Examples for program extraction in Higher-Order Logic

Stefan Berghofer

January 18, 2026

Contents

1 Auxiliary lemmas used in program extraction examples

theory *Util*
imports *Main*
begin

Decidability of equality on natural numbers.

lemma *nat-eq-dec*: $\bigwedge n::nat.\ m = n \vee m \neq n$
  ⟨*proof*⟩

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*⟩

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*⟩

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*⟩

extract *division*

The program extracted from the above proof looks as follows

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

The corresponding correctness theorem is

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

lemma $division\ 9\ 2 = (0,\ 3)$ ⟨*proof*⟩

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*⟩

extract *greatest-common-divisor*

The extracted program for computing the greatest common divisor is

greatest-common-divisor ≡
λ*x*. *nat-wf-ind-P x*
  (λ*x H2 xa*.
    let (*xa*, *y*) = *division xa x*
    in nat-exhaust-P xa (*Suc x*, *y*, *1*)
      (λ*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 ⟨*proof*⟩

end

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

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

instance ⟨*proof*⟩

end

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

definition *default* = (λ*x*. *default*)

instance ⟨*proof*⟩

end

lemma *greatest-common-divisor 7 12* = (*4*, *3*, *2*) ⟨*proof*⟩

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

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

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*]: $\bigwedge x.\ is\text{-}path'\ r\ x\ (ys\ @\ [y])\ z = (is\text{-}path'\ r\ x\ ys\ y \wedge r\ y\ z = T)$
  ⟨*proof*⟩

theorem *list-all-scoc* [*simp*]: $list\text{-}all\ P\ (xs\ @\ [x]) \longleftrightarrow P\ x \wedge list\text{-}all\ P\ xs$
  ⟨*proof*⟩

theorem *list-all-lemma*: $list\text{-}all\ P\ xs \Longrightarrow (\bigwedge x.\ P\ x \Longrightarrow Q\ x) \Longrightarrow list\text{-}all\ Q\ xs$
⟨*proof*⟩

theorem *lemma1*: $\bigwedge p.\ is\text{-}path\ r\ p\ i\ j\ k \Longrightarrow is\text{-}path\ r\ p\ (Suc\ i)\ j\ k$
  ⟨*proof*⟩

theorem *lemma2*: $\bigwedge p.\ is\text{-}path\ r\ p\ 0\ j\ k \Longrightarrow r\ j\ k = T$
  ⟨*proof*⟩

theorem *is-path'-conc*: $is\text{-}path'\ r\ j\ xs\ i \Longrightarrow is\text{-}path'\ r\ i\ ys\ k \Longrightarrow$
  $is\text{-}path'\ r\ j\ (xs\ @\ i\ \#\ ys)\ k$
⟨*proof*⟩

theorem *lemma3*:
  $\bigwedge p\ q.\ is\text{-}path\ r\ p\ i\ j\ i \Longrightarrow is\text{-}path\ r\ q\ i\ i\ k \Longrightarrow$
    $is\text{-}path\ r\ (conc\ p\ q)\ (Suc\ i)\ j\ k$
  ⟨*proof*⟩

theorem *lemma5*:
  $\bigwedge p.\ is\text{-}path\ r\ p\ (Suc\ i)\ j\ k \Longrightarrow \neg\ is\text{-}path\ r\ p\ i\ j\ k \Longrightarrow$

