

Examples for program extraction in Higher-Order Logic

Stefan Berghofer

June 21, 2010

Contents

1	Auxiliary lemmas used in program extraction examples	1
2	Quotient and remainder	3
3	Greatest common divisor	4
4	Warshall's algorithm	6
5	Combinator syntax for generic, open state monads (single threaded monads)	12
5.1	Motivation	12
5.2	State transformations and combinators	12
5.3	Monad laws	13
5.4	Syntax	13
6	Higman's lemma	15
6.1	Extracting the program	21
6.2	Some examples	23
7	The pigeonhole principle	25
8	Euclid's theorem	31

1 Auxiliary lemmas used in program extraction examples

```
theory Util
imports Main
begin
```

Decidability of equality on natural numbers.

```

lemma nat-eq-dec:  $\bigwedge n::nat. m = n \vee m \neq n$ 
  apply (induct m)
  apply (case-tac n)
  apply (case-tac [?] n)
  apply (simp only: nat.simps, iprover?)
  done

```

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  $P z$ 
proof (rule R)
  show  $\bigwedge y. y < z \implies P y$ 
  proof (induct z)
    case 0
    thus ?case by simp
  next
  case (Suc n y)
  from nat-eq-dec show ?case
  proof
    assume ny:  $n = y$ 
    have  $P n$ 
    by (rule R) (rule Suc)
    with ny show ?case by simp
  next
  assume  $n \neq y$ 
  with Suc have  $y < n$  by simp
  thus ?case by (rule Suc)
  qed
qed
qed

```

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 (induct y)
  case 0 show ?case by simp
next
  case (Suc z)
  thus ?case
  proof
    assume  $\exists x < z. P x$ 
    then obtain x where le:  $x < z$  and P:  $P x$  by iprover
    from le have  $x < Suc z$  by simp
    with P show ?case by iprover
  next
  assume nex:  $\neg (\exists x < z. P x)$ 

```

```

from dec show ?case
proof
  assume  $P: P\ z$ 
  have  $z < \text{Suc}\ z$  by simp
  with  $P$  show ?thesis by iprover
next
  assume  $nP: \neg P\ z$ 
  have  $\neg (\exists x < \text{Suc}\ z. P\ x)$ 
  proof
    assume  $\exists x < \text{Suc}\ z. P\ x$ 
    then obtain  $x$  where  $le: x < \text{Suc}\ z$  and  $P: P\ x$  by iprover
    have  $x < z$ 
    proof (cases  $x = z$ )
      case True
        with  $nP$  and  $P$  show ?thesis by simp
      next
        case False
          with  $le$  show ?thesis by simp
    qed
    with  $P$  have  $\exists x < z. P\ x$  by iprover
    with  $nex$  show False ..
  qed
  thus ?case by iprover
qed
qed
qed
end

```

2 Quotient and remainder

```

theory QuotRem
imports Util
begin

```

Derivation of quotient and remainder using program extraction.

```

theorem division:  $\exists r\ q. a = \text{Suc}\ b * q + r \wedge r \leq b$ 
proof (induct  $a$ )
  case  $0$ 
    have  $0 = \text{Suc}\ b * 0 + 0 \wedge 0 \leq b$  by simp
    thus ?case by iprover
  next
    case ( $\text{Suc}\ a$ )
    then obtain  $r\ q$  where  $I: a = \text{Suc}\ b * q + r$  and  $r \leq b$  by iprover
    from nat-eq-dec show ?case
  proof
    assume  $r = b$ 
    with  $I$  have  $\text{Suc}\ a = \text{Suc}\ b * (\text{Suc}\ q) + 0 \wedge 0 \leq b$  by simp

```

```

    thus ?case by iprover
next
  assume  $r \neq b$ 
  with  $\langle r \leq b \rangle$  have  $r < b$  by (simp add: order-less-le)
  with  $I$  have  $\text{Suc } a = \text{Suc } b * q + (\text{Suc } r) \wedge (\text{Suc } r) \leq b$  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-induct- $P$   $x$   $(0, 0)$ 
  ( $\lambda a H.$  let  $(x, y) = H$ 
    in case nat-eq-dec  $x xa$  of Left  $\Rightarrow (0, \text{Suc } y)$ 
    | Right  $\Rightarrow (\text{Suc } x, y)$ )

