Examples for program extraction in Higher-Order Logic

Stefan Berghofer

November 22, 2007

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 18

6 The pigeonhole principle **21**

7 Euclid's theorem **27**

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*: $\bigwedge x::nat.\ (\bigwedge y.\ y < x \Longrightarrow P\ y) \Longrightarrow P\ x$
  shows $P\ z$
proof (*rule R*)
  show $\bigwedge y.\ y < z \Longrightarrow P\ y$
  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
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*)
  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** $x < Suc\ z$ **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 ¬ (∃ x<Suc z. P x)
      proof
        assume ∃ 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 ∃ x<z. P x by iprover
        with nex show False ..
      qed
      thus ?case by iprover
    qed
  qed
qed

end
```

2 Quotient and remainder

```
theory QuotRem imports Util begin
```

Derivation of quotient and remainder using program extraction.

```
theorem division: ∃ r q. a = Suc b ∗ q + r ∧ r ≤ b
proof (induct a)
  case 0
  have 0 = Suc b ∗ 0 + 0 ∧ 0 ≤ b by simp
  thus ?case by iprover
next
  case (Suc a)
  then obtain r q where I: a = Suc b ∗ q + r and r ≤ b by iprover
  from nat-eq-dec show ?case
  proof
    assume r = b
    with I have Suc a = Suc b ∗ (Suc q) + 0 ∧ 0 ≤ b by simp
    thus ?case by iprover
  next
    assume r ≠ b
    with ‹r ≤ b› have r < b by (simp add: order-less-le)
    with I have Suc a = Suc b ∗ q + (Suc r) ∧ (Suc r) ≤ b by simp
    thus ?case by iprover
  qed
qed
```

extract *division*

The program extracted from the above proof looks as follows

$division \equiv$
$\lambda x\ xa.$
  $nat\text{-}rec\ (0,\ 0)$
  $(\lambda a\ H.\ let\ (x,\ y) = H$
    $in\ case\ nat\text{-}eq\text{-}dec\ x\ xa\ of\ Left \Rightarrow (0,\ Suc\ y)$
      $\mid Right \Rightarrow (Suc\ x,\ y))$
  x

The corresponding correctness theorem is

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

code-module *Div*
contains
  $test = division\ 9\ 2$

export-code *division* **in** *SML*

end

3 Greatest common divisor

theory *Greatest-Common-Divisor*
imports *QuotRem*
begin

theorem *greatest-common-divisor*:
  $\bigwedge n{::}nat.\ Suc\ m < n \Longrightarrow \exists\, k\ n1\ m1.\ k * n1 = n \wedge k * m1 = Suc\ m \wedge$
    $(\forall\, l\ l1\ l2.\ l * l1 = n \longrightarrow l * l2 = Suc\ m \longrightarrow l \leq k)$
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*: $r \leq m$
    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 {
      fix *l2* **have** $\bigwedge l\ l1.\ l * l1 = n \Longrightarrow l * l2 = Suc\ m \Longrightarrow l \leq Suc\ m$
        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 ∃ k m1 r1. k * m1 = Suc m ∧ k * r1 = Suc nat ∧
      (∀ l l1 l2. l * l1 = Suc m ⟶ l * l2 = Suc nat ⟶ l ≤ k)
      by (rule 1)
    then obtain k m1 r1 where
      h1': k * m1 = Suc m
      and h2': k * r1 = Suc nat
      and h3': ⋀l l1 l2. l * l1 = Suc m ⟹ l * l2 = Suc nat ⟹ l ≤ 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 nat-mult-assoc [symmetric])
    moreover have ⋀l l1 l2. l * l1 = n ⟹ l * l2 = Suc m ⟹ l ≤ k
    proof −
      fix l l1 l2
      assume ll1n: l * l1 = n
      assume ll2m: l * l2 = Suc m
      moreover have l * (l1 − l2 * q) = Suc nat
      by (simp add: diff-mult-distrib2 h1 Suc [symmetric] mn ll1n ll2m [symmetric])