$(\exists\, q.\ \textit{is-path}\ r\ q\ i\ j\ i) \wedge (\exists\, q'.\ \textit{is-path}\ r\ q'\ i\ i\ k)$
⟨*proof*⟩

theorem *lemma5′*:
$\bigwedge p.\ \textit{is-path}\ r\ p\ (\textit{Suc}\ i)\ j\ k \Longrightarrow \neg\ \textit{is-path}\ r\ p\ i\ j\ k \Longrightarrow$
$\neg\ (\forall\, q.\ \neg\ \textit{is-path}\ r\ q\ i\ j\ i) \wedge \neg\ (\forall\, q'.\ \neg\ \textit{is-path}\ r\ q'\ i\ i\ k)$
⟨*proof*⟩

theorem *warshall*: $\bigwedge j\ k.\ \neg\ (\exists\, p.\ \textit{is-path}\ r\ p\ i\ j\ k) \vee (\exists\, p.\ \textit{is-path}\ r\ p\ i\ j\ k)$
⟨*proof*⟩

extract *warshall*

The program extracted from the above proof looks as follows

```
warshall ≡
λx xa xb xc.
   nat-induct-P xa
    (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)
    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*⟩

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* ⟹ *R a* (*w* # *vs*) ((*a* # *w*) # *ws*)

inductive *T* :: *letter* ⇒ *letter list list* ⇒ *letter list list* ⇒ *bool*
for *a* :: *letter*
where
T0 [*Pure.intro*]: *a* ≠ *b* ⟹ *R b ws zs* ⟹ *T a* (*w* # *zs*) ((*a* # *w*) # *zs*)
| *T1* [*Pure.intro*]: *T a ws zs* ⟹ *T a* (*w* # *ws*) ((*a* # *w*) # *zs*)
| *T2* [*Pure.intro*]: *a* ≠ *b* ⟹ *T a ws zs* ⟹ *T a ws* ((*b* # *w*) # *zs*)

inductive *bar* :: *letter list list* ⇒ *bool*
where
bar1 [*Pure.intro*]: *good ws* ⟹ *bar ws*
| *bar2* [*Pure.intro*]: (⋀*w*. *bar* (*w* # *ws*)) ⟹ *bar ws*

theorem *prop1*: *bar* ([] # *ws*)
⟨*proof*⟩

theorem *lemma1*: *L as ws* ⟹ *L* (*a* # *as*) *ws*
⟨*proof*⟩

lemma *lemma2'*: *R a vs ws* ⟹ *L as vs* ⟹ *L* (*a* # *as*) *ws*
⟨*proof*⟩

lemma *lemma2*: *R a vs ws* ⟹ *good vs* ⟹ *good ws*
⟨*proof*⟩

lemma *lemma3'*: *T a vs ws* ⟹ *L as vs* ⟹ *L* (*a* # *as*) *ws*
⟨*proof*⟩

lemma *lemma3*: *T a ws zs* ⟹ *good ws* ⟹ *good zs*

⟨*proof*⟩

lemma *lemma4*: $R\ a\ ws\ zs \Longrightarrow ws \neq [] \Longrightarrow T\ a\ ws\ zs$
⟨*proof*⟩

lemma *letter-neq*: $a \neq b \Longrightarrow c \neq a \Longrightarrow c = b$ **for** $a\ b\ c :: letter$
⟨*proof*⟩

lemma *letter-eq-dec*: $a = b \vee a \neq b$ **for** $a\ b :: letter$
⟨*proof*⟩

theorem *prop2*:
assumes *ab*: $a \neq b$ **and** *bar*: $bar\ xs$
shows $\bigwedge ys\ zs.\ bar\ ys \Longrightarrow T\ a\ xs\ zs \Longrightarrow T\ b\ ys\ zs \Longrightarrow bar\ zs$
⟨*proof*⟩

theorem *prop3*:
assumes *bar*: $bar\ xs$
shows $\bigwedge zs.\ xs \neq [] \Longrightarrow R\ a\ xs\ zs \Longrightarrow bar\ zs$
⟨*proof*⟩

theorem *higman*: $bar\ []$
⟨*proof*⟩

primrec *is-prefix* :: $'a\ list \Rightarrow (nat \Rightarrow 'a) \Rightarrow bool$
where
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* $ws\ f \Longrightarrow \exists\, i.\ emb\ (f\ i)\ w \wedge i < length\ ws$
⟨*proof*⟩

theorem *good-idx*:
assumes *good*: $good\ ws$
shows *is-prefix* $ws\ f \Longrightarrow \exists\, i\ j.\ emb\ (f\ i)\ (f\ j) \wedge i < j$
⟨*proof*⟩

theorem *bar-idx*:
assumes *bar*: $bar\ ws$
shows *is-prefix* $ws\ f \Longrightarrow \exists\, i\ j.\ emb\ (f\ i)\ (f\ j) \wedge i < j$
⟨*proof*⟩

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*⟩

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-prefix vs f* $\wedge$ *good vs*
⟨*proof*⟩

theorem *good-prefix*: $\exists$ *vs. is-prefix vs f* $\wedge$ *good vs*
⟨*proof*⟩

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*:

higman-idx ≡ λ*x. bar-idx x higman*

Corresponding correctness theorem:

emb (*f* (*fst* (*higman-idx f*))) (*f* (*snd* (*higman-idx f*))) ∧
fst (*higman-idx f*) < *snd* (*higman-idx f*)

Program extracted from the proof of *higman*:

higman ≡
bar2 [] (*rec-list* (*prop1* []) (λ*a w H. prop3 a* [*a* # *w*] *H* (*R1* [] [] *w R0*)))

Program extracted from the proof of *prop1*:

prop1 ≡
λ*x. bar2* ([] # *x*) (λ*w. bar1* (*w* # [] # *x*) (*good0 w* ([] # *x*) (*L0* [] *x*)))

Program extracted from the proof of *prop2*:

prop2 ≡
λ*x xa xb xc H.*
compat-barT.rec-split-barT
(λ*ws xa xb xba H Ha Haa. bar1 xba* (*lemma3 x Ha xa*))

