Examples for program extraction in Higher-Order Logic

Stefan Berghofer

November 22, 2007

Contents

1 Auxiliary lemmas used in program extraction examples

theory *Util*
imports *Main*
begin

Decidability of equality on natural numbers.

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

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* **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 ≡
λx xa.
   nat-rec (0, 0)
    (λa H. let (x, y) = H
             in case nat-eq-dec x xa of Left ⇒ (0, Suc y)
                 | Right ⇒ (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*:
$\bigwedge n::nat.\ Suc\ m < n \Longrightarrow \exists k\ n1\ m1.\ k * n1 = n \wedge k * m1 = Suc\ m \wedge$
$(\forall l\ l1\ l2.\ l * l1 = n \longrightarrow l * l2 = Suc\ m \longrightarrow l \leq k)$
$\langle proof \rangle$

extract *greatest-common-divisor*

The extracted program for computing the greatest common divisor is

$greatest\text{-}common\text{-}divisor \equiv$
$\lambda x.\ nat\text{-}wf\text{-}ind\text{-}P\ x$
$\quad (\lambda x\ H2\ xa.$
$\quad\quad let\ (xa,\ y) = division\ xa\ x$
$\quad\quad in\ case\ xa\ of\ 0 \Rightarrow (Suc\ x,\ y,\ 1)$
$\quad\quad\quad |\ Suc\ nat \Rightarrow$
$\quad\quad\quad\quad let\ (x,\ ya) = H2\ nat\ (Suc\ x);\ (xa,\ ya) = ya$
$\quad\quad\quad\quad in\ (x,\ xa * y + ya,\ xa))$

consts-code
arbitrary ((*error arbitrary*))

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

$\langle ML \rangle$

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

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

constdefs

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

conc :: $('a * 'a\ list * 'a) \Rightarrow ('a * 'a\ list * 'a) \Rightarrow ('a * 'a\ list * 'a)$
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*]: $list\text{-}all\ P\ (xs\ @\ [x]) = (P\ x \wedge list\text{-}all\ P\ xs)$
⟨*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*:
$\bigwedge p.\ is\text{-}path\ r\ p\ (Suc\ i)\ j\ k \Longrightarrow {\sim}\ is\text{-}path\ r\ p\ i\ j\ k \Longrightarrow$
$(\exists q.\ is\text{-}path\ r\ q\ i\ j\ i) \wedge (\exists q'.\ is\text{-}path\ r\ q'\ i\ i\ k)$
⟨*proof*⟩

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

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

$\langle proof \rangle$

extract *warshall*

The program extracted from the above proof looks as follows

warshall ≡
λx 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*]: $L\ w\ ws \Longrightarrow good\ (w\ \#\ ws)$
| *good1* [*Pure.intro*]: $good\ ws \Longrightarrow good\ (w\ \#\ ws)$

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*]: $R\ a\ vs\ ws \Longrightarrow R\ a\ (w\ \#\ vs)\ ((a\ \#\ w)\ \#\ ws)$

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*: $bar\ ([]\ \#\ ws)$ ⟨*proof*⟩

theorem *lemma1*: $L\ as\ ws \Longrightarrow L\ (a\ \#\ as)\ ws$
 ⟨*proof*⟩

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

lemma *lemma2*: $R\ a\ vs\ ws \Longrightarrow good\ vs \Longrightarrow good\ ws$
 ⟨*proof*⟩

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

lemma *lemma3*: $T\ a\ ws\ zs \Longrightarrow good\ ws \Longrightarrow good\ zs$
 ⟨*proof*⟩

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

lemma *letter-neq*: $(a{::}letter) \neq b \Longrightarrow c \neq a \Longrightarrow c = b$
 ⟨*proof*⟩

lemma *letter-eq-dec*: $(a{::}letter) = b \vee a \neq b$
 ⟨*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*⟩

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

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

theorem *L-idx*:
assumes *L*: *L w ws*
shows $is\text{-}prefix\ ws\ f \Longrightarrow \exists\, i.\ emb\ (f\ i)\ w \wedge i < length\ ws$ ⟨*proof*⟩

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

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

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

5.1 Extracting the program

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

declare *bar.induct* [*ind-realizer*]

extract *higman-idx*

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

```
higman-idx ≡ λx. 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 [] (list-rec (prop1 []) (λa 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*⟩

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*⟩

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

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

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

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

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

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

consts-code
  $arbitrary :: nat\ (\{*\ 0::nat\ *\})$
  $arbitrary :: nat \times nat\ (\{*\ (0::nat, 0::nat)\ *\})$

definition
  $arbitrary\text{-}nat\text{-}pair :: nat \times nat$ **where**
  $[symmetric, code\ inline]: arbitrary\text{-}nat\text{-}pair = arbitrary$

definition
  $arbitrary\text{-}nat :: nat$ **where**
  $[symmetric, code\ inline]: arbitrary\text{-}nat = arbitrary$

code-const $arbitrary\text{-}nat\text{-}pair\ (SML\ (\sim 1, \sim 1))$

code-const $arbitrary\text{-}nat\ (SML\ \sim 1)$

code-module $PH1$
contains
  $test = test$
  $test' = test'$
  $test'' = test''$

export-code $test\ test'\ test''$ **in** SML **module-name** $PH2$

$\langle ML\rangle$

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* ∧ (∀ *m*. *m dvd p* ⟶ *1* < *m* ⟶ *m* = *p*))
⟨*proof*⟩

lemma *prime-eq′*: *prime p* = (*1* < *p* ∧ (∀ *m k*. *p* = *m* ∗ *k* ⟶ *1* < *m* ⟶ *m* = *p*))
⟨*proof*⟩

lemma *factor-greater-one1*: *n* = *m* ∗ *k* ⟹ *m* < *n* ⟹ *k* < *n* ⟹ *Suc 0* < *m*
⟨*proof*⟩

lemma *factor-greater-one2*: *n* = *m* ∗ *k* ⟹ *m* < *n* ⟹ *k* < *n* ⟹ *Suc 0* < *k*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

lemma *factor-exists*: *Suc 0* < *n* ⟹ (∃ *l*. *primel l* ∧ *prod l* = *n*)
⟨*proof*⟩

lemma *dvd-prod* [*iff*]: *n dvd prod* (*n* # *ns*)
⟨*proof*⟩

consts *fact* :: *nat* ⇒ *nat*   ((-!) [*1000*] *999*)
primrec
0! = *1*
(*Suc n*)! = *n*! ∗ *Suc n*

lemma *fact-greater-0* [*iff*]: *0* < *n*!
⟨*proof*⟩

lemma *dvd-factorial*: *0* < *m* ⟹ *m* ≤ *n* ⟹ *m dvd n*!
⟨*proof*⟩

lemma *prime-factor-exists*:
assumes *N*: (*1*::*nat*) < *n*
shows ∃ *p*. *prime p* ∧ *p dvd n*
⟨*proof*⟩

Euclid's theorem: there are infinitely many primes.

lemma *Euclid*: ∃ *p*. *prime p* ∧ *n* < *p*

⟨*proof*⟩

extract *Euclid*

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

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

The program corresponding to the proof of the factorization theorem is

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

consts-code
arbitrary ((*error arbitrary*))

code-module *Prime*
contains *Euclid*

⟨*ML*⟩

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.