Examples for program extraction in Higher-Order Logic

Stefan Berghofer

April 19, 2009

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

The corresponding correctness theorem is

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

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

instantiation *nat* :: *default*
begin

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

instance ⟨*proof*⟩

end

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

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

instance ⟨*proof*⟩

end

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

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

instance ⟨*proof*⟩

end

consts-code
default ((*error default*))

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

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* | *F*

primrec
is-path' :: ('*a* ⇒ '*a* ⇒ *b*) ⇒ '*a* ⇒ '*a list* ⇒ '*a* ⇒ *bool*
where
is-path' r x [] *z* = (*r x z* = *T*)
| *is-path' r x* (*y* # *ys*) *z* = (*r x y* = *T* ∧ *is-path' r y ys z*)

definition
is-path :: (*nat* ⇒ *nat* ⇒ *b*) ⇒ (*nat* ∗ *nat list* ∗ *nat*) ⇒
nat ⇒ *nat* ⇒ *nat* ⇒ *bool*
where
is-path r p i j k ⟷ *fst p* = *j* ∧ *snd* (*snd p*) = *k* ∧
list-all (λ*x*. *x* < *i*) (*fst* (*snd p*)) ∧
is-path' r (*fst p*) (*fst* (*snd p*)) (*snd* (*snd p*))

definition
conc :: ('*a* ∗ '*a list* ∗ '*a*) ⇒ ('*a* ∗ '*a list* ∗ '*a*) ⇒ ('*a* ∗ '*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*]:
$\bigwedge x.\ \textit{is-path}'\ r\ x\ (ys\ @\ [y])\ z = (\textit{is-path}'\ r\ x\ ys\ y \wedge r\ y\ z = T)$
⟨*proof*⟩

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

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

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

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

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

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

theorem *lemma5*:
$\bigwedge p.\ \textit{is-path}\ r\ p\ (\textit{Suc}\ i)\ j\ k \Longrightarrow \sim \textit{is-path}\ r\ p\ i\ j\ k \Longrightarrow$
$(\exists q.\ \textit{is-path}\ r\ q\ i\ j\ i) \wedge (\exists q'.\ \textit{is-path}\ r\ q'\ i\ i\ k)$
⟨*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 ≡
λ*x xa xb xc*.
nat-induct-P xa
(λ*xa xb*. *case x xa xb of T* ⇒ *Some* (*xa*, [], *xb*) | *F* ⇒ *None*)
(λ*x 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 Combinator syntax for generic, open state monads (single threaded monads)

theory *State-Monad*
imports *Main*
begin

5.1 Motivation

The logic HOL has no notion of constructor classes, so it is not possible to model monads the Haskell way in full genericity in Isabelle/HOL.

However, this theory provides substantial support for a very common class of monads: *state monads* (or *single-threaded monads*, since a state is transformed single-threaded).

To enter from the Haskell world, http://www.engr.mun.ca/~theo/Misc/haskell_and_monads.htm makes a good motivating start. Here we just sketch briefly how those monads enter the game of Isabelle/HOL.

5.2 State transformations and combinators

We classify functions operating on states into two categories:

transformations with type signature $\sigma \Rightarrow \sigma'$, transforming a state.

"yielding" transformations with type signature $\sigma \Rightarrow \alpha \times \sigma'$, "yielding" a side result while transforming a state.

queries with type signature $\sigma \Rightarrow \alpha$, computing a result dependent on a state.

By convention we write σ for types representing states and α, β, γ, ... for types representing side results. Type changes due to transformations are not excluded in our scenario.

We aim to assert that values of any state type σ are used in a single-threaded way: after application of a transformation on a value of type σ, the former value should not be used again. To achieve this, we use a set of monad combinators:

notation *fcomp* (**infixl** *o>* *60*)
notation (*xsymbols*) *fcomp* (**infixl** *o>* *60*)
notation *scomp* (**infixl** *o–>* *60*)
notation (*xsymbols*) *scomp* (**infixl** *o→* *60*)

abbreviation (*input*)
return ≡ *Pair*

Given two transformations f and g, they may be directly composed using the *op o>* combinator, forming a forward composition: $(f\ o{>}\ g)\ s = f\ (g\ s)$.

After any yielding transformation, we bind the side result immediately using a lambda abstraction. This is the purpose of the *op o→* combinator: $(f\ o{\rightarrow}\ (\lambda x.\ g))\ s = (let\ (x,\ s') = f\ s\ in\ g\ s')$.

For queries, the existing *Let* is appropriate.

Naturally, a computation may yield a side result by pairing it to the state from the left; we introduce the suggestive abbreviation *Pair* for this purpose.

The most crucial distinction to Haskell is that we do not need to introduce distinguished type constructors for different kinds of state. This has two consequences:

- The monad model does not state anything about the kind of state; the model for the state is completely orthogonal and may be specified completely independently.
- There is no distinguished type constructor encapsulating away the state transformation, i.e. transformations may be applied directly without using any lifting or providing and dropping units ("open monad").
- The type of states may change due to a transformation.

