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: ⋀n::nat. m = n ∨ m ≠ n
  apply (induct m)
  apply (case-tac n)
  apply (case-tac [3] n)
  apply (simp only: nat.simps, iprover?)+
  done

theorem division: ∃ r q. a = Suc b * q + r ∧ r ≤ b
proof (induct a)
  case 0
  have 0 = Suc b * 0 + 0 ∧ 0 ≤ b by simp
  thus ?case by iprover
next
  case (Suc a)
  then obtain r q where I: a = Suc b * q + r and r ≤ b by iprover
  from nat-eq-dec show ?case
  proof
    assume r = b
```

with I **have** $Suc\ a = Suc\ b * (Suc\ q) + 0 \wedge 0 \leq b$ **by** *simp*
thus *?case* **by** *iprover*
next
assume $r \neq b$
hence $r < b$ **by** (*simp add*: *order-less-le*)
with I **have** $Suc\ a = Suc\ b * q + (Suc\ r) \wedge (Suc\ r) \leq b$ **by** *simp*
thus *?case* **by** *iprover*
qed
qed

extract *division*

The program extracted from the above proof looks as follows

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

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-path' :: $('a \Rightarrow 'a \Rightarrow b) \Rightarrow 'a \Rightarrow 'a\ list \Rightarrow 'a \Rightarrow bool$

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

```
constdefs
  is-path :: (nat ⇒ nat ⇒ b) ⇒ (nat ∗ nat list ∗ nat) ⇒
    nat ⇒ nat ⇒ nat ⇒ bool
  is-path r p i j k == fst p = j ∧ snd (snd p) = k ∧
     list-all (λx. x < i) (fst (snd p)) ∧
     is-path′ r (fst p) (fst (snd p)) (snd (snd p))

  conc :: ('a ∗ 'a list ∗ 'a) ⇒ ('a ∗ 'a list ∗ 'a) ⇒ ('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]:
  ⋀x. is-path′ r x (ys @ [y]) z = (is-path′ r x ys y ∧ r y z = T)
  by (induct ys) simp+

theorem list-all-scoc [simp]: list-all P (xs @ [x]) = (P x ∧ list-all P xs)
  by (induct xs, simp+, iprover)

theorem list-all-lemma:
  list-all P xs ⟹ (⋀x. P x ⟹ Q x) ⟹ list-all Q xs
proof −
  assume PQ: ⋀x. P x ⟹ Q x
  show list-all P xs ⟹ list-all Q xs
  proof (induct xs)
    case Nil
    show ?case by simp
  next
    case (Cons y ys)
    hence Py: P y by simp
    from Cons have Pys: list-all P ys by simp
    show ?case
      by simp (rule conjI PQ Py Cons Pys)+
  qed
qed

theorem lemma1: ⋀p. is-path r p i j k ⟹ is-path r p (Suc i) j k
  apply (unfold is-path-def)
  apply (simp cong add: conj-cong add: split-paired-all)
  apply (erule conjE)+
  apply (erule list-all-lemma)
  apply simp
  done

theorem lemma2: ⋀p. is-path r p 0 j k ⟹ r j k = T
  apply (unfold is-path-def)
  apply (simp cong add: conj-cong add: split-paired-all)
  apply (case-tac aa)
  apply simp+
  done
```

```
theorem is-path'-conc: is-path' r j xs i ⟹ is-path' r i ys k ⟹
  is-path' r j (xs @ i # ys) k
proof −
  assume pys: is-path' r i ys k
  show ⋀j. is-path' r j xs i ⟹ is-path' r j (xs @ i # ys) k
  proof (induct xs)
    case (Nil j)
    hence r j i = T by simp
    with pys show ?case by simp
  next
    case (Cons z zs j)
    hence jzr: r j z = T by simp
    from Cons have pzs: is-path' r z zs i by simp
    show ?case
      by simp (rule conjI jzr Cons pzs)+
  qed
qed