```

The corresponding correctness theorem is

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

```

lemma division 9 2 = (0, 3) by evaluation
lemma division 9 2 = (0, 3) by eval

```

```

end

```

3 Greatest common divisor

```

theory Greatest-Common-Divisor
imports QuotRem
begin

```

```

theorem greatest-common-divisor:

```

$$\bigwedge n::\text{nat}. \text{Suc } m < n \implies \exists k \ n1 \ m1. k * n1 = n \wedge k * m1 = \text{Suc } m \wedge$$

$$(\forall l \ l1 \ l2. l * l1 = n \longrightarrow l * l2 = \text{Suc } m \longrightarrow l \leq k)$$

```

proof (induct m rule: nat-wf-ind)

```

```

  case (1 m n)

```

```

  from division obtain r q where h1:  $n = \text{Suc } m * q + r$  and h2:  $r \leq m$ 
  by iprover

```

```

  show ?case

```

```

  proof (cases r)

```

```

    case 0

```

```

    with h1 have  $\text{Suc } m * q = n$  by simp

```

```

    moreover have  $\text{Suc } m * 1 = \text{Suc } m$  by simp

```

```

    moreover {

```

```

    fix l2 have  $\bigwedge l \ l1. \ l * l1 = n \implies l * l2 = \text{Suc } m \implies l \leq \text{Suc } m$ 
      by (cases l2) simp-all }
    ultimately show ?thesis by iprover
next
case (Suc nat)
with h2 have h:  $\text{nat} < m$  by simp
moreover from h have  $\text{Suc } \text{nat} < \text{Suc } m$  by simp
ultimately have  $\exists k \ m1 \ r1. \ k * m1 = \text{Suc } m \wedge k * r1 = \text{Suc } \text{nat} \wedge$ 
  ( $\forall l \ l1 \ l2. \ l * l1 = \text{Suc } m \longrightarrow l * l2 = \text{Suc } \text{nat} \longrightarrow l \leq k$ )
  by (rule 1)
then obtain k m1 r1 where
  h1':  $k * m1 = \text{Suc } m$ 
  and h2':  $k * r1 = \text{Suc } \text{nat}$ 
  and h3':  $\bigwedge l \ l1 \ l2. \ l * l1 = \text{Suc } m \implies l * l2 = \text{Suc } \text{nat} \implies l \leq k$ 
  by iprover
have mn:  $\text{Suc } m < n$  by (rule 1)
from h1 h1' h2' Suc have  $k * (m1 * q + r1) = n$ 
  by (simp add: add-mult-distrib2 nat-mult-assoc [symmetric])
moreover have  $\bigwedge l \ l1 \ l2. \ l * l1 = n \implies l * l2 = \text{Suc } m \implies l \leq k$ 
proof -
  fix l l1 l2
  assume ll1n:  $l * l1 = n$ 
  assume ll2m:  $l * l2 = \text{Suc } m$ 
  moreover have  $l * (l1 - l2 * q) = \text{Suc } \text{nat}$ 
  by (simp add: diff-mult-distrib2 h1 Suc [symmetric] mn ll1n ll2m [symmetric])
  ultimately show  $l \leq k$  by (rule h3')
qed
ultimately show ?thesis using h1' by iprover
qed
qed

```

extract *greatest-common-divisor*

The extracted program for computing the greatest common divisor is

```

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

```

instantiation $\text{nat} :: \text{default}$
begin

definition $\text{default} = (0 :: \text{nat})$

instance ..

```

end

instantiation * :: (default, default) default
begin

definition default = (default, default)

instance ..

end

instantiation fun :: (type, default) default
begin

definition default = ( $\lambda x.$  default)

instance ..

end

consts-code
  default ((error default))

lemma greatest-common-divisor 7 12 = (4, 3, 2) by evaluation
lemma greatest-common-divisor 7 12 = (4, 3, 2) by eval

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  $\Rightarrow$  'a  $\Rightarrow$  b)  $\Rightarrow$  'a  $\Rightarrow$  'a list  $\Rightarrow$  'a  $\Rightarrow$  bool
where
  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)