      ultimately show l ≤ 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 ≡
λx. nat-wf-ind-P x
      (λx H2 xa.
          let (xa, y) = division xa x
          in case xa of 0 ⇒ (Suc x, y, 1)
             | Suc nat ⇒
                 let (x, ya) = H2 nat (Suc x); (xa, ya) = ya
                 in (x, xa * y + ya, xa))
```

```
consts-code
  arbitrary ((error arbitrary))

code-module GCD
contains
  test = greatest-common-divisor 7 12

ML GCD.test

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 $b = T \mid F$

consts
  $\mathit{is\text{-}path'} :: ('a \Rightarrow 'a \Rightarrow b) \Rightarrow 'a \Rightarrow 'a\ \mathit{list} \Rightarrow 'a \Rightarrow \mathit{bool}$

primrec
  $\mathit{is\text{-}path'}\ r\ x\ []\ z = (r\ x\ z = T)$
  $\mathit{is\text{-}path'}\ r\ x\ (y\ \#\ ys)\ z = (r\ x\ y = T \wedge \mathit{is\text{-}path'}\ r\ y\ ys\ z)$

constdefs
  $\mathit{is\text{-}path} :: (\mathit{nat} \Rightarrow \mathit{nat} \Rightarrow b) \Rightarrow (\mathit{nat} * \mathit{nat}\ \mathit{list} * \mathit{nat}) \Rightarrow$
    $\mathit{nat} \Rightarrow \mathit{nat} \Rightarrow \mathit{nat} \Rightarrow \mathit{bool}$
  $\mathit{is\text{-}path}\ r\ p\ i\ j\ k == \mathit{fst}\ p = j \wedge \mathit{snd}\ (\mathit{snd}\ p) = k \wedge$
    $\mathit{list\text{-}all}\ (\lambda x.\ x < i)\ (\mathit{fst}\ (\mathit{snd}\ p)) \wedge$
    $\mathit{is\text{-}path'}\ r\ (\mathit{fst}\ p)\ (\mathit{fst}\ (\mathit{snd}\ p))\ (\mathit{snd}\ (\mathit{snd}\ p))$

  $\mathit{conc} :: ('a * 'a\ \mathit{list} * 'a) \Rightarrow ('a * 'a\ \mathit{list} * 'a) \Rightarrow ('a * 'a\ \mathit{list} * 'a)$
  $\mathit{conc}\ p\ q == (\mathit{fst}\ p,\ \mathit{fst}\ (\mathit{snd}\ p)\ @\ \mathit{fst}\ q\ \#\ \mathit{fst}\ (\mathit{snd}\ q),\ \mathit{snd}\ (\mathit{snd}\ q))$

theorem *is-path′-snoc* [*simp*]:
  $\bigwedge x.\ \mathit{is\text{-}path'}\ r\ x\ (ys\ @\ [y])\ z = (\mathit{is\text{-}path'}\ r\ x\ ys\ y \wedge r\ y\ z = T)$
  by (*induct ys*) *simp+*

theorem *list-all-scoc* [*simp*]: $\mathit{list\text{-}all}\ P\ (xs\ @\ [x]) = (P\ x \wedge \mathit{list\text{-}all}\ P\ xs)$
  by (*induct xs*, *simp+*, *iprover*)

theorem *list-all-lemma*:
  $\mathit{list\text{-}all}\ P\ xs \Longrightarrow (\bigwedge x.\ P\ x \Longrightarrow Q\ x) \Longrightarrow \mathit{list\text{-}all}\ Q\ xs$
proof −
  assume *PQ*: $\bigwedge x.\ P\ x \Longrightarrow Q\ x$
  show $\mathit{list\text{-}all}\ P\ xs \Longrightarrow \mathit{list\text{-}all}\ Q\ xs$
  proof (*induct xs*)
    case *Nil*
    show *?case* **by** *simp*
  next
    case (*Cons y ys*)
    hence *Py*: *P y* **by** *simp*
    from *Cons* **have** *Pys*: *list-all P ys* **by** *simp*
    show *?case*
      by *simp* (*rule conjI PQ Py Cons Pys*)+
  qed

```
qed

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

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

theorem is-path'-conc: is-path' r j xs i ⟹ is-path' r i ys k ⟹
  is-path' r j (xs @ i # ys) k
proof −
  assume pys: is-path' r i ys k
  show ⋀j. is-path' r j xs i ⟹ is-path' r j (xs @ i # ys) k
  proof (induct xs)
    case (Nil j)
    hence r j i = T by simp
    with pys show ?case by simp
  next
    case (Cons z zs j)
    hence jzr: r j z = T by simp
    from Cons have pzs: is-path' r z zs i by simp
    show ?case
      by simp (rule conjI jzr Cons pzs)+
  qed
qed

