \in\# \text{mset } ms. m) * (\prod m::nat \in\# \text{mset } ns. m)$
(is $\exists qs. ?P \text{ } qs \wedge ?Q \text{ } qs)$
proof –
from *assms* **have** *all-prime (ms @ ns)*
by (*simp add: all-prime-append*)
moreover
have $(\prod m::nat \in\# \text{mset } (ms @ ns). m) = (\prod m::nat \in\# \text{mset } ms. m) * (\prod m::nat \in\# \text{mset } ns. m)$
using *assms* **by** (*simp add: prod-mset-Un*)
ultimately have $?P (ms @ ns) \wedge ?Q (ms @ ns) ..$
then show *?thesis ..*
qed

lemma *all-prime-empty-g-one*:
assumes *all-prime ps and ps ≠ []*
shows $Suc\ 0 < (\prod m::nat \in\# \text{mset } ps. m)$
using $\langle ps \neq [] \rangle$ *all-prime ps*
unfolding *One-nat-def [symmetric]*
by (*induct ps rule: list-nonempty-induct*)
(simp-all add: all-prime-simps prod-mset-Un prime-gt-1-nat less-1-mult del: One-nat-def)

lemma *factor-exists*: $Suc\ 0 < n \implies (\exists ps. \text{all-prime } ps \wedge (\prod m::nat \in\# \text{mset } ps. m) = n)$
proof (*induct n rule: nat-wf-ind*)
case $(1\ n)$
from $\langle Suc\ 0 < n \rangle$
have $(\exists m\ k. Suc\ 0 < m \wedge Suc\ 0 < k \wedge m < n \wedge k < n \wedge n = m * k) \vee \text{prime } n$
by (*rule not-prime-ex-mk*)
then show *?case*
proof
assume $\exists m\ k. Suc\ 0 < m \wedge Suc\ 0 < k \wedge m < n \wedge k < n \wedge n = m * k$
then obtain $m\ k$ **where** $m: Suc\ 0 < m$ **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** $\exists ps. \text{all-prime } ps \wedge (\prod m::nat \in\# \text{mset } ps. m) = m$
by (*rule 1*)
then obtain $ps1$ **where** *all-prime ps1* **and** $prod-ps1-m: (\prod m::nat \in\# \text{mset } ps1. m) = m$
by *iprover*
from kn **and** k **have** $\exists ps. \text{all-prime } ps \wedge (\prod m::nat \in\# \text{mset } ps. m) = k$
by (*rule 1*)
then obtain $ps2$ **where** *all-prime ps2* **and** $prod-ps2-k: (\prod m::nat \in\# \text{mset } ps2. m) = k$

```

ps2. m) = k
  by iprover
  from ⟨all-prime ps1⟩ ⟨all-prime ps2⟩
  have ∃ ps. all-prime ps ∧ (∏ m::nat ∈# mset ps. m) =
    (∏ m::nat ∈# mset ps1. m) * (∏ m::nat ∈# mset 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 (∏ m::nat ∈# mset [n]. m) = n by (simp)
  ultimately have all-prime [n] ∧ (∏ m::nat ∈# mset [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 = (∏ m::nat ∈# mset
ps. m)
  using factor-exists by simp iprover
  with N have ps ≠ []
  by (auto simp add: all-prime-nempty-g-one)
  then obtain p qs where ps: 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

```

```

extract Euclid

```

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

```

Euclid ≡ λx. prime-factor-exists (fact x + 1)

```

The program corresponding to the proof of the factorization theorem is

```

factor-exists ≡
λx. nat-wf-ind-P x
  (λx H2.
    case not-prime-ex-mk x of None ⇒ [x]
    | Some p ⇒ let (x, y) = p 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 ⇒ ('a ⇒ 'a) ⇒ 'a ⇒ 'a list

```

```

where

```

```

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

```

```

lemma factor-exists 1007 = [53, 19] by eval

```

```

lemma factor-exists 567 = [7, 3, 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

- [1] U. Berger, H. Schwichtenberg, and M. Seisenberger. The Warshall algorithm and Dickson's lemma: Two examples of realistic program extraction. *Journal of Automated Reasoning*, 26:205–221, 2001.
- [2] T. Coquand and D. Fridlender. A proof of Higman's lemma by structural induction. Technical report, Chalmers University, November 1993.
- [3] A. Nogin. Writing constructive proofs yielding efficient extracted programs. In D. Galmiche, editor, *Proceedings of the Workshop on Type-Theoretic Languages: Proof Search and Semantics*, volume 37 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2000.
- [4] M. Wenzel and F. Wiedijk. A comparison of the mathematical proof languages Mizar and Isar. *Journal of Automated Reasoning*, 29(3-4):389–411, 2002.