```

```

definition
  is-path :: (nat  $\Rightarrow$  nat  $\Rightarrow$  b)  $\Rightarrow$  (nat * nat list * nat)  $\Rightarrow$ 
    nat  $\Rightarrow$  nat  $\Rightarrow$  nat  $\Rightarrow$  bool

```

where

$is-path\ r\ p\ i\ j\ k \iff fst\ p = j \wedge snd\ (snd\ p) = k \wedge$
 $list-all\ (\lambda x. x < i)\ (fst\ (snd\ p)) \wedge$
 $is-path'\ r\ (fst\ p)\ (fst\ (snd\ p))\ (snd\ (snd\ p))$

definition

$conc :: ('a * 'a\ list * 'a) \Rightarrow ('a * 'a\ list * 'a) \Rightarrow ('a * 'a\ list * 'a)$

where

$conc\ p\ q = (fst\ p, fst\ (snd\ p) @ fst\ q \# fst\ (snd\ q), snd\ (snd\ q))$

theorem *is-path'-snoc* [simp]:

$\bigwedge x. is-path'\ r\ x\ (ys @ [y])\ z = (is-path'\ r\ x\ ys\ y \wedge r\ y\ z = T)$
by (induct ys) simp+

theorem *list-all-scoc* [simp]: $list-all\ P\ (xs @ [x]) = (P\ x \wedge list-all\ P\ xs)$

by (induct xs, simp+, iprover)

theorem *list-all-lemma*:

$list-all\ P\ xs \implies (\bigwedge x. P\ x \implies Q\ x) \implies list-all\ Q\ xs$

proof –

assume $PQ: \bigwedge x. P\ x \implies Q\ x$

show $list-all\ P\ xs \implies list-all\ Q\ xs$

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*: $\bigwedge p. is-path\ r\ p\ i\ j\ k \implies is-path\ r\ p\ (Suc\ i)\ j\ k$

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*: $\bigwedge p. is-path\ r\ p\ 0\ j\ k \implies r\ j\ k = T$

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*: $is-path' r j xs i \implies is-path' r i ys k \implies is-path' r j (xs @ i \# ys) k$

proof –

assume *pys*: $is-path' r i ys k$

show $\bigwedge j. is-path' r j xs i \implies is-path' r j (xs @ i \# ys) k$

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*:

$\bigwedge p q. is-path r p i j i \implies is-path r q i i k \implies$

$is-path r (conc p q) (Suc i) j k$

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*:

$\bigwedge p. is-path r p (Suc i) j k \implies \sim is-path r p i j k \implies$

$(\exists q. is-path r q i j i) \wedge (\exists q'. is-path r q' i i k)$

proof (*simp cong add: conj-cong add: split-paired-all is-path-def, (erule conjE)*) +

fix *xs*

assume *asms*:

list-all $(\lambda x. x < Suc i) xs$

$is-path' r j xs k$

$\neg list-all (\lambda x. x < i) xs$

show $(\exists ys. list-all (\lambda x. x < i) ys \wedge is-path' r j ys i) \wedge$

$(\exists ys. list-all (\lambda x. x < i) ys \wedge is-path' r i ys k)$

proof

show $\bigwedge j. list-all (\lambda x. x < Suc i) xs \implies is-path' r j xs k \implies$