theorem lemma3:
  ⋀p q. is-path r p i j i ⟹ is-path r q i i k ⟹
   is-path r (conc p q) (Suc i) j k
  apply (unfold is-path-def conc-def)
  apply (simp cong add: conj-cong add: split-paired-all)
  apply (erule conjE)+
  apply (rule conjI)
  apply (erule list-all-lemma)
  apply simp
  apply (rule conjI)
  apply (erule list-all-lemma)
  apply simp
  apply (rule is-path'-conc)
  apply assumption+
  done

theorem lemma5:
  ⋀p. is-path r p (Suc i) j k ⟹ ~ is-path r p i j k ⟹
   (∃ q. is-path r q i j i) ∧ (∃ q'. is-path r q' i i k)
proof (simp cong add: conj-cong add: split-paired-all is-path-def, (erule conjE)+)
  fix xs
  assume list-all (λx. x < Suc i) xs
  assume is-path' r j xs k
  assume ¬ list-all (λx. x < i) xs
  show (∃ ys. list-all (λx. x < i) ys ∧ is-path' r j ys i) ∧
    (∃ ys. list-all (λx. x < i) ys ∧ is-path' r i ys k)
  proof
    show ⋀j. list-all (λx. x < Suc i) xs ⟹ is-path' r j xs k ⟹
      ¬ list-all (λx. x < i) xs ⟹
    ∃ ys. list-all (λx. x < i) ys ∧ is-path' r j ys i (is PROP ?ih xs)
```

```
proof (induct xs)
  case Nil
  thus ?case by simp
next
  case (Cons a as j)
  show ?case
  proof (cases a=i)
    case True
    show ?thesis
    proof
      from True and Cons have r j i = T by simp
      thus list-all (λx. x < i) [] ∧ is-path' r j [] i by simp
    qed
  next
    case False
    have PROP ?ih as by (rule Cons)
    then obtain ys where ys: list-all (λx. x < i) ys ∧ is-path' r a ys i
    proof
      from Cons show list-all (λx. x < Suc i) as by simp
      from Cons show is-path' r a as k by simp
      from Cons and False show ¬ list-all (λx. x < i) as by (simp)
    qed
    show ?thesis
    proof
      from Cons False ys
      show list-all (λx. x<i) (a#ys) ∧ is-path' r j (a#ys) i by simp
    qed
  qed
qed
show ⋀k. list-all (λx. x < Suc i) xs ⟹ is-path' r j xs k ⟹
  ¬ list-all (λx. x < i) xs ⟹
  ∃ys. list-all (λx. x < i) ys ∧ is-path' r i ys k (is PROP ?ih xs)
proof (induct xs rule: rev-induct)
  case Nil
  thus ?case by simp
next
  case (snoc a as k)
  show ?case
  proof (cases a=i)
    case True
    show ?thesis
    proof
      from True and snoc have r i k = T by simp
      thus list-all (λx. x < i) [] ∧ is-path' r i [] k by simp
    qed
  next
    case False
    have PROP ?ih as by (rule snoc)
    then obtain ys where ys: list-all (λx. x < i) ys ∧ is-path' r i ys a
```

```
        proof
          from snoc show list-all (λx. x < Suc i) as by simp
          from snoc show is-path' r j as a by simp
          from snoc and False show ¬ list-all (λx. x < i) as by simp
        qed
        show ?thesis
        proof
          from snoc False ys
          show list-all (λx. x < i) (ys @ [a]) ∧ is-path' r i (ys @ [a]) k
            by simp
        qed
      qed
    qed
  qed
qed

theorem lemma5':
  ⋀p. is-path r p (Suc i) j k ⟹ ¬ is-path r p i j k ⟹
   ¬ (∀ q. ¬ is-path r q i j i) ∧ ¬ (∀ q'. ¬ is-path r q' i i k)
  by (iprover dest: lemma5)