theorem lemma3:
  ⋀p q. is-path r p i j i ⟹ is-path r q i i k ⟹
   is-path r (conc p q) (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:
  ⋀p. is-path r p (Suc i) j k ⟹ ~ is-path r p i j k ⟹
   (∃ q. is-path r q i j i) ∧ (∃ q'. 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 (λx. x < Suc i) xs
    is-path' r j xs k
    ¬ list-all (λx. x < i) xs
  show (∃ ys. list-all (λx. x < i) ys ∧ is-path' r j ys i) ∧
    (∃ ys. list-all (λx. x < i) ys ∧ is-path' r i ys k)
  proof
    show ⋀j. list-all (λx. x < Suc i) xs ⟹ is-path' r j xs k ⟹
      ¬ list-all (λx. x < i) xs ⟹
    ∃ ys. list-all (λx. x < i) ys ∧ 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 (λx. x < i) [] ∧ 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 (λx. x < i) ys ∧ is-path' r a ys i
        proof
          from Cons show list-all (λx. x < Suc i) as by simp
          from Cons show is-path' r a as k by simp
          from Cons and False show ¬ list-all (λx. x < i) as by (simp)
        qed
        show ?thesis
        proof
          from Cons False ys
          show list-all (λx. x<i) (a#ys) ∧ is-path' r j (a#ys) i by simp
        qed
      qed
    qed
    show ⋀k. list-all (λx. x < Suc i) xs ⟹ is-path' r j xs k ⟹
      ¬ list-all (λx. x < i) xs ⟹
      ∃ ys. list-all (λx. x < i) ys ∧ is-path' r i ys k (is PROP ?ih xs)
```

```
    proof (induct xs rule: rev-induct)
      case Nil
      thus ?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
          thus list-all (λx. x < i) [] ∧ is-path' r i [] k by simp
        qed
      next
        case False
        have PROP ?ih as by (rule snoc)
        then obtain ys where ys: list-all (λx. x < i) ys ∧ is-path' r i ys a
        proof
          from snoc show list-all (λx. x < Suc i) as by simp
          from snoc show is-path' r j as a by simp
          from snoc and False show ¬ list-all (λx. x < i) as by simp
        qed
        show ?thesis
        proof
          from snoc False ys
          show list-all (λx. x < i) (ys @ [a]) ∧ is-path' r i (ys @ [a]) k
            by simp
        qed
      qed
    qed
  qed (rule asms)+
qed

theorem lemma5':
  ⋀p. is-path r p (Suc i) j k ⟹ ¬ is-path r p i j k ⟹
   ¬ (∀q. ¬ is-path r q i j i) ∧ ¬ (∀q'. ¬ is-path r q' i i k)
  by (iprover dest: lemma5)

theorem warshall:
  ⋀j k. ¬ (∃p. is-path r p i j k) ∨ (∃p. is-path r p i j k)
proof (induct i)
  case (0 j k)
  show ?case
  proof (cases r j k)
    assume r j k = T
    hence is-path r (j, [], k) 0 j k
      by (simp add: is-path-def)
    hence ∃p. is-path r p 0 j k ..
    thus ?thesis ..
```

```
  next
    assume r j k = F
    hence r j k ~= T by simp
    hence ¬ (∃p. is-path r p 0 j k)
      by (iprover dest: lemma2)
    thus ?thesis ..
  qed
next
  case (Suc i j k)
  thus ?case
  proof
    assume h1: ¬ (∃p. is-path r p i j k)
    from Suc show ?case
    proof
      assume ¬ (∃p. is-path r p i j i)
      with h1 have ¬ (∃p. is-path r p (Suc i) j k)
        by (iprover dest: lemma5')
      thus ?case ..
    next
      assume ∃p. 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 ¬ (∃p. is-path r p i i k)
        with h1 have ¬ (∃p. is-path r p (Suc i) j k)
          by (iprover dest: lemma5')
        thus ?case ..
      next
        assume ∃q. is-path r q i i k
        then obtain q where is-path r q i i k ..
        with h2 have is-path r (conc p q) (Suc i) j k
          by (rule lemma3)
        hence ∃pq. is-path r pq (Suc i) j k ..
        thus ?case ..
      qed
    qed
  next
    assume ∃p. is-path r p i j k
    hence ∃p. is-path r p (Suc i) j k
      by (iprover intro: lemma1)
    thus ?case ..
  qed
qed

extract warshall
```

The program extracted from the above proof looks as follows

$warshall \equiv$
$\lambda x\ xa\ xb\ xc.$