$\neg list-all (\lambda x. x < i) xs \implies$

$\exists ys. list-all (\lambda x. x < i) ys \wedge is-path' r j ys i$ (**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  $list\text{-}all\ (\lambda x. x < i)\ [] \wedge is\text{-}path'\ r\ j\ []\ i$  by simp
    qed
  next
  case False
  have PROP ?ih as by (rule Cons)
  then obtain ys where ys:  $list\text{-}all\ (\lambda x. x < i)\ ys \wedge is\text{-}path'\ r\ a\ ys\ i$ 
  proof
    from Cons show  $list\text{-}all\ (\lambda x. x < Suc\ i)\ as$  by simp
    from Cons show  $is\text{-}path'\ r\ a\ as\ k$  by simp
    from Cons and False show  $\neg list\text{-}all\ (\lambda x. x < i)\ as$  by (simp)
  qed
  show ?thesis
  proof
    from Cons False ys
    show  $list\text{-}all\ (\lambda x. x < i)\ (a\#ys) \wedge is\text{-}path'\ r\ j\ (a\#ys)\ i$  by simp
  qed
qed
qed
show  $\bigwedge k. list\text{-}all\ (\lambda x. x < Suc\ i)\ xs \implies is\text{-}path'\ r\ j\ xs\ k \implies$ 
 $\neg list\text{-}all\ (\lambda x. x < i)\ xs \implies$ 
 $\exists ys. list\text{-}all\ (\lambda x. x < i)\ ys \wedge is\text{-}path'\ r\ i\ ys\ k$  (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  $list\text{-}all\ (\lambda x. x < i)\ [] \wedge is\text{-}path'\ r\ i\ []\ k$  by simp
    qed
  next
  case False
  have PROP ?ih as by (rule snoc)
  then obtain ys where ys:  $list\text{-}all\ (\lambda x. x < i)\ ys \wedge is\text{-}path'\ r\ i\ ys\ a$ 

```

```

proof
  from snoc show list-all ( $\lambda x. x < \text{Suc } i$ ) as by simp
  from snoc show is-path' r j as a by simp
  from snoc and False show  $\neg \text{list-all } (\lambda x. x < i)$  as by simp
qed
show ?thesis
proof
  from snoc False ys
  show list-all ( $\lambda x. x < i$ ) (ys @ [a])  $\wedge$  is-path' r i (ys @ [a]) k
    by simp
qed
qed
qed
qed (rule asms) +
qed

```

theorem lemma5':
 $\bigwedge p. \text{is-path } r p (\text{Suc } i) j k \implies \neg \text{is-path } r p i j k \implies$
 $\neg (\forall q. \neg \text{is-path } r q i j i) \wedge \neg (\forall q'. \neg \text{is-path } r q' i i k)$
by (*iprover dest: lemma5*)

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

```

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  $\exists p. \text{is-path } r p 0 j k$  ..
    thus ?thesis ..
  next
    assume r j k = F
    hence r j k  $\sim = T$  by simp
    hence  $\neg (\exists p. \text{is-path } r p 0 j k)$ 
      by (iprover dest: lemma2)
    thus ?thesis ..

```

```

qed
next
  case (Suc i j k)
  thus ?case
  proof
    assume h1:  $\neg (\exists p. \text{is-path } r p i j k)$ 
    from Suc show ?case
    proof
      assume  $\neg (\exists p. \text{is-path } r p i j i)$ 
      with h1 have  $\neg (\exists p. \text{is-path } r p (\text{Suc } i) j k)$ 
        by (iprover dest: lemma5')
    
```

```

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

```

extract *warshall*

The program extracted from the above proof looks as follows

```

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

```

The corresponding correctness theorem is

```

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

```

ML @{code warshall}

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 $\alpha, \beta, \gamma, \dots$ 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** $o \rightarrow 60$)

abbreviation (*input*)

return \equiv *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* $o\rightarrow$ 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