```
(λws xb r xba xbb H.
    compat-barT.rec-split-barT (λws x xb H Ha. bar1 xb (lemma3 xa Ha x))
     (λwsa xb ra xc H Ha.
         bar2 xc
          (λw. case w of [] ⇒ prop1 xc
                | a # 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 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 ⟨*proof*⟩

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)
   }))
  ⟨proof⟩
termination
  ⟨proof⟩

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⟩

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*:
$\bigwedge f.\ (\bigwedge i.\ i \leq Suc\ n \Longrightarrow f\ i \leq n) \Longrightarrow \exists\, i\ j.\ i \leq Suc\ n \wedge j < i \wedge f\ i = f\ j$
⟨*proof*⟩

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

theorem *pigeonhole-slow*:
$\bigwedge f.\ (\bigwedge i.\ i \leq Suc\ n \Longrightarrow f\ i \leq n) \Longrightarrow \exists\, i\ j.\ i \leq Suc\ n \wedge j < i \wedge f\ i = f\ j$
⟨*proof*⟩

extract *pigeonhole pigeonhole-slow*

The programs extracted from the above proofs look as follows:

```
pigeonhole ≡
λx. nat-induct-P x (λx. (Suc 0, 0))
      (λx H2 xa.
          nat-induct-P (Suc (Suc x)) default
           (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)))

pigeonhole-slow ≡
λx. nat-induct-P x (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-induct-P x None
         (λy Ha.
             case Ha of None ⇒ case H y of Left ⇒ Some y | Right ⇒ None
             | Some p ⇒ Some p)
```

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.

instantiation *nat* :: *default*
begin

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

instance ⟨*proof*⟩

end

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

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

instance ⟨*proof*⟩

end

definition $test\ n\ u = pigeonhole\ (nat\text{-}of\text{-}integer\ n)\ (\lambda m.\ m - 1)$
definition $test'\ n\ u = pigeonhole\text{-}slow\ (nat\text{-}of\text{-}integer\ n)\ (\lambda m.\ m - 1)$
definition $test''\ u = pigeonhole\ 8\ (List.nth\ [0, 1, 2, 3, 4, 5, 6, 3, 7, 8])$