5.3 Monad laws

The common monadic laws hold and may also be used as normalization rules for monadic expressions:

lemmas *monad-simp* = *Pair-scomp scomp-Pair id-fcomp fcomp-id*
scomp-scomp scomp-fcomp fcomp-scomp fcomp-assoc

Evaluation of monadic expressions by force:

lemmas *monad-collapse = monad-simp fcomp-apply scomp-apply split-beta*

5.4 Syntax

We provide a convenient do-notation for monadic expressions well-known from Haskell. *Let* is printed specially in do-expressions.

nonterminals *do-expr*

syntax
-do :: do-expr ⇒ 'a
(do - done [12] 12)
-scomp :: pttrn ⇒ 'a ⇒ do-expr ⇒ do-expr
(- <− -;// - [1000, 13, 12] 12)
-fcomp :: 'a ⇒ do-expr ⇒ do-expr
(-;// - [13, 12] 12)
-let :: pttrn ⇒ 'a ⇒ do-expr ⇒ do-expr
(let - = -;// - [1000, 13, 12] 12)
-done :: 'a ⇒ do-expr
(- [12] 12)

syntax (*xsymbols*)
-scomp :: pttrn ⇒ 'a ⇒ do-expr ⇒ do-expr
(- ← -;// - [1000, 13, 12] 12)

translations
-do f => f
-scomp x f g => f o→ (λx. g)
-fcomp f g => f o> g
-let x t f => CONST Let t (λx. f)
-done f => f

⟨*ML*⟩

For an example, see HOL/ex/Random.thy.

end

6 A HOL random engine

theory *Random*
imports *Code-Index*
begin

notation *fcomp* (**infixl** *o> 60*)
notation *scomp* (**infixl** *o→ 60*)

6.1 Auxiliary functions

definition *inc-shift* :: *index* ⇒ *index* ⇒ *index* **where**
inc-shift v k = (*if v* = *k then 1 else k* + *1*)

definition *minus-shift* :: *index* ⇒ *index* ⇒ *index* ⇒ *index* **where**
minus-shift r k l = (*if k* < *l then r* + *k* − *l else k* − *l*)

fun *log* :: *index* ⇒ *index* ⇒ *index* **where**
log b i = (*if b* ≤ *1* ∨ *i* < *b then 1 else 1* + *log b* (*i div b*))

6.2 Random seeds

types *seed* = *index* × *index*

primrec *next* :: *seed* ⇒ *index* × *seed* **where**
next (*v*, *w*) = (*let*
k = *v div 53668*;
v' = *minus-shift 2147483563* (*40014* ∗ (*v mod 53668*)) (*k* ∗ *12211*);
l = *w div 52774*;
w' = *minus-shift 2147483399* (*40692* ∗ (*w mod 52774*)) (*l* ∗ *3791*);
z = *minus-shift 2147483562 v'* (*w'* + *1*) + *1*
in (*z*, (*v'*, *w'*)))

lemma *next-not-0*:
fst (*next s*) ≠ *0*
⟨*proof*⟩

primrec *seed-invariant* :: *seed* ⇒ *bool* **where**
seed-invariant (*v*, *w*) ⟷ *0* < *v* ∧ *v* < *9438322952* ∧ *0* < *w* ∧ *True*

lemma *if-same*: (*if b then f x else f y*) = *f* (*if b then x else y*)
⟨*proof*⟩

definition *split-seed* :: *seed* ⇒ *seed* × *seed* **where**
split-seed s = (*let*
(*v*, *w*) = *s*;
(*v'*, *w'*) = *snd* (*next s*);
v'' = *inc-shift 2147483562 v*;
s'' = (*v''*, *w'*);
w'' = *inc-shift 2147483398 w*;
s''' = (*v'*, *w''*)
in (*s''*, *s'''*))

6.3 Base selectors

fun *iterate* :: *index* ⇒ ('*b* ⇒ '*a* ⇒ '*b* × '*a*) ⇒ '*b* ⇒ '*a* ⇒ '*b* × '*a* **where**
iterate k f x = (*if k* = *0 then Pair x else f x* o→ *iterate* (*k* − *1*) *f*)

definition *range* :: *index* ⇒ *seed* ⇒ *index* × *seed* **where**

```
  range k = iterate (log 2147483561 k)
      (λl. next o→ (λv. Pair (v + l * 2147483561))) 1
    o→ (λv. Pair (v mod k))

lemma range:
  k > 0 ⟹ fst (range k s) < k
  ⟨proof⟩

definition select :: 'a list ⇒ seed ⇒ 'a × seed where
  select xs = range (Code-Index.of-nat (length xs))
    o→ (λk. Pair (nth xs (Code-Index.nat-of k)))

lemma select:
  assumes xs ≠ []
  shows fst (select xs s) ∈ set xs
⟨proof⟩

definition select-default :: index ⇒ 'a ⇒ 'a ⇒ seed ⇒ 'a × seed where
  [code del]: select-default k x y = range k
      o→ (λl. Pair (if l + 1 < k then x else y))

lemma select-default-zero:
  fst (select-default 0 x y s) = y
  ⟨proof⟩

lemma select-default-code [code]:
  select-default k x y = (if k = 0
    then range 1 o→ (λ-. Pair y)
    else range k o→ (λl. Pair (if l + 1 < k then x else y)))
⟨proof⟩
```