```

print-translation \ll

```

let
  fun dest-abs-eta (Abs (abs as (-, ty, -))) =
    let
      val (v, t) = Syntax.variant-abs abs;
    in (Free (v, ty), t) end
  | dest-abs-eta t =
    let
      val (v, t) = Syntax.variant-abs (, dummyT, t $ Bound 0);
    in (Free (v, dummyT), t) end;
fun unfold-monad (Const (@{const-syntax scomp}, -) $ f $ g) =
  let
    val (v, g') = dest-abs-eta g;
  in Const (@{syntax-const -scomp}, dummyT) $ v $ f $ unfold-monad g' end
  | unfold-monad (Const (@{const-syntax fcomp}, -) $ f $ g) =
    Const (@{syntax-const -fcomp}, dummyT) $ f $ unfold-monad g
  | unfold-monad (Const (@{const-syntax Let}, -) $ f $ g) =
    let
      val (v, g') = dest-abs-eta g;
    in Const (@{syntax-const -let}, dummyT) $ v $ f $ unfold-monad g' end
  | unfold-monad (Const (@{const-syntax Pair}, -) $ f) =
    Const (@{const-syntax return}, dummyT) $ f
  | unfold-monad f = f;
fun contains-scomp (Const (@{const-syntax scomp}, -) $ - $ -) = true
  | contains-scomp (Const (@{const-syntax fcomp}, -) $ - $ t) =
    contains-scomp t

```

```

| contains-scomp (Const (@{const-syntax Let}, -) $ - $ Abs (-, -, t)) =
  contains-scomp t;
fun scomp-monad-tr' (f::g::ts) = list-comb
  (Const (@{syntax-const -do}, dummyT) $
    unfold-monad (Const (@{const-syntax scomp}, dummyT) $ f $ g), ts);
fun fcomp-monad-tr' (f::g::ts) =
  if contains-scomp g then list-comb
    (Const (@{syntax-const -do}, dummyT) $
      unfold-monad (Const (@{const-syntax fcomp}, dummyT) $ f $ g), ts)
  else raise Match;
fun Let-monad-tr' (f :: (g as Abs (-, -, g')) :: ts) =
  if contains-scomp g' then list-comb
    (Const (@{syntax-const -do}, dummyT) $
      unfold-monad (Const (@{const-syntax Let}, dummyT) $ f $ g), ts)
  else raise Match;
in
  [(@{const-syntax scomp}, scomp-monad-tr'),
   (@{const-syntax fcomp}, fcomp-monad-tr'),
   (@{const-syntax Let}, Let-monad-tr')]
end;
>>

```

For an example, see HOL/Extraction/Higman.thy.

end

6 Higman's lemma

```

theory Higman
imports Main State-Monad Random
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  $\Rightarrow$  bool
where
  good0 [Pure.intro]: L w ws  $\Rightarrow$  good (w # ws)
  | good1 [Pure.intro]: good ws  $\Rightarrow$  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  $\Rightarrow$  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  $\Rightarrow$  R b ws zs  $\Rightarrow$  T a (w # zs) ((a # w) # zs)
  | T1 [Pure.intro]: T a ws zs  $\Rightarrow$  T a (w # ws) ((a # w) # zs)
  | T2 [Pure.intro]: a  $\neq$  b  $\Rightarrow$  T a ws zs  $\Rightarrow$  T a ws ((b # w) # zs)

inductive bar :: letter list list  $\Rightarrow$  bool
where
  bar1 [Pure.intro]: good ws  $\Rightarrow$  bar ws
  | bar2 [Pure.intro]: ( $\bigwedge$  w. bar (w # ws))  $\Rightarrow$  bar ws

theorem prop1: bar ([] # ws) by iprover

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

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

lemma lemma2: R a vs ws  $\Rightarrow$  good vs  $\Rightarrow$  good ws
apply (induct set: R)
apply iprover
apply (erule good.cases)
apply simp-all
apply (rule good0)
apply (erule lemma2')
apply assumption
apply (erule good1)
done