⟨*ML*⟩

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*: $n = m * k \Longrightarrow m < n \Longrightarrow k < n \Longrightarrow Suc\ 0 < m$
⟨*proof*⟩

lemma *factor-greater-one2*: $n = m * k \Longrightarrow m < n \Longrightarrow k < n \Longrightarrow Suc\ 0 < k$
⟨*proof*⟩

lemma *prod-mn-less-k*: $0 < n \Longrightarrow 0 < k \Longrightarrow Suc\ 0 < m \Longrightarrow m * n = k \Longrightarrow n < k$
⟨*proof*⟩

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

lemma *prime-eq'*: $prime\ (p::nat) \longleftrightarrow 1 < p \wedge (\forall m\ k.\ p = m * k \longrightarrow 1 < m \longrightarrow m = p)$
⟨*proof*⟩

lemma *not-prime-ex-mk*:
assumes n: $Suc\ 0 < n$
shows $(\exists m\ k.\ Suc\ 0 < m \wedge Suc\ 0 < k \wedge m < n \wedge k < n \wedge n = m * k) \vee prime\ n$
⟨*proof*⟩

lemma *dvd-factorial*: $0 < m \Longrightarrow m \leq n \Longrightarrow m\ dvd\ fact\ n$
⟨*proof*⟩

lemma *dvd-prod* [*iff*]: $n\ dvd\ (\prod m::nat \in\#\ mset\ (n\ \#\ ns).\ m)$
⟨*proof*⟩

definition *all-prime* :: *nat list* ⇒ *bool*
where *all-prime ps* ⟷ (∀ *p*∈*set ps. prime p*)

lemma *all-prime-simps*:
all-prime []
all-prime (*p* # *ps*) ⟷ *prime p* ∧ *all-prime ps*
⟨*proof*⟩

lemma *all-prime-append*: *all-prime* (*ps* @ *qs*) ⟷ *all-prime ps* ∧ *all-prime qs*
⟨*proof*⟩

lemma *split-all-prime*:
assumes *all-prime ms* **and** *all-prime ns*
shows ∃ *qs. all-prime qs* ∧
$(\prod m::nat \in\#\ mset\ qs.\ m) = (\prod m::nat \in\#\ mset\ ms.\ m) * (\prod m::nat \in\#\ mset\ ns.\ m)$
(**is** ∃ *qs. ?P qs* ∧ *?Q qs*)
⟨*proof*⟩

lemma *all-prime-nempty-g-one*:
assumes *all-prime ps* **and** *ps* ≠ []
shows $Suc\ 0 < (\prod m::nat \in\#\ mset\ ps.\ m)$

⟨*proof*⟩

lemma *factor-exists*: $Suc\ 0 < n \Longrightarrow (\exists ps.\ all\text{-}prime\ ps \wedge (\prod m{::}nat \in\#\ mset\ ps.\ m) = n)$
⟨*proof*⟩

lemma *prime-factor-exists*:
  assumes N: $(1{::}nat) < n$
  shows $\exists p.\ prime\ p \wedge p\ dvd\ n$
⟨*proof*⟩

Euclid's theorem: there are infinitely many primes.

lemma *Euclid*: $\exists p{::}nat.\ prime\ p \wedge n < p$
⟨*proof*⟩

extract *Euclid*

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

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

The program corresponding to the proof of the factorization theorem is

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

instantiation *nat* :: *default*
begin

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

instance ⟨*proof*⟩

end

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

definition *default* = []

instance ⟨*proof*⟩

end

primrec *iterate* :: $nat \Rightarrow ('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a\ list$
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] ⟨proof⟩
lemma factor-exists 567 = [7, 3, 3, 3, 3] ⟨proof⟩
lemma factor-exists 345 = [23, 5, 3] ⟨proof⟩
lemma factor-exists 999 = [37, 3, 3, 3] ⟨proof⟩
lemma factor-exists 876 = [73, 3, 2, 2] ⟨proof⟩

lemma iterate 4 Euclid 0 = [2, 3, 7, 71] ⟨proof⟩

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.