6.4 *ML* interface

```
⟨ML⟩

no-notation fcomp (infixl o> 60)
no-notation scomp (infixl o→ 60)

end
```

7 Higman's lemma

```
theory Higman
imports Main State-Monad Random
begin
```

Formalization by Stefan Berghofer and Monika Seisenberger, based on Co-

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

primrec
is-prefix :: *'a list* ⇒ (*nat* ⇒ *'a*) ⇒ *bool*
where
is-prefix [] *f* = *True*
| *is-prefix* (*x* # *xs*) *f* = (*x* = *f* (*length xs*) ∧ *is-prefix xs f*)

theorem *L-idx*:
assumes *L*: *L w ws*
shows *is-prefix ws f* ⟹ ∃ *i*. *emb* (*f i*) *w* ∧ *i* < *length ws* ⟨*proof*⟩

theorem *good-idx*:
assumes *good*: *good ws*
shows *is-prefix ws f* ⟹ ∃ *i j*. *emb* (*f i*) (*f j*) ∧ *i* < *j* ⟨*proof*⟩

theorem *bar-idx*:
assumes *bar*: *bar ws*
shows *is-prefix ws f* ⟹ ∃ *i j*. *emb* (*f i*) (*f j*) ∧ *i* < *j* ⟨*proof*⟩

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

theorem *higman-idx*: ∃ (*i*::*nat*) *j*. *emb* (*f i*) (*f j*) ∧ *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*⟩

7.1 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 $\equiv$ λx. *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 [] (*list-rec* (*prop1* []) (λa *w H*. *prop3 a* [*a* # *w*] *H* (*R1* [] [] *w R0*)))

Program extracted from the proof of *prop1*:

prop1 $\equiv$
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
```

7.2 Some examples

```
instantiation LT and TT :: default
begin

definition default = L0 [] []

definition default = T0 A [] [] [] R0

instance ⟨proof⟩

end

function mk-word-aux :: nat ⇒ seed ⇒ letter list × seed where
  mk-word-aux k = (do
     i ← range 10;
     (if i > 7 ∧ k > 2 ∨ k > 1000 then return []
     else do
       let l = (if i mod 2 = 0 then A else B);
       ls ← mk-word-aux (Suc k);
       return (l # ls)
     done)
   done)
⟨proof⟩
```

termination ⟨*proof*⟩

definition *mk-word* :: *seed* ⇒ *letter list* × *seed* **where**
mk-word = *mk-word-aux 0*

primrec *mk-word-s* :: *nat* ⇒ *seed* ⇒ *letter list* × *seed* **where**
mk-word-s 0 = *mk-word*
| *mk-word-s* (*Suc n*) = (*do*
- ← *mk-word*;
mk-word-s n
done)

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*⟩

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

⟨*ML*⟩

end

8 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*⟩

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

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

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 ∗ :: (*default*, *default*) *default*
begin

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

instance ⟨*proof*⟩

end

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

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

code-module *PH*
contains
test = *test*
test′ = *test*′
test″ = *test*″

⟨*ML*⟩

end

9 Euclid's theorem

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

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

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

primrec *fact* :: *nat* ⇒ *nat* ((-!) [*1000*] *999*)
where
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 ≡ λ*x*. *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))

instantiation nat :: default
begin

definition default = (0::nat)

instance ⟨proof⟩

end

instantiation list :: (type) default
begin

definition default = []

instance ⟨proof⟩

end

consts-code
  default ((error default))

lemma factor-exists 1007 = [53, 19] ⟨proof⟩
lemma factor-exists 1007 = [53, 19] ⟨proof⟩

lemma factor-exists 567 = [7, 3, 3, 3, 3] ⟨proof⟩
lemma factor-exists 567 = [7, 3, 3, 3, 3] ⟨proof⟩

lemma factor-exists 345 = [23, 5, 3] ⟨proof⟩
lemma factor-exists 345 = [23, 5, 3] ⟨proof⟩

lemma factor-exists 999 = [37, 3, 3, 3] ⟨proof⟩
lemma factor-exists 999 = [37, 3, 3, 3] ⟨proof⟩

lemma factor-exists 876 = [73, 3, 2, 2] ⟨proof⟩
lemma factor-exists 876 = [73, 3, 2, 2] ⟨proof⟩

primrec iterate :: nat ⇒ ('a ⇒ 'a) ⇒ 'a ⇒ 'a list where
  iterate 0 f x = []
  | iterate (Suc n) f x = (let y = f x in y # iterate n f y)
```

lemma *iterate 4 Euclid 0 = [2, 3, 7, 71]* ⟨*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.