```



```

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

```

```

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

```

```

lemma lemma4:  $R\ a\ ws\ zs \implies ws \neq [] \implies T\ a\ ws\ zs$ 
  apply (induct set: R)
  apply iprover
  apply (case-tac vs)
  apply (erule R.cases)
  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-neg:  $(a::letter) \neq b \implies c \neq a \implies c = b$ 
  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  $\neq$  b and bar: bar xs
  shows  $\bigwedge$ ys zs. bar ys  $\implies$  T a xs zs  $\implies$  T b ys zs  $\implies$  bar zs using bar
proof induct
  fix xs zs assume T a xs zs and good xs
  hence good zs by (rule lemma3)
  then show bar zs by (rule bar1)
next
  fix xs ys
  assume I:  $\bigwedge$ w ys zs. bar ys  $\implies$  T a (w # xs) zs  $\implies$  T b ys zs  $\implies$  bar zs
  assume bar ys
  thus  $\bigwedge$ zs. T a xs zs  $\implies$  T b ys zs  $\implies$  bar zs
  proof induct
    fix ys zs assume T b ys zs and good ys
    then have good zs by (rule lemma3)
    then show bar zs by (rule bar1)
  next
    fix ys zs assume I':  $\bigwedge$ w zs. T a xs zs  $\implies$  T b (w # ys) zs  $\implies$  bar zs
    and ys:  $\bigwedge$ w. bar (w # ys) and Ta: T a xs zs and Tb: T b ys zs
    show bar zs
    proof (rule bar2)
      fix w
      show bar (w # zs)
      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 bar ((a # cs) # zs) by (iprover intro: I ys Ta Tb)
          qed
      qed
    qed
  qed

```

```

      thus ?thesis by (simp add: Cons ca)
    next
      assume  $c \neq a$ 
      with  $ab$  have  $cb: c = b$  by (rule letter-neq)
      from  $ab$  have  $bar ((b \# cs) \# zs)$  by (iprover intro: I' Ta Tb)
      thus ?thesis by (simp add: Cons cb)
    qed
  qed
qed
qed
qed

theorem prop3:
  assumes  $bar: bar\ xs$ 
  shows  $\bigwedge zs. xs \neq [] \implies R\ a\ xs\ zs \implies bar\ zs$  using  $bar$ 
proof induct
  fix  $xs\ zs$ 
  assume  $R\ a\ xs\ zs$  and  $good\ xs$ 
  then have  $good\ zs$  by (rule lemma2)
  then show  $bar\ zs$  by (rule bar1)
next
  fix  $xs\ zs$ 
  assume  $I: \bigwedge w\ zs. w \# xs \neq [] \implies R\ a\ (w \# xs)\ zs \implies bar\ zs$ 
  and  $xs_b: \bigwedge w. bar\ (w \# xs)$  and  $xsn: xs \neq []$  and  $R: R\ a\ xs\ zs$ 
  show  $bar\ zs$ 
proof (rule bar2)
  fix  $w$ 
  show  $bar\ (w \# zs)$ 
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  $T: T\ a\ xs\ zs$  by (rule lemma4)
  assume  $c \neq a$ 
  thus ?thesis by (iprover intro: prop2 Cons  $xs_b\ xsn\ R\ T$ )
qed
qed
qed
qed

theorem hlgman:  $bar\ []$ 
proof (rule bar2)
  fix  $w$ 

```

```

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

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

theorem L-idx:
  assumes L: L w ws
  shows is-prefix ws f  $\Longrightarrow$   $\exists i. \text{emb } (f\ i)\ w \wedge i < \text{length } ws$  using L
proof induct
  case (L0 v ws)
  hence emb (f (length ws)) w by simp
  moreover have length ws < length (v # ws) by simp
  ultimately show ?case by iprover
next
  case (L1 ws v)
  then obtain i where emb: emb (f i) w and i < length ws
  by simp iprover
  hence i < length (v # ws) by simp
  with emb show ?case by iprover
qed

theorem good-idx:
  assumes good: good ws
  shows is-prefix ws f  $\Longrightarrow$   $\exists i\ j. \text{emb } (f\ i)\ (f\ j) \wedge i < j$  using good
proof induct
  case (good0 w ws)
  hence w = f (length ws) and is-prefix ws f by simp-all
  with good0 show ?case by (iprover dest: L-idx)
next
  case (good1 ws w)
  thus ?case by simp
qed

theorem bar-idx:
  assumes bar: bar ws
  shows is-prefix ws f  $\Longrightarrow$   $\exists i\ j. \text{emb } (f\ i)\ (f\ j) \wedge i < j$  using bar
proof induct
  case (bar1 ws)
  thus ?case by (rule good-idx)