theorem warshall:
  ⋀j k. ¬ (∃ p. is-path r p i j k) ∨ (∃ p. is-path r p i j k)
proof (induct i)
  case (0 j k)
  show ?case
  proof (cases r j k)
    assume r j k = T
    hence is-path r (j, [], k) 0 j k
      by (simp add: is-path-def)
    hence ∃ p. is-path r p 0 j k ..
    thus ?thesis ..
  next
    assume r j k = F
    hence r j k ~= T by simp
    hence ¬ (∃ p. is-path r p 0 j k)
      by (iprover dest: lemma2)
    thus ?thesis ..
  qed
next
  case (Suc i j k)
  thus ?case
  proof
    assume h1: ¬ (∃ p. is-path r p i j k)
    from Suc show ?case
    proof
      assume ¬ (∃ p. is-path r p i j i)
      with h1 have ¬ (∃ p. is-path r p (Suc i) j k)
        by (iprover dest: lemma5')
```

```
      thus ?case ..
    next
      assume ∃ p. is-path r p i j i
      then obtain p where h2: is-path r p i j i ..
      from Suc show ?case
      proof
        assume ¬ (∃ p. is-path r p i i k)
        with h1 have ¬ (∃ p. is-path r p (Suc i) j k)
          by (iprover dest: lemma5')
        thus ?case ..
      next
        assume ∃ q. is-path r q i i k
        then obtain q where is-path r q i i k ..
        with h2 have is-path r (conc p q) (Suc i) j k
          by (rule lemma3)
        hence ∃ pq. is-path r pq (Suc i) j k ..
        thus ?case ..
      qed
    qed
  next
    assume ∃ p. is-path r p i j k
    hence ∃ p. is-path r p (Suc i) j k
      by (iprover intro: lemma1)
    thus ?case ..
  qed
qed
```

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* | *B*

consts
  emb :: (*letter list* × *letter list*) *set*

inductive *emb*
intros
  emb0 [*Pure.intro*]: ([], *bs*) ∈ *emb*
  emb1 [*Pure.intro*]: (*as*, *bs*) ∈ *emb* ⟹ (*as*, *b* # *bs*) ∈ *emb*
  emb2 [*Pure.intro*]: (*as*, *bs*) ∈ *emb* ⟹ (*a* # *as*, *a* # *bs*) ∈ *emb*

consts
  L :: *letter list* ⇒ *letter list list set*

inductive *L v*
intros
  L0 [*Pure.intro*]: (*w*, *v*) ∈ *emb* ⟹ *w* # *ws* ∈ *L v*
  L1 [*Pure.intro*]: *ws* ∈ *L v* ⟹ *w* # *ws* ∈ *L v*

consts
  good :: *letter list list set*

inductive *good*
intros
  good0 [*Pure.intro*]: *ws* ∈ *L w* ⟹ *w* # *ws* ∈ *good*
  good1 [*Pure.intro*]: *ws* ∈ *good* ⟹ *w* # *ws* ∈ *good*

consts
  R :: *letter* ⇒ (*letter list list* × *letter list list*) *set*

inductive *R a*
intros
  R0 [*Pure.intro*]: ([], []) ∈ *R a*
  R1 [*Pure.intro*]: (*vs*, *ws*) ∈ *R a* ⟹ (*w* # *vs*, (*a* # *w*) # *ws*) ∈ *R a*

consts
  T :: *letter* ⇒ (*letter list list* × *letter list list*) *set*

inductive *T a*
intros
  T0 [*Pure.intro*]: *a* ≠ *b* ⟹ (*ws*, *zs*) ∈ *R b* ⟹ (*w* # *zs*, (*a* # *w*) # *zs*) ∈ *T a*
