Examples for program extraction in Higher-Order Logic

Stefan Berghofer

October 1, 2005

Contents

1 Quotient and remainder

theory *QuotRem* **imports** *Main* **begin**

Derivation of quotient and remainder using program extraction.

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

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

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

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*

end

2 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
  $is\text{-}path :: (nat \Rightarrow nat \Rightarrow b) \Rightarrow (nat * nat\ list * nat) \Rightarrow$
    $nat \Rightarrow nat \Rightarrow nat \Rightarrow bool$
  $is\text{-}path\ r\ p\ i\ j\ k == fst\ p = j \wedge snd\ (snd\ p) = k \wedge$
    $list\text{-}all\ (\lambda x.\ x < i)\ (fst\ (snd\ p)) \wedge$
    $is\text{-}path'\ r\ (fst\ p)\ (fst\ (snd\ p))\ (snd\ (snd\ p))$

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

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

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

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

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

$\langle proof \rangle$

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

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

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

theorem *lemma5*:
$\bigwedge p.\ is\text{-}path\ r\ p\ (Suc\ i)\ j\ k \Longrightarrow {}^{\sim}\ is\text{-}path\ r\ p\ i\ j\ k \Longrightarrow$
$(\exists\, q.\ is\text{-}path\ r\ q\ i\ j\ i) \wedge (\exists\, q'.\ is\text{-}path\ r\ q'\ i\ i\ k)$
$\langle proof \rangle$

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

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

3 Higman's lemma

theory *Higman* **imports** *Main* **begin**

Formalization by Stefan Berghofer and Monika Seisenberger, based on Coquand and Fridlender [2].

datatype $letter = A \mid B$

consts
$emb :: (letter\ list \times letter\ list)\ set$

inductive *emb*
intros
$emb0\ [Pure.intro]: ([], bs) \in emb$
$emb1\ [Pure.intro]: (as, bs) \in emb \Longrightarrow (as, b \# bs) \in emb$
$emb2\ [Pure.intro]: (as, bs) \in emb \Longrightarrow (a \# as, a \# bs) \in emb$

consts
$L :: letter\ list \Rightarrow letter\ list\ list\ set$

inductive $L\ v$
intros
$L0\ [Pure.intro]: (w, v) \in emb \Longrightarrow w \# ws \in L\ v$
$L1\ [Pure.intro]: ws \in L\ v \Longrightarrow w \# ws \in L\ v$

consts
$good :: letter\ list\ list\ set$

inductive *good*
intros
$good0\ [Pure.intro]: ws \in L\ w \Longrightarrow w \# ws \in good$
$good1\ [Pure.intro]: ws \in good \Longrightarrow w \# ws \in good$

consts
$R :: letter \Rightarrow (letter\ list\ list \times letter\ list\ list)\ set$

inductive $R\ a$
intros
$R0\ [Pure.intro]: ([], []) \in R\ a$
$R1\ [Pure.intro]: (vs, ws) \in R\ a \Longrightarrow (w \# vs, (a \# w) \# ws) \in R\ a$

consts
$T :: letter \Rightarrow (letter\ list\ list \times letter\ list\ list)\ set$

inductive $T\ a$
intros
$T0\ [Pure.intro]: a \neq b \Longrightarrow (ws, zs) \in R\ b \Longrightarrow (w \# zs, (a \# w) \# zs) \in T\ a$
$T1\ [Pure.intro]: (ws, zs) \in T\ a \Longrightarrow (w \# ws, (a \# w) \# zs) \in T\ a$
$T2\ [Pure.intro]: a \neq b \Longrightarrow (ws, zs) \in T\ a \Longrightarrow (ws, (b \# w) \# zs) \in T\ a$

consts
  bar :: *letter list list set*

inductive *bar*
intros
  bar1 [*Pure.intro*]: $ws \in good \Longrightarrow ws \in bar$
  bar2 [*Pure.intro*]: $(\bigwedge w.\ w \# ws \in bar) \Longrightarrow ws \in bar$

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

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

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

lemma *lemma2*: $(vs, ws) \in R\ a \Longrightarrow vs \in good \Longrightarrow ws \in good$
  ⟨*proof*⟩

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

lemma *lemma3*: $(ws, zs) \in T\ a \Longrightarrow ws \in good \Longrightarrow zs \in good$
  ⟨*proof*⟩

lemma *lemma4*: $(ws, zs) \in R\ a \Longrightarrow ws \neq [] \Longrightarrow (ws, zs) \in T\ a$
  ⟨*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*: $xs \in bar$
  shows $\bigwedge ys\ zs.\ ys \in bar \Longrightarrow (xs, zs) \in T\ a \Longrightarrow (ys, zs) \in T\ b \Longrightarrow zs \in bar$
⟨*proof*⟩