```
nat-rec (λxa xb. case x 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)
 xa 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
```

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 ⟹ L v (w # ws)
 | L1 [Pure.intro]: L v ws ⟹ L v (w # ws)

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

inductive R :: letter ⇒ letter list list ⇒ letter list list ⇒ bool
  for a :: letter
where
```

$R0$ [*Pure.intro*]: $R\ a\ []\ []$
| $R1$ [*Pure.intro*]: $R\ a\ vs\ ws \Longrightarrow R\ a\ (w\ \#\ vs)\ ((a\ \#\ w)\ \#\ ws)$

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

inductive *bar* :: *letter list list* $\Rightarrow$ *bool*
where
bar1 [*Pure.intro*]: *good ws* $\Longrightarrow$ *bar ws*
| *bar2* [*Pure.intro*]: $(\bigwedge w.\ bar\ (w\ \#\ ws)) \Longrightarrow bar\ ws$

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

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

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

lemma *lemma2*: $R\ a\ vs\ ws \Longrightarrow good\ vs \Longrightarrow good\ ws$
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'*: $T\ a\ vs\ ws \Longrightarrow L\ as\ vs \Longrightarrow L\ (a\ \#\ as)\ ws$
apply (*induct set*: *T*)
apply (*erule L.cases*)
apply *simp-all*
apply (*rule L0*)
apply (*erule emb2*)
apply (*rule L1*)

```
  apply (erule lemma1)
  apply (erule L.cases)
  apply simp-all
  apply iprover+
  done

lemma lemma3: T a ws zs ⟹ good ws ⟹ 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 ⟹ ws ≠ [] ⟹ 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
  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 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
        thus ?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)
          thus ?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)
          thus ?thesis by (simp add: Cons cb)
        qed
      qed
```

```
    qed
  qed
qed

theorem prop3:
  assumes bar: bar xs
  shows ⋀zs. xs ≠ [] ⟹ R a xs zs ⟹ 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: ⋀w zs. w # xs ≠ [] ⟹ R a (w # xs) zs ⟹ bar zs
  and xsb: ⋀w. bar (w # xs) and xsn: xs ≠ [] 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
        thus ?thesis by (iprover intro: I [simplified] R)
      next
        from R xsn have T: T a xs zs by (rule lemma4)
        assume c ≠ a
        thus ?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]
    thus bar [c # cs] by (rule prop3) (simp, iprover)
  qed
qed
```

consts
  is-prefix :: $'a$ *list* $\Rightarrow$ (*nat* $\Rightarrow$ $'a$) $\Rightarrow$ *bool*

primrec
  is-prefix [] *f* = *True*
  is-prefix (*x* # *xs*) *f* = (*x* = *f* (*length xs*) $\wedge$ *is-prefix xs f*)

theorem *L-idx*:
  assumes *L*: *L w ws*
  shows *is-prefix ws f* $\Longrightarrow$ $\exists$ *i*. *emb* (*f i*) *w* $\wedge$ *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* $\Longrightarrow$ $\exists$ *i j*. *emb* (*f i*) (*f j*) $\wedge$ *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
  case (*good1 ws w*)
  thus *?case* **by** *simp*
qed

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

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

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

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

5.1 Extracting the program

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

extract *higman-idx*

Program extracted from the proof of *higman-idx*:

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

Corresponding correctness theorem:

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

Program extracted from the proof of *higman*:

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

Program extracted from the proof of *prop1*:

```
prop1 ≡
λx. bar2 ([] # x) (λw. bar1 (w # [] # x) (good0 w ([] # x) (L0 [] x)))
```

Program extracted from the proof of *prop2*:

```
prop2 ≡
λx xa xb xc H.
   barT-rec (λws xa xb xc H Ha Hb. bar1 xc (lemma3 x Ha xa))
    (λws xb r xc xd H.
        barT-rec (λws x xb H Ha. bar1 xb (lemma3 xa Ha x))
         (λwsa xb ra xc H Ha.
             bar2 xc
              (list-case (prop1 xc)
                (λa list.
                    case letter-eq-dec a x of
                    Left ⇒
                      r list wsa ((x # list) # xc) (bar2 wsa xb)
                       (T1 ws xc list H) (T2 x wsa xc list Ha)
                    | Right ⇒
                        ra list ((xa # list) # xc) (T2 xa ws xc list H)
                         (T1 wsa xc list Ha))))
         H xd)
    H xb xc
```