  T1 [*Pure.intro*]: (*ws*, *zs*) ∈ *T a* ⟹ (*w* # *ws*, (*a* # *w*) # *zs*) ∈ *T a*
  T2 [*Pure.intro*]: *a* ≠ *b* ⟹ (*ws*, *zs*) ∈ *T a* ⟹ (*ws*, (*b* # *w*) # *zs*) ∈ *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*: $([] \# ws) \in bar$ **by** *iprover*

theorem *lemma1*: $ws \in L\ as \Longrightarrow ws \in L\ (a \# as)$
  by (*erule L.induct*, *iprover+*)

lemma *lemma2'*: $(vs, ws) \in R\ a \Longrightarrow vs \in L\ as \Longrightarrow ws \in L\ (a \# as)$
  apply (*induct set*: *R*)
  apply (*erule L.elims*)
  apply *simp+*
  apply (*erule L.elims*)
  apply *simp-all*
  apply (*rule L0*)
  apply (*erule emb2*)
  apply (*erule L1*)
  done

lemma *lemma2*: $(vs, ws) \in R\ a \Longrightarrow vs \in good \Longrightarrow ws \in good$
  apply (*induct set*: *R*)
  apply *iprover*
  apply (*erule good.elims*)
  apply *simp-all*
  apply (*rule good0*)
  apply (*erule lemma2'*)
  apply *assumption*
  apply (*erule good1*)
  done

lemma *lemma3'*: $(vs, ws) \in T\ a \Longrightarrow vs \in L\ as \Longrightarrow ws \in L\ (a \# as)$
  apply (*induct set*: *T*)
  apply (*erule L.elims*)
  apply *simp-all*
  apply (*rule L0*)
  apply (*erule emb2*)
  apply (*rule L1*)
  apply (*erule lemma1*)
  apply (*erule L.elims*)
  apply *simp-all*
  apply *iprover+*
  done

lemma *lemma3*: $(ws, zs) \in T\ a \Longrightarrow ws \in good \Longrightarrow zs \in good$
  apply (*induct set*: *T*)
  apply (*erule good.elims*)
  apply *simp-all*
  apply (*rule good0*)
  apply (*erule lemma1*)
  apply (*erule good1*)
  apply (*erule good.elims*)
  apply *simp-all*
  apply (*rule good0*)
  apply (*erule lemma3'*)
  apply *iprover+*
  done

lemma *lemma4*: $(ws, zs) \in R\ a \Longrightarrow ws \neq [] \Longrightarrow (ws, zs) \in T\ a$
  apply (*induct set*: *R*)
  apply *iprover*
  apply (*case-tac vs*)
  apply (*erule R.elims*)
  apply *simp*
  apply (*case-tac a*)
  apply (*rule-tac b=B* **in** *T0*)
  apply *simp*
  apply (*rule R0*)
  apply (*rule-tac b=A* **in** *T0*)
  apply *simp*
  apply (*rule R0*)
  apply *simp*
  apply (*rule T1*)
  apply *simp*
  done

lemma *letter-neq*: $(a::letter) \neq b \Longrightarrow c \neq a \Longrightarrow c = b$
  apply (*case-tac a*)
  apply (*case-tac b*)
  apply (*case-tac c*, *simp*, *simp*)
  apply (*case-tac c*, *simp*, *simp*)
  apply (*case-tac b*)
  apply (*case-tac c*, *simp*, *simp*)
  apply (*case-tac c*, *simp*, *simp*)
  done

lemma *letter-eq-dec*: $(a::letter) = b \vee a \neq b$
  apply (*case-tac a*)
  apply (*case-tac b*)
  apply *simp*
  apply *simp*
  apply (*case-tac b*)
  apply *simp*