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

theorem *higman*: $[] \in bar$
⟨*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-prefix* xs $f)$

theorem *good-prefix-lemma*:
assumes *bar*: $ws \in bar$
shows *is-prefix* ws $f \Longrightarrow \exists vs.$ *is-prefix* vs $f \wedge vs \in good$ ⟨*proof*⟩

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

3.1 Extracting the program

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

extract *good-prefix*

Program extracted from the proof of *good-prefix*:

good-prefix $\equiv \lambda x.$ *good-prefix-lemma* x *higman*

Corresponding correctness theorem:

is-prefix (*good-prefix* f) $f \wedge$ *good-prefix* $f \in good$

Program extracted from the proof of *good-prefix-lemma*:

good-prefix-lemma $\equiv \lambda x.$ *barT-rec* $(\lambda ws.\ ws)$ $(\lambda ws\ xa\ r.\ r\ (x\ (length\ ws)))$

Program extracted from the proof of *higman*:

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

Program extracted from the proof of *prop1*:

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

Program extracted from the proof of *prop2*:

```
prop2 ≡
λx xa xb xc H.
   barT-rec (λws x xa H. bar1 xa)
    (λws xb r xc xd H.
        barT-rec (λws. bar1)
         (λws xb ra xc.
             bar2 xc
              (list-case (prop1 xc)
                (λa list.
                    case letter-eq-dec a x of
```

Left ⇒ *r list ws* ((*x* # *list*) # *xc*) (*bar2 ws xb*)
| *Right* ⇒ *ra list* ((*xa* # *list*) # *xc*))))
H xd)
H xb xc

Program extracted from the proof of *prop3*:

prop3 ≡
λ*x xa H*.
barT-rec (λ*ws*. *bar1*)
(λ*ws x r xb*.
bar2 xb
(*list-rec* (*prop1 xb*)
(λ*a w-- H*.
case letter-eq-dec a xa of Left ⇒ *r w--* ((*xa* # *w--*) # *xb*)
| *Right* ⇒ *prop2 a xa ws* ((*a* # *w--*) # *xb*) *H* (*bar2 ws x*))))
H x

code-module *Higman*
contains
test = *good-prefix*

⟨*ML*⟩

end

4 The pigeonhole principle

theory *Pigeonhole* **imports** *EfficientNat* **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].

We need decidability of equality on natural numbers:

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

We can decide whether an array f of length l contains an element x.

lemma *search*: $(\exists j{<}(l{::}nat).\ (x{::}nat) = f\,j) \vee \neg\,(\exists j{<}l.\ x = f\,j)$
⟨*proof*⟩

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

extract *pigeonhole pigeonhole-slow*

The programs extracted from the above proofs look as follows:

$$
\begin{array}{l}
pigeonhole \equiv \\
nat\text{-}rec\ (\lambda x.\ (Suc\ 0,\ 0)) \\
\ (\lambda x\ H2\ xa. \\
\quad nat\text{-}rec\ arbitrary \\
\quad\ (\lambda x\ H2. \\
\qquad case\ search\ (Suc\ x)\ (xa\ (Suc\ x))\ xa\ of \\
\qquad None \Rightarrow let\ (x,\ y) = H2\ in\ (x,\ y) \mid Some\ p \Rightarrow (Suc\ x,\ p)) \\
\quad\ (Suc\ (Suc\ x)))
\end{array}
$$

$$
\begin{array}{l}
pigeonhole\text{-}slow \equiv \\
nat\text{-}rec\ (\lambda x.\ (Suc\ 0,\ 0)) \\
\ (\lambda x\ H2\ xa. \\
\quad case\ search\ (Suc\ (Suc\ x))\ (xa\ (Suc\ (Suc\ x)))\ xa\ of \\
\quad None \Rightarrow \\
\quad\ \ let\ (x,\ y) = H2\ (\lambda i.\ if\ xa\ i = Suc\ x\ then\ xa\ (Suc\ (Suc\ x))\ else\ xa\ i) \\
\quad\ \ in\ (x,\ y) \\
\quad \mid Some\ p \Rightarrow (Suc\ (Suc\ x),\ p))
\end{array}
$$

The program for searching for an element in an array is

$$
\begin{array}{l}
search \equiv \\
\lambda x\ xa\ xb. \\
\quad nat\text{-}rec\ None \\
\quad\ (\lambda l\ H.\ case\ H\ of \\
\qquad\quad None \Rightarrow case\ nat\text{-}eq\text{-}dec\ xa\ (xb\ l)\ of\ Left \Rightarrow Some\ l \mid Right \Rightarrow None \\
\qquad\quad \mid Some\ p \Rightarrow Some\ p) \\
\quad\ x
\end{array}
$$

The correctness statement for *pigeonhole* is

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

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

consts-code

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

code-module *PH*
contains
$test = \lambda n.\ pigeonhole\ n\ (\lambda m.\ m - 1)$
$test' = \lambda n.\ pigeonhole\text{-}slow\ n\ (\lambda m.\ m - 1)$
$sel = op\ !$

⟨*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.