

# Examples for program extraction in Higher-Order Logic

Stefan Berghofer

December 17, 2025

## Contents

<b>1</b>	<b>Auxiliary lemmas used in program extraction examples</b>	<b>1</b>
<b>2</b>	<b>Quotient and remainder</b>	<b>2</b>
<b>3</b>	<b>Greatest common divisor</b>	<b>2</b>
<b>4</b>	<b>Warshall's algorithm</b>	<b>3</b>
<b>5</b>	<b>Higman's lemma</b>	<b>5</b>
5.1	Extracting the program . . . . .	8
5.2	Some examples . . . . .	9
<b>6</b>	<b>The pigeonhole principle</b>	<b>10</b>
<b>7</b>	<b>Euclid's theorem</b>	<b>12</b>

## 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:  $\bigwedge x::nat. (\bigwedge y. y < x \implies P\ y) \implies P\ x$ 
```

**shows**  $P\ z$   
 $\langle proof \rangle$

Bounded search for a natural number satisfying a decidable predicate.

**lemma** *search*:  
**assumes**  $dec: \bigwedge x::nat. P\ x \vee \neg P\ x$   
**shows**  $(\exists x < y. P\ x) \vee \neg (\exists x < y. P\ x)$   
 $\langle proof \rangle$

**end**

## 2 Quotient and remainder

**theory** *QuotRem*  
**imports** *Util HOL-Library.Realizers*  
**begin**

Derivation of quotient and remainder using program extraction.

**theorem** *division*:  $\exists r\ q. a = Suc\ b * q + r \wedge r \leq b$   
 $\langle proof \rangle$

**extract** *division*

The program extracted from the above proof looks as follows

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

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* =  $(0, 3)$   $\langle proof \rangle$

**end**

## 3 Greatest common divisor

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

**theorem** *greatest-common-divisor*:  
 $\bigwedge n::nat. Suc\ m < n \implies$

$\exists k \ n1 \ m1. k * n1 = n \wedge k * m1 = \text{Suc } m \wedge$   
 $(\forall l \ l1 \ l2. l * l1 = n \longrightarrow l * l2 = \text{Suc } m \longrightarrow l \leq k)$   
 $\langle \text{proof} \rangle$

**extract** *greatest-common-divisor*

The extracted program for computing the greatest common divisor is

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

**instantiation** *nat* :: *default*

**begin**

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

**instance**  $\langle \text{proof} \rangle$

**end**

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

**begin**

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

**instance**  $\langle \text{proof} \rangle$

**end**

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

**begin**

**definition** *default* = ( $\lambda x. \text{ default}$ )

**instance**  $\langle \text{proof} \rangle$

**end**

**lemma** *greatest-common-divisor* 7 12 = (4, 3, 2)  $\langle \text{proof} \rangle$

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

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

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

**theorem**  $lemma1: \bigwedge p. is-path\ r\ p\ i\ j\ k \Longrightarrow is-path\ r\ p\ (Suc\ i)\ j\ k$   
 $\langle proof \rangle$

**theorem**  $lemma2: \bigwedge p. is-path\ r\ p\ 0\ j\ k \Longrightarrow r\ j\ k = T$   
 $\langle proof \rangle$

**theorem**  $is-path'-conc: is-path'\ r\ j\ xs\ i \Longrightarrow is-path'\ r\ i\ ys\ k \Longrightarrow is-path'\ r\ j\ (xs @ i \# ys)\ k$   
 $\langle proof \rangle$

**theorem**  $lemma3:$   
 $\bigwedge p\ q. is-path\ r\ p\ i\ j\ i \Longrightarrow is-path\ r\ q\ i\ i\ k \Longrightarrow is-path\ r\ (conc\ p\ q)\ (Suc\ i)\ j\ k$   
 $\langle proof \rangle$

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

$(\exists q. \text{is-path } r \ q \ i \ j \ i) \wedge (\exists q'. \text{is-path } r \ q' \ i \ i \ k)$   
 $\langle \text{proof} \rangle$

**theorem** *lemma5'*:

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

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

**extract** *warshall*

The program extracted from the above proof looks as follows

*warshall*  $\equiv$   
 $\lambda x \ x a \ x b \ x c.$   
 $\text{nat-induct-}P \ x a$   
 $(\lambda x a \ x b. \text{case } x \ x a \ x b \text{ of } T \Rightarrow \text{Some } (x a, [], x b) \mid F \Rightarrow \text{None})$   