```
  apply simp
  done

theorem prop2:
  assumes ab: a ≠ b and bar: xs ∈ bar
  shows ⋀ys zs. ys ∈ bar ⟹ (xs, zs) ∈ T a ⟹ (ys, zs) ∈ T b ⟹ zs ∈ bar
using bar
proof induct
  fix xs zs assume xs ∈ good and (xs, zs) ∈ T a
  show zs ∈ bar by (rule bar1) (rule lemma3)
next
  fix xs ys
  assume I: ⋀w ys zs. ys ∈ bar ⟹ (w # xs, zs) ∈ T a ⟹ (ys, zs) ∈ T b ⟹
zs ∈ bar
  assume ys ∈ bar
  thus ⋀zs. (xs, zs) ∈ T a ⟹ (ys, zs) ∈ T b ⟹ zs ∈ bar
  proof induct
    fix ys zs assume ys ∈ good and (ys, zs) ∈ T b
    show zs ∈ bar by (rule bar1) (rule lemma3)
  next
    fix ys zs assume I′: ⋀w zs. (xs, zs) ∈ T a ⟹ (w # ys, zs) ∈ T b ⟹ zs ∈
bar
    and ys: ⋀w. w # ys ∈ bar and Ta: (xs, zs) ∈ T a and Tb: (ys, zs) ∈ T b
    show zs ∈ bar
    proof (rule bar2)
      fix w
      show w # zs ∈ bar
      proof (cases w)
        case Nil
        thus ?thesis by simp (rule prop1)
      next
        case (Cons c cs)
        from letter-eq-dec show ?thesis
        proof
          assume ca: c = a
          from ab have (a # cs) # zs ∈ bar by (iprover intro: I ys Ta Tb)
          thus ?thesis by (simp add: Cons ca)
        next
          assume c ≠ a
          with ab have cb: c = b by (rule letter-neq)
          from ab have (b # cs) # zs ∈ bar by (iprover intro: I′ Ta Tb)
          thus ?thesis by (simp add: Cons cb)
        qed
      qed
    qed
  qed
qed

theorem prop3:
```

```
  assumes bar: xs ∈ bar
  shows ⋀zs. xs ≠ [] ⟹ (xs, zs) ∈ R a ⟹ zs ∈ bar using bar
proof induct
  fix xs zs
  assume xs ∈ good and (xs, zs) ∈ R a
  show zs ∈ bar by (rule bar1) (rule lemma2)
next
  fix xs zs
  assume I: ⋀w zs. w # xs ≠ [] ⟹ (w # xs, zs) ∈ R a ⟹ zs ∈ bar
  and xsb: ⋀w. w # xs ∈ bar and xsn: xs ≠ [] and R: (xs, zs) ∈ R a
  show zs ∈ bar
  proof (rule bar2)
    fix w
    show w # zs ∈ bar
    proof (induct w)
      case Nil
      show ?case by (rule prop1)
    next
      case (Cons c cs)
      from letter-eq-dec show ?case
      proof
        assume c = a
        thus ?thesis by (iprover intro: I [simplified] R)
      next
        from R xsn have T: (xs, zs) ∈ T a by (rule lemma4)
        assume c ≠ a
        thus ?thesis by (iprover intro: prop2 Cons xsb xsn R T)
      qed
    qed
  qed
qed

theorem higman: [] ∈ bar
proof (rule bar2)
  fix w
  show [w] ∈ bar
  proof (induct w)
    show [[]] ∈ bar by (rule prop1)
  next
    fix c cs assume [cs] ∈ bar
    thus [c # cs] ∈ bar by (rule prop3) (simp, iprover)
  qed
qed