Program extracted from the proof of *prop3*:

```
prop3 ≡
λx xa H.
   barT-rec (λws xa xb H. bar1 xb (lemma2 x H xa))
    (λws xa r xb H.
        bar2 xb
         (list-rec (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

consts-code
  arbitrary :: *LT* (({∗ *L0* [] [] ∗}))
  arbitrary :: *TT* (({∗ *T0 A* [] [] [] *R0* ∗}))

code-module *Higman*
contains
  higman = *higman-idx*

ML ⟪

```
local open Higman in

val a = 16807.0;
val m = 2147483647.0;

fun nextRand seed =
  let val t = a*seed
  in  t − m * real (Real.floor(t/m)) end;

fun mk-word seed l =
  let
    val r = nextRand seed;
    val i = Real.round (r / m * 10.0);
  in if i > 7 andalso l > 2 then (r, []) else
    apsnd (cons (if i mod 2 = 0 then A else B)) (mk-word r (l+1))
  end;

fun f s zero = mk-word s 0
  | f s (Suc n) = f (fst (mk-word s 0)) n;

val g1 = snd o (f 20000.0);

val g2 = snd o (f 50000.0);

fun f1 zero = [A,A]
  | f1 (Suc zero) = [B]
  | f1 (Suc (Suc zero)) = [A,B]
  | f1 - = [];

fun f2 zero = [A,A]
  | f2 (Suc zero) = [B]
  | f2 (Suc (Suc zero)) = [B,A]
  | f2 - = [];

val (i1, j1) = higman g1;
val (v1, w1) = (g1 i1, g1 j1);
val (i2, j2) = higman g2;
val (v2, w2) = (g2 i2, g2 j2);
val (i3, j3) = higman f1;
val (v3, w3) = (f1 i3, f1 j3);
val (i4, j4) = higman f2;
val (v4, w4) = (f2 i4, f2 j4);

end;
```

⟫

definition

arbitrary-LT :: *LT* **where**
[*symmetric*, *code inline*]: *arbitrary-LT* = *arbitrary*

definition
arbitrary-TT :: *TT* **where**
[*symmetric*, *code inline*]: *arbitrary-TT* = *arbitrary*

code-datatype *L0 L1 arbitrary-LT*
code-datatype *T0 T1 T2 arbitrary-TT*

export-code *higman-idx* **in** *SML* **module-name** *Higman*

ML ⟪
```
local
  open Higman
in

val a = 16807.0;
val m = 2147483647.0;

fun nextRand seed =
  let val t = a*seed
  in  t - m * real (Real.floor(t/m)) end;

fun mk-word seed l =
  let
    val r = nextRand seed;
    val i = Real.round (r / m * 10.0);
  in if i > 7 andalso l > 2 then (r, []) else
    apsnd (cons (if i mod 2 = 0 then A else B)) (mk-word r (l+1))
  end;

fun f s Zero-nat = mk-word s 0
  | f s (Suc n) = f (fst (mk-word s 0)) n;

val g1 = snd o (f 20000.0);

val g2 = snd o (f 50000.0);

fun f1 Zero-nat = [A,A]
  | f1 (Suc Zero-nat) = [B]
  | f1 (Suc (Suc Zero-nat)) = [A,B]
  | f1 - = [];

fun f2 Zero-nat = [A,A]
  | f2 (Suc Zero-nat) = [B]
  | f2 (Suc (Suc Zero-nat)) = [B,A]
  | f2 - = [];
```

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 Efficient-Nat*
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*:
  $\bigwedge f. \ (\bigwedge i. \ i \leq Suc\ n \Longrightarrow f\ i \leq n) \Longrightarrow \exists\, i\ j. \ i \leq Suc\ n \wedge j < i \wedge f\ i = f\ j$
proof (*induct n*)
  case *0*
  hence $Suc\ 0 \leq Suc\ 0 \wedge 0 < Suc\ 0 \wedge f\ (Suc\ 0) = f\ 0$ **by** *simp*
  thus *?case* **by** *iprover*
next
  case (*Suc n*)
  {
    fix *k*
    have
      $k \leq Suc\ (Suc\ n) \Longrightarrow$
      $(\bigwedge i\ j. \ Suc\ k \leq i \Longrightarrow i \leq Suc\ (Suc\ n) \Longrightarrow j < i \Longrightarrow f\ i \neq f\ j) \Longrightarrow$
      $(\exists\, i\ j. \ i \leq k \wedge j < i \wedge f\ i = f\ j)$
    proof (*induct k*)
      case *0*
      let $?f = \lambda i. \ if\ f\ i = Suc\ n\ then\ f\ (Suc\ (Suc\ n))\ else\ f\ i$
      have $\neg\ (\exists\, i\ j. \ i \leq Suc\ n \wedge j < i \wedge ?f\ i = ?f\ j)$
      proof
        assume $\exists\, i\ j. \ i \leq Suc\ n \wedge j < i \wedge ?f\ i = ?f\ j$
        then obtain *i j* **where** *i*: $i \leq Suc\ n$ **and** *j*: $j < i$
          and *f*: $?f\ i = ?f\ j$ **by** *iprover*
        from *j* **have** *i-nz*: $Suc\ 0 \leq i$ **by** *simp*