```

```

next
  case (bar2 ws)
  hence is-prefix (f (length ws) # ws) f by simp
  thus ?case by (rule bar2)
qed

```

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

```

theorem higman-idx:  $\exists (i::nat) j. \text{emb } (f\ i) (f\ j) \wedge i < j$ 
proof (rule bar-idx)
  show bar [] by (rule higman)
  show is-prefix [] f by simp
qed

```

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  $\implies \exists vs. \text{is-prefix } vs\ f \wedge \text{good } vs$  using bar
proof induct
  case bar1
  thus ?case by iprover
next
  case (bar2 ws)
  from bar2.prem1s have is-prefix (f (length ws) # ws) f by simp
  thus ?case by (iprover intro: bar2)
qed

```

```

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

```

6.1 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*:

$\text{higman-idx} \equiv \lambda x. \text{bar-idx } x \text{ higman}$

Corresponding correctness theorem:

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

Program extracted from the proof of *higman*:

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

Program extracted from the proof of *prop1*:

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

Program extracted from the proof of *prop2*:

prop2 \equiv
 $\lambda x \ x a \ x b \ x c \ H.$
 $\text{barT-rec } (\lambda w s \ x a \ x b \ x c \ H \ H a \ H b. \text{ bar1 } x c \ (\text{lemma3 } x \ H a \ x a))$
 $(\lambda w s \ x b \ r \ x c \ x d \ H.$
 $\text{barT-rec } (\lambda w s \ x \ x b \ H \ H a. \text{ bar1 } x b \ (\text{lemma3 } x a \ H a \ x))$
 $(\lambda w s a \ x b \ r a \ x c \ H \ H a.$
 $\text{bar2 } x c$
 $(\text{list-case } (\text{prop1 } x c)$
 $(\lambda a \ \text{list}.$
 $\text{case letter-eq-dec } a \ x \ \text{of}$
 $\text{Left} \Rightarrow$
 $r \ \text{list} \ w s a \ ((x \ \# \ \text{list}) \ \# \ x c) \ (\text{bar2 } w s a \ x b)$
 $(T1 \ w s \ x c \ \text{list} \ H) \ (T2 \ x \ w s a \ x c \ \text{list} \ H a)$
 $| \ \text{Right} \Rightarrow$
 $r a \ \text{list} \ ((x a \ \# \ \text{list}) \ \# \ x c) \ (T2 \ x a \ w s \ x c \ \text{list} \ H)$
 $(T1 \ w s a \ x c \ \text{list} \ H a))))$
 $H \ x d)$
 $H \ x b \ x c$

Program extracted from the proof of *prop3*:

prop3 \equiv
 $\lambda x \ x a \ H.$
 $\text{barT-rec } (\lambda w s \ x a \ x b \ H. \text{ bar1 } x b \ (\text{lemma2 } x \ H \ x a))$
 $(\lambda w s \ x a \ r \ x b \ H.$
 $\text{bar2 } x b$
 $(\text{list-rec } (\text{prop1 } x b)$
 $(\lambda a \ w \ H a.$
 $\text{case letter-eq-dec } a \ x \ \text{of}$
 $\text{Left} \Rightarrow r \ w \ ((x \ \# \ w) \ \# \ x b) \ (R1 \ w s \ x b \ w \ H)$
 $| \ \text{Right} \Rightarrow$
 $\text{prop2 } a \ x \ w s \ ((a \ \# \ w) \ \# \ x b) \ H a \ (\text{bar2 } w s \ x a)$
 $(T0 \ x \ w s \ x b \ w \ H) \ (T2 \ a \ w s \ x b \ w \ (\text{lemma4 } x \ H))))$
 $H \ x a$

6.2 Some examples

instantiation *LT* and *TT* :: *default*
begin

definition *default* = *L0* [] []

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

instance ..

end

function *mk-word-aux* :: *nat* ⇒ *Random.seed* ⇒ *letter list* × *Random.seed* **where**

mk-word-aux *k* = (do
i ← *Random.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)

by *pat-completeness auto*

termination by (*relation measure* ((*op* −) 1001)) *auto*

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*) = (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 \ll

local