consts
  is-prefix :: 'a list ⇒ (nat ⇒ 'a) ⇒ bool

primrec
  is-prefix [] f = True
```

is-prefix $(x \# xs)\ f = (x = f\ (length\ xs) \wedge is\text{-}prefix\ xs\ f)$

theorem *good-prefix-lemma*:
assumes *bar*: $ws \in bar$
shows *is-prefix ws f* $\Longrightarrow \exists vs.\ is\text{-}prefix\ vs\ f \wedge vs \in good$ **using** *bar*
proof *induct*
case *bar1*
thus *?case* **by** *iprover*
next
case (*bar2 ws*)
have *is-prefix* $(f\ (length\ ws) \# ws)\ f$ **by** *simp*
thus *?case* **by** (*iprover intro*: *bar2*)
qed

theorem *good-prefix*: $\exists vs.\ is\text{-}prefix\ vs\ f \wedge vs \in good$
using *higman*
by (*rule good-prefix-lemma*) *simp+*

3.1 Extracting the program

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

extract *good-prefix*

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

$good\text{-}prefix \equiv \lambda x.\ good\text{-}prefix\text{-}lemma\ x\ higman$

Corresponding correctness theorem:

$is\text{-}prefix\ (good\text{-}prefix\ f)\ f \wedge good\text{-}prefix\ f \in good$

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

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

Program extracted from the proof of *higman*:

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

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*:

$prop2 \equiv$
$\lambda x\ xa\ xb\ xc\ H.$
$\quad barT\text{-}rec\ (\lambda ws\ x\ xa\ H.\ bar1\ xa)$
$\quad\ (\lambda ws\ xb\ r\ xc\ xd\ H.$

```
    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 ≡
λx xa H.
   barT-rec (λws. bar1)
    (λws x r xb.
        bar2 xb
         (list-rec (prop1 xb)
           (λa 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 ⟪
local open Higman in

val a = 16807.0;
val m = 2147483647.0;

fun nextRand seed =
  let val t = a∗seed
  in  t − m ∗ real (Real.floor(t/m)) end;

fun mk-word seed l =
  let
    val r = nextRand seed;
    val i = Real.round (r / m ∗ 10.0);
  in if i > 7 andalso l > 2 then (r, []) else
    apsnd (cons (if i mod 2 = 0 then A else B)) (mk-word r (l+1))
  end;

fun f s id-0 = mk-word s 0
  | f s (Suc n) = f (fst (mk-word s 0)) n;

val g1 = snd o (f 20000.0);
```

```
val g2 = snd o (f 50000.0);

fun f1 id-0 = [A,A]
  | f1 (Suc id-0) = [B]
  | f1 (Suc (Suc id-0)) = [A,B]
  | f1 - = [];

fun f2 id-0 = [A,A]
  | f2 (Suc id-0) = [B]
  | f2 (Suc (Suc id-0)) = [B,A]
  | f2 - = [];

val xs1 = test g1;
val xs2 = test g2;
val xs3 = test f1;
val xs4 = test f2;

end;
⟫

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: ⋀n::nat. m = n ∨ m ≠ n
  apply (induct m)
  apply (case-tac n)
  apply (case-tac [3] n)
  apply (simp only: nat.simps, iprover?)+
  done
```

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

```
lemma search: (∃j<(l::nat). (x::nat) = f j) ∨ ¬ (∃j<l. x = f j)
proof (induct l)
  case 0
  have ¬ (∃j<0. x = f j)
  proof
    assume ∃j<0. x = f j
    then obtain j where j: j < (0::nat) by iprover
    thus False by simp
```

```
  qed
  thus ?case ..
next
  case (Suc l)
  thus ?case
  proof
    assume ∃j<l. x = f j
    then obtain j where j: j < l
      and eq: x = f j by iprover
    from j have j < Suc l by simp
    with eq show ?case by iprover
  next
    assume nex: ¬ (∃j<l. x = f j)
    from nat-eq-dec show ?case
    proof
      assume eq: x = f l
      have l < Suc l by simp
      with eq show ?case by iprover
    next
      assume neq: x ≠ f l
      have ¬ (∃j<Suc l. x = f j)
      proof
        assume ∃j<Suc l. x = f j
        then obtain j where j: j < Suc l
          and eq: x = f j by iprover
        show False
        proof cases
          assume j = l
          with eq have x = f l by simp
          with neq show False ..
        next
          assume j ≠ l
          with j have j < l by simp
          with nex and eq show False by iprover
        qed
      qed
      thus ?case ..
    qed
  qed
qed
```

This proof yields a polynomial program.