```
  from i have iSSn: i ≤ Suc (Suc n) by simp
  have S0SSn: Suc 0 ≤ Suc (Suc n) by simp
  show False
  proof cases
    assume fi: f i = Suc n
    show False
    proof cases
      assume fj: f j = Suc n
      from i-nz and iSSn and j have f i ≠ f j by (rule 0)
      moreover from fi have f i = f j
        by (simp add: fj [symmetric])
      ultimately show ?thesis ..
    next
      from i and j have j < Suc (Suc n) by simp
      with S0SSn and le-refl have f (Suc (Suc n)) ≠ f j
        by (rule 0)
      moreover assume f j ≠ Suc n
      with fi and f have f (Suc (Suc n)) = f j by simp
      ultimately show False ..
    qed
  next
    assume fi: f i ≠ Suc n
    show False
    proof cases
      from i have i < Suc (Suc n) by simp
      with S0SSn and le-refl have f (Suc (Suc n)) ≠ f i
        by (rule 0)
      moreover assume f j = Suc n
      with fi and f have f (Suc (Suc n)) = f i by simp
      ultimately show False ..
    next
      from i-nz and iSSn and j
      have f i ≠ f j by (rule 0)
      moreover assume f j ≠ Suc n
      with fi and f have f i = f j by simp
      ultimately show False ..
    qed
  qed
qed
moreover have ⋀i. i ≤ Suc n ⟹ ?f i ≤ n
proof -
  fix i assume i ≤ Suc n
  hence i: i < Suc (Suc n) by simp
  have f (Suc (Suc n)) ≠ f i
    by (rule 0) (simp-all add: i)
  moreover have f (Suc (Suc n)) ≤ Suc n
    by (rule Suc) simp
  moreover from i have i ≤ Suc (Suc n) by simp
  hence f i ≤ Suc n by (rule Suc)
```

```
      ultimately show ?thesis i
        by simp
    qed
    hence ∃ i j. i ≤ Suc n ∧ j < i ∧ ?f i = ?f j
      by (rule Suc)
    ultimately show ?case ..
  next
    case (Suc k)
    from search [OF nat-eq-dec] show ?case
    proof
      assume ∃ j<Suc k. f (Suc k) = f j
      thus ?case by (iprover intro: le-refl)
    next
      assume nex: ¬ (∃ j<Suc k. f (Suc k) = f j)
      have ∃ i j. i ≤ k ∧ j < i ∧ f i = f j
      proof (rule Suc)
        from Suc show k ≤ Suc (Suc n) by simp
        fix i j assume k: Suc k ≤ i and i: i ≤ Suc (Suc n)
          and j: j < i
        show f i ≠ f j
        proof cases
          assume eq: i = Suc k
          show ?thesis
          proof
            assume f i = f j
            hence f (Suc k) = f j by (simp add: eq)
            with nex and j and eq show False by iprover
          qed
        next
          assume i ≠ Suc k
          with k have Suc (Suc k) ≤ 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 = 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:

```
theorem pigeonhole-slow:
  ⋀f. (⋀i. i ≤ Suc n ⟹ f i ≤ n) ⟹ ∃ i j. i ≤ Suc n ∧ j < i ∧ f i = f j
proof (induct n)
  case 0
  have Suc 0 ≤ Suc 0 ..
```

```
  moreover have 0 < Suc 0 ..
  moreover from 0 have f (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 ∃ j < Suc (Suc n). f (Suc (Suc n)) = f j
    thus ?case by (iprover intro: le-refl)
  next
    assume ¬ (∃ j < Suc (Suc n). f (Suc (Suc n)) = f j)
    hence nex: ∀ j < Suc (Suc n). f (Suc (Suc n)) ≠ f j by iprover
    let ?f = λi. if f i = Suc n then f (Suc (Suc n)) else f i
    have ⋀i. i ≤ Suc n ⟹ ?f i ≤ n
    proof −
      fix i assume i: i ≤ 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
    hence ∃ 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
qed
```