 $(\lambda x \ H2 \ x a \ x b.$   
 $\text{case } H2 \ x a \ x b \text{ of}$   
 $\text{None} \Rightarrow$   
 $\text{case } H2 \ x a \ x \text{ of } \text{None} \Rightarrow \text{None}$   
 $\mid \text{Some } q \Rightarrow$   
 $\text{case } H2 \ x \ x b \text{ of } \text{None} \Rightarrow \text{None} \mid \text{Some } q a \Rightarrow \text{Some } (\text{conc } q \ q a)$   
 $\mid \text{Some } q \Rightarrow \text{Some } q)$   
 $x b \ x c$

The corresponding correctness theorem is

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

$\langle ML \rangle$

**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*  $\mid$  *B*

**inductive** *emb* :: *letter list*  $\Rightarrow$  *letter list*  $\Rightarrow$  *bool*  
**where**

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

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

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

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

**inductive**  $T :: letter \Rightarrow letter list list \Rightarrow letter list list \Rightarrow bool$   
**for**  $a :: letter$   
**where**  
 $T0$  [Pure.intro]:  $a \neq b \implies R b ws zs \implies T a (w \# zs) ((a \# w) \# zs)$   
 $| T1$  [Pure.intro]:  $T a ws zs \implies T a (w \# ws) ((a \# w) \# zs)$   
 $| T2$  [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**  $prop1$ :  $bar ([] \# ws)$   
 $\langle proof \rangle$

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

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

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

**lemma**  $lemma3'$ :  $T a vs ws \implies L as vs \implies L (a \# as) ws$   
 $\langle proof \rangle$

**lemma**  $lemma3$ :  $T a ws zs \implies good ws \implies good zs$

*<proof>*

**lemma** *lemma4*:  $R\ a\ ws\ zs \implies ws \neq [] \implies T\ a\ ws\ zs$   
*<proof>*

**lemma** *letter-neg*:  $a \neq b \implies c \neq a \implies c = b$  **for**  $a\ b\ c :: \text{letter}$   
*<proof>*

**lemma** *letter-eq-dec*:  $a = b \vee a \neq b$  **for**  $a\ b :: \text{letter}$   
*<proof>*

**theorem** *prop2*:  
 **assumes** *ab*:  $a \neq b$  **and** *bar*:  $\text{bar}\ xs$   
 **shows**  $\bigwedge ys\ zs. \text{bar}\ ys \implies T\ a\ xs\ zs \implies T\ b\ ys\ zs \implies \text{bar}\ zs$   
*<proof>*

**theorem** *prop3*:  
 **assumes** *bar*:  $\text{bar}\ xs$   
 **shows**  $\bigwedge zs. xs \neq [] \implies R\ a\ xs\ zs \implies \text{bar}\ zs$   
*<proof>*

**theorem** *higman*:  $\text{bar}\ []$   
*<proof>*

**primrec** *is-prefix* ::  $'a\ \text{list} \Rightarrow (\text{nat} \Rightarrow 'a) \Rightarrow \text{bool}$   
**where**  
 *is-prefix* []  $f = \text{True}$   
 | *is-prefix* ( $x \# xs$ )  $f = (x = f\ (\text{length}\ xs) \wedge \text{is-prefix}\ xs\ f)$

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

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

**theorem** *bar-idx*:  
 **assumes** *bar*:  $\text{bar}\ ws$   
 **shows**  $\text{is-prefix}\ ws\ f \implies \exists i\ j. \text{emb}\ (f\ i)\ (f\ j) \wedge i < j$   
*<proof>*

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

**theorem** *higman-idx*:  $\exists (i::\text{nat})\ j. \text{emb}\ (f\ i)\ (f\ j) \wedge i < j$   
*<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*  $\implies \exists vs. is-prefix vs f \wedge good vs$   
 $\langle proof \rangle$

**theorem** *good-prefix*:  $\exists vs. is-prefix vs f \wedge good vs$   
 $\langle proof \rangle$

**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*:

$higman-idx \equiv \lambda x. bar-idx x higman$

Corresponding correctness theorem:

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

Program extracted from the proof of *higman*:

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

Program extracted from the proof of *prop1*:

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

Program extracted from the proof of *prop2*:

$prop2 \equiv$   
 $\lambda x xa xb xc H.$   
 $compat-barT.rec-split-barT$   
 $(\lambda ws xa xb xba H Ha Haa. bar1 xba (lemma3 x Ha xa))$