```
theorem pigeonhole:
  ⋀f. (⋀i. i ≤ Suc n ⟹ f i ≤ n) ⟹ ∃i j. i ≤ Suc n ∧ j < i ∧ f i = f j
proof (induct n)
  case 0
  hence Suc 0 ≤ Suc 0 ∧ 0 < Suc 0 ∧ f (Suc 0) = f 0 by simp
  thus ?case by iprover
next
```

```
case (Suc n)
{
  fix k
  have
    k ≤ Suc (Suc n) ⟹
    (⋀i j. Suc k ≤ i ⟹ i ≤ Suc (Suc n) ⟹ j < i ⟹ f i ≠ f j) ⟹
    (∃ i j. i ≤ k ∧ j < i ∧ f i = f j)
  proof (induct k)
    case 0
    let ?f = λi. if f i = Suc n then f (Suc (Suc n)) else f i
    have ¬ (∃ i j. i ≤ Suc n ∧ j < i ∧ ?f i = ?f j)
    proof
      assume ∃ i j. i ≤ Suc n ∧ j < i ∧ ?f i = ?f j
      then obtain i j where i: i ≤ Suc n and j: j < i
        and f: ?f i = ?f j by iprover
      from j have i-nz: Suc 0 ≤ i by simp
      from i have iSSn: i ≤ Suc (Suc n) by simp
      have S0SSn: Suc 0 ≤ Suc (Suc n) by simp
      show False
      proof cases
        assume fi: f i = Suc n
        show False
        proof cases
          assume fj: f j = Suc n
          from i-nz and iSSn and j have f i ≠ f j by (rule 0)
          moreover from fi have f i = f j
            by (simp add: fj [symmetric])
          ultimately show ?thesis ..
        next
          from i and j have j < Suc (Suc n) by simp
          with S0SSn and le-refl have f (Suc (Suc n)) ≠ f j
            by (rule 0)
          moreover assume f j ≠ Suc n
          with fi and f have f (Suc (Suc n)) = f j by simp
          ultimately show False ..
        qed
      next
        assume fi: f i ≠ Suc n
        show False
        proof cases
          from i have i < Suc (Suc n) by simp
          with S0SSn and le-refl have f (Suc (Suc n)) ≠ f i
            by (rule 0)
          moreover assume f j = Suc n
          with fi and f have f (Suc (Suc n)) = f i by simp
          ultimately show False ..
        next
          from i-nz and iSSn and j
          have f i ≠ f j by (rule 0)
```

```
      moreover assume f j ≠ Suc n
      with fi and f have f i = f j by simp
      ultimately show False ..
    qed
  qed
qed
moreover have ⋀i. i ≤ Suc n ⟹ ?f i ≤ n
proof −
  fix i assume i ≤ Suc n
  hence i: i < Suc (Suc n) by simp
  have f (Suc (Suc n)) ≠ f i
    by (rule 0) (simp-all add: i)
  moreover have f (Suc (Suc n)) ≤ Suc n
    by (rule Suc) simp
  moreover from i have i ≤ Suc (Suc n) by simp
  hence f i ≤ Suc n by (rule Suc)
  ultimately show ?thesis i
    by simp
qed
hence ∃ i j. i ≤ Suc n ∧ j < i ∧ ?f i = ?f j
  by (rule Suc)
ultimately show ?case ..
next
case (Suc k)
from search show ?case
proof
  assume ∃ j<Suc k. f (Suc k) = f j
  thus ?case by (iprover intro: le-refl)
next
  assume nex: ¬ (∃ j<Suc k. f (Suc k) = f j)
  have ∃ i j. i ≤ k ∧ j < i ∧ f i = f j
  proof (rule Suc)
    from Suc show k ≤ Suc (Suc n) by simp
    fix i j assume k: Suc k ≤ i and i: i ≤ Suc (Suc n)
      and j: j < i
    show f i ≠ f j
    proof cases
      assume eq: i = Suc k
      show ?thesis
      proof
        assume f i = f j
        hence f (Suc k) = f j by (simp add: eq)
        with nex and j and eq show False by iprover
      qed
    next
      assume i ≠ Suc k
      with k have Suc (Suc k) ≤ i by simp
      thus ?thesis using i and j by (rule Suc)
    qed
```