extract *pigeonhole pigeonhole-slow*

The programs extracted from the above proofs look as follows:

```
pigeonhole ≡
nat-rec (λx. (Suc 0, 0))
 (λx H2 xa.
     nat-rec arbitrary
      (λx H2.
          case search (Suc x) (λxb. nat-eq-dec (xa (Suc x)) (xa xb)) of
          None ⇒ let (x, y) = H2 in (x, y) | Some p ⇒ (Suc x, p))
      (Suc (Suc x)))
```

```
pigeonhole-slow ≡
nat-rec (λx. (Suc 0, 0))
 (λx H2 xa.
     case search (Suc (Suc x))
            (λxb. nat-eq-dec (xa (Suc (Suc x))) (xa xb)) of
     None ⇒
       let (x, y) = H2 (λi. if xa i = Suc x then xa (Suc (Suc x)) else xa i)
       in (x, y)
     | Some p ⇒ (Suc (Suc x), p))
```

The program for searching for an element in an array is

```
search ≡
λx H. nat-rec None
        (λy Ha.
            case Ha of None ⇒ case H y of Left ⇒ Some y | Right ⇒ None
            | Some p ⇒ Some p)
        x
```

The correctness statement for *pigeonhole* is

$$(\bigwedge i.\ i \le Suc\ n \Longrightarrow f\ i \le n) \Longrightarrow$$
$$fst\ (pigeonhole\ n\ f) \le Suc\ n\ \wedge$$
$$snd\ (pigeonhole\ n\ f) < fst\ (pigeonhole\ n\ f)\ \wedge$$
$$f\ (fst\ (pigeonhole\ n\ f)) = f\ (snd\ (pigeonhole\ n\ f))$$

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

definition
  test n u = *pigeonhole n* ($\lambda m.\ m - 1$)
definition
  test′ n u = *pigeonhole-slow n* ($\lambda m.\ m - 1$)
definition
  test′′ u = *pigeonhole 8* (*op* ! [0, 1, 2, 3, 4, 5, 6, 3, 7, 8])

consts-code
  arbitrary :: *nat* ({∗ 0::*nat* ∗})
  arbitrary :: *nat* × *nat* ({∗ (0::*nat*, 0::*nat*) ∗})

definition
  arbitrary-nat-pair :: *nat* × *nat* **where**
  [*symmetric*, *code inline*]: *arbitrary-nat-pair* = *arbitrary*

definition
  arbitrary-nat :: *nat* **where**
  [*symmetric*, *code inline*]: *arbitrary-nat* = *arbitrary*

code-const *arbitrary-nat-pair* (*SML* (~1, ~1))

code-const *arbitrary-nat* (*SML* ~1)

code-module *PH1*
contains
  test = *test*
  test′ = *test′*
  test′′ = *test′′*

export-code *test test′ test′′* **in** *SML* **module-name** *PH2*

ML *timeit* (*PH1.test 10*)
ML *timeit* (*PH2.test 10*)

ML *timeit* (*PH1.test′ 10*)
ML *timeit* (*PH2.test′ 10*)

ML *timeit* (*PH1.test 20*)
ML *timeit* (*PH2.test 20*)

ML *timeit* (*PH1.test′ 20*)
ML *timeit* (*PH2.test′ 20*)

ML *timeit* (*PH1.test 25*)
ML *timeit* (*PH2.test 25*)

ML *timeit* (*PH1.test′ 25*)
ML *timeit* (*PH2.test′ 25*)

ML *timeit* (*PH1.test 500*)
ML *timeit* (*PH2.test 500*)

ML *timeit PH1.test″*
ML *timeit PH2.test″*

end

7 Euclid's theorem

theory *Euclid*
imports *~~/src/HOL/NumberTheory/Factorization Efficient-Nat Util*
begin