```

(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*:

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

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

**theorem** *pigeonhole-slow*:

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

**extract** *pigeonhole pigeonhole-slow*

The programs extracted from the above proofs look as follows:

*pigeonhole*  $\equiv$

$\lambda x. \text{nat-induct-}P\ x\ (\lambda x. (\text{Suc } 0, 0))$   
 $(\lambda x\ H2\ xa.$   
 $\quad \text{nat-induct-}P\ (\text{Suc } (\text{Suc } x))\ \text{default}$   
 $\quad (\lambda x\ H2.$   
 $\quad \quad \text{case search } (\text{Suc } x)\ (\lambda xb. \text{nat-eq-dec } (xa\ (\text{Suc } x))\ (xa\ xb))\ \text{of}$   
 $\quad \quad \text{None} \Rightarrow \text{let } (x, y) = H2\ \text{in } (x, y) \mid \text{Some } p \Rightarrow (\text{Suc } x, p)))$

*pigeonhole-slow*  $\equiv$

$\lambda x. \text{nat-induct-}P\ x\ (\lambda x. (\text{Suc } 0, 0))$   
 $(\lambda x\ H2\ xa.$   
 $\quad \text{case search } (\text{Suc } (\text{Suc } x))$   
 $\quad \quad (\lambda xb. \text{nat-eq-dec } (xa\ (\text{Suc } (\text{Suc } x)))\ (xa\ xb))\ \text{of}$   
 $\quad \text{None} \Rightarrow$   
 $\quad \quad \text{let } (x, y) =$   
 $\quad \quad \quad H2\ (\lambda i. \text{if } xa\ i = \text{Suc } x \text{ then } xa\ (\text{Suc } (\text{Suc } x)) \text{ else } xa\ i)$   
 $\quad \quad \text{in } (x, y)$   
 $\quad \mid \text{Some } p \Rightarrow (\text{Suc } (\text{Suc } x), p))$

The program for searching for an element in an array is

*search*  $\equiv$

$\lambda x\ H. \text{nat-induct-}P\ x\ \text{None}$   
 $(\lambda y\ Ha.$   
 $\quad \text{case } Ha\ \text{of } \text{None} \Rightarrow \text{case } H\ y\ \text{of } \text{Left} \Rightarrow \text{Some } y \mid \text{Right} \Rightarrow \text{None}$   
 $\quad \mid \text{Some } p \Rightarrow \text{Some } p)$

The correctness statement for *pigeonhole* is

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

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**  $\langle \text{proof} \rangle$

**end**

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

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

**instance**  $\langle \text{proof} \rangle$

**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])

$\langle \text{ML} \rangle$

**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*:  $n = m * k \implies m < n \implies k < n \implies \text{Suc } 0 < m$   
 $\langle \text{proof} \rangle$

**lemma** *factor-greater-one2*:  $n = m * k \implies m < n \implies k < n \implies \text{Suc } 0 < k$   
 ⟨proof⟩

**lemma** *prod-mn-less-k*:  $0 < n \implies 0 < k \implies \text{Suc } 0 < m \implies m * n = k \implies n < k$   
 ⟨proof⟩

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

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

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

**lemma** *dvd-factorial*:  $0 < m \implies m \leq n \implies m \text{ dvd fact } n$   
 ⟨proof⟩

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

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

**lemma** *all-prime-simps*:  
 $\text{all-prime } []$   
 $\text{all-prime } (p \# ps) \longleftrightarrow \text{prime } p \wedge \text{all-prime } ps$   
 ⟨proof⟩

**lemma** *all-prime-append*:  $\text{all-prime } (ps @ qs) \longleftrightarrow \text{all-prime } ps \wedge \text{all-prime } qs$   
 ⟨proof⟩

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

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

$\langle proof \rangle$

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

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

Euclid's theorem: there are infinitely many primes.

**lemma** *Euclid*:  $\exists p::nat. \text{prime } p \wedge n < p$   
 $\langle proof \rangle$

**extract** *Euclid*

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

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

The program corresponding to the proof of the factorization theorem is

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

**instantiation**  $nat :: default$   
**begin**

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

**instance**  $\langle proof \rangle$

**end**

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

**definition**  $default = []$

**instance**  $\langle proof \rangle$

**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.