```
      qed
      thus ?thesis by (iprover intro: le-SucI)
    qed
  qed
 }
 note r = this
 show ?case by (rule r) simp-all
qed
```

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

```
theorem pigeonhole-slow:
  ⋀f. (⋀i. i ≤ Suc n ⟹ f i ≤ n) ⟹ ∃ i j. i ≤ Suc n ∧ j < i ∧ f i = f j
proof (induct n)
  case 0
  have Suc 0 ≤ Suc 0 ..
  moreover have 0 < Suc 0 ..
  moreover from 0 have f (Suc 0) = f 0 by simp
  ultimately show ?case by iprover
next
  case (Suc n)
  from search show ?case
  proof
    assume ∃ j < Suc (Suc n). f (Suc (Suc n)) = f j
    thus ?case by (iprover intro: le-refl)
  next
    assume ¬ (∃ j < Suc (Suc n). f (Suc (Suc n)) = f j)
    hence nex: ∀ j < Suc (Suc n). f (Suc (Suc n)) ≠ f j by iprover
    let ?f = λi. if f i = Suc n then f (Suc (Suc n)) else f i
    have ⋀i. i ≤ Suc n ⟹ ?f i ≤ n
    proof -
      fix i assume i: i ≤ Suc n
      show ?thesis i
      proof (cases f i = Suc n)
        case True
        from i and nex have f (Suc (Suc n)) ≠ f i by simp
        with True have f (Suc (Suc n)) ≠ Suc n by simp
        moreover from Suc have f (Suc (Suc n)) ≤ Suc n by simp
        ultimately have f (Suc (Suc n)) ≤ n by simp
        with True show ?thesis by simp
      next
        case False
        from Suc and i have f i ≤ Suc n by simp
        with False show ?thesis by simp
      qed
    qed
    hence ∃ i j. i ≤ Suc n ∧ j < i ∧ ?f i = ?f j by (rule Suc)
    then obtain i j where i: i ≤ Suc n and ji: j < i and f: ?f i = ?f j
      by iprover
```

```
    have f i = f j
    proof (cases f i = Suc n)
      case True
      show ?thesis
      proof (cases f j = Suc n)
        assume f j = Suc n
        with True show ?thesis by simp
      next
        assume f j ≠ Suc n
        moreover from i ji nex have f (Suc (Suc n)) ≠ f j by simp
        ultimately show ?thesis using True f by simp
      qed
    next
      case False
      show ?thesis
      proof (cases f j = Suc n)
        assume f j = Suc n
        moreover from i nex have f (Suc (Suc n)) ≠ f i by simp
        ultimately show ?thesis using False f by simp
      next
        assume f j ≠ Suc n
        with False f show ?thesis by simp
      qed
    qed
    moreover from i have i ≤ Suc (Suc n) by simp
    ultimately show ?thesis using ji by iprover
  qed
qed

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) (xa (Suc x)) xa 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)) (xa (Suc (Suc x))) xa 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

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

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.

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

code-module *PH*
contains
  test = λ*n*. *pigeonhole n* (λ*m*. *m* − *1*)
  test′ = λ*n*. *pigeonhole-slow n* (λ*m*. *m* − *1*)
  sel = *op* !

ML *timeit* (*fn* () => *PH*.*test 10*)
ML *timeit* (*fn* () => *PH*.*test′ 10*)
ML *timeit* (*fn* () => *PH*.*test 20*)
ML *timeit* (*fn* () => *PH*.*test′ 20*)
ML *timeit* (*fn* () => *PH*.*test 25*)
ML *timeit* (*fn* () => *PH*.*test′ 25*)
ML *timeit* (*fn* () => *PH*.*test 500*)

ML *PH*.*pigeonhole 8* (*PH*.*sel* [*0*,*1*,*2*,*3*,*4*,*5*,*6*,*3*,*7*,*8*])

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.