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

lemma *prime-eq*: $prime\ p = (1 < p \wedge (\forall m.\ m\ dvd\ p \longrightarrow 1 < m \longrightarrow m = p))$
  apply (*simp add*: *prime-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′*: $prime\ p = (1 < p \wedge (\forall m\ k.\ p = m * k \longrightarrow 1 < m \longrightarrow m = p))$
  by (*simp add*: *prime-eq dvd-def all-simps* [*symmetric*] *del*: *all-simps*)

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

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

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

Unfortunately, the proof in the *Factorization* theory using *metis* is non-constructive.

```
lemma split-primel':
  primel xs ⟹ primel ys ⟹ ∃ l. primel l ∧ prod l = prod xs ∗ prod ys
  apply (rule exI)
  apply safe
   apply (rule-tac [2] prod-append)
  apply (simp add: primel-append)
  done

lemma factor-exists: Suc 0 < n ⟹ (∃ l. primel l ∧ prod l = n)
proof (induct n rule: nat-wf-ind)
  case (1 n)
  from ‹Suc 0 < n›
  have (∃ m k. Suc 0 < m ∧ Suc 0 < k ∧ m < n ∧ k < n ∧ n = m ∗ k) ∨ prime
n
    by (rule not-prime-ex-mk)
  then show ?case
  proof
    assume ∃ m k. Suc 0 < m ∧ Suc 0 < k ∧ m < n ∧ k < n ∧ n = m ∗ k
    then obtain m k where 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 ∃ l. primel l ∧ prod l = m by (rule 1)
    then obtain l1 where primel-l1: primel l1 and prod-l1-m: prod l1 = m
      by iprover
    from kn and k have ∃ l. primel l ∧ prod l = k by (rule 1)
    then obtain l2 where primel-l2: primel l2 and prod-l2-k: prod l2 = k
      by iprover
    from primel-l1 primel-l2
    have ∃ l. primel l ∧ prod l = prod l1 ∗ prod l2
      by (rule split-primel')
    with prod-l1-m prod-l2-k nmk show ?thesis by simp
  next
    assume prime n
    hence primel [n] ∧ prod [n] = n by (rule prime-primel)
    thus ?thesis ..
  qed
qed

lemma dvd-prod [iff]: n dvd prod (n # ns)
  by simp

consts fact :: nat ⇒ nat    ((-!) [1000] 999)
```

```
primrec
  0! = 1
  (Suc n)! = n! * Suc n

lemma fact-greater-0 [iff]: 0 < n!
  by (induct n) simp-all

lemma dvd-factorial: 0 < m ⟹ m ≤ n ⟹ m dvd n!
proof (induct n)
  case 0
  then show ?case by simp
next
  case (Suc n)
  from ‹m ≤ Suc n› show ?case
  proof (rule le-SucE)
    assume m ≤ n
    with ‹0 < m› have m dvd n! by (rule Suc)
    then have m dvd (n! * Suc n) by (rule dvd-mult2)
    then show ?thesis by simp
  next
    assume m = Suc n
    then have m dvd (n! * Suc n)
      by (auto intro: dvdI simp: mult-ac)
    then show ?thesis by simp
  qed
qed

lemma prime-factor-exists:
  assumes N: (1::nat) < n
  shows ∃ p. prime p ∧ p dvd n
proof −
  from N obtain l where primel-l: primel l
    and prod-l: n = prod l using factor-exists
    by simp iprover
  from prems have l ≠ []
    by (auto simp add: primel-nempty-g-one)
  then obtain x xs where l: l = x # xs
    by (cases l) simp
  from primel-l l have prime x by (simp add: primel-hd-tl)
  moreover from primel-l l prod-l
  have x dvd n by (simp only: dvd-prod)
  ultimately show ?thesis by iprover
qed
```

Euclid's theorem: there are infinitely many primes.

```
lemma Euclid: ∃ p. prime p ∧ n < p
proof −
  let ?k = n! + 1
  have 1 < n! + 1 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-g-zero)
      then have p dvd n! using pn by (rule dvd-factorial)
      with dvd have p dvd ?k − n! by (rule dvd-diff)
      then have p dvd 1 by simp
      with prime show False using prime-nd-one 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 (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-primel′ (H2 x) (H2 y))

consts-code
  arbitrary ((error arbitrary))

code-module Prime
contains Euclid

ML Prime.factor-exists 1007
ML Prime.factor-exists 567
ML Prime.factor-exists 345
ML Prime.factor-exists 999
ML Prime.factor-exists 876

ML Prime.Euclid 0
ML Prime.Euclid it
ML Prime.Euclid it
ML Prime.Euclid it

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.