```

  val higman-idx = @{code higman-idx};
  val g1 = @{code g1};
  val g2 = @{code g2};
  val f1 = @{code f1};
  val f2 = @{code f2};

```

in

```

  val (i1, j1) = higman-idx g1;
  val (v1, w1) = (g1 i1, g1 j1);
  val (i2, j2) = higman-idx g2;
  val (v2, w2) = (g2 i2, g2 j2);
  val (i3, j3) = higman-idx f1;
  val (v3, w3) = (f1 i3, f1 j3);
  val (i4, j4) = higman-idx f2;
  val (v4, w4) = (f2 i4, f2 j4);

```

end;

\gg

code-module *Higman*

contains

higman = *higman-idx*

ML \ll

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 zero = 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 zero = [A,A]
  | f1 (Suc zero) = [B]
  | f1 (Suc (Suc zero)) = [A,B]
  | f1 - = [];

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

val (i1, j1) = hlgman g1;
val (v1, w1) = (g1 i1, g1 j1);
val (i2, j2) = hlgman g2;
val (v2, w2) = (g2 i2, g2 j2);
val (i3, j3) = hlgman f1;
val (v3, w3) = (f1 i3, f1 j3);
val (i4, j4) = hlgman f2;
val (v4, w4) = (f2 i4, f2 j4);

end;
>>

end

```

7 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:
   $\bigwedge f. (\bigwedge i. i \leq \text{Suc } n \implies f\ i \leq n) \implies \exists i\ j. i \leq \text{Suc } n \wedge j < i \wedge f\ i = f\ j$ 
proof (induct n)
  case 0
  hence  $\text{Suc } 0 \leq \text{Suc } 0 \wedge 0 < \text{Suc } 0 \wedge f\ (\text{Suc } 0) = f\ 0$  by simp
  thus ?case by iprover
next
  case (Suc n)
  {
    fix k

```

have
 $k \leq \text{Suc } (\text{Suc } n) \implies$
 $(\bigwedge i j. \text{Suc } k \leq i \implies i \leq \text{Suc } (\text{Suc } n) \implies j < i \implies f i \neq f j) \implies$
 $(\exists i j. i \leq k \wedge j < i \wedge f i = f j)$
 proof (induct k)
 case 0
 let ?f = $\lambda i. \text{if } f i = \text{Suc } n \text{ then } f (\text{Suc } (\text{Suc } n)) \text{ else } f i$
 have $\neg (\exists i j. i \leq \text{Suc } n \wedge j < i \wedge ?f i = ?f j)$
 proof
 assume $\exists i j. i \leq \text{Suc } n \wedge j < i \wedge ?f i = ?f j$
 then obtain i j where i: $i \leq \text{Suc } n$ and j: $j < i$
 and f: $?f i = ?f j$ by iprover
 from j have i-nz: $\text{Suc } 0 \leq i$ by simp
 from i have iSSn: $i \leq \text{Suc } (\text{Suc } n)$ by simp
 have S0SSn: $\text{Suc } 0 \leq \text{Suc } (\text{Suc } n)$ by simp
 show False
 proof cases
 assume fi: $f i = \text{Suc } n$
 show False
 proof cases
 assume fj: $f j = \text{Suc } n$
 from i-nz and iSSn and j have $f i \neq f j$ 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 $j < \text{Suc } (\text{Suc } n)$ by simp
 with S0SSn and le-refl have $f (\text{Suc } (\text{Suc } n)) \neq f j$
 by (rule 0)
 moreover assume $f j \neq \text{Suc } n$
 with fi and f have $f (\text{Suc } (\text{Suc } n)) = f j$ by simp
 ultimately show False ..
 qed
 next
 assume fi: $f i \neq \text{Suc } n$
 show False
 proof cases
 from i have $i < \text{Suc } (\text{Suc } n)$ by simp
 with S0SSn and le-refl have $f (\text{Suc } (\text{Suc } n)) \neq f i$
 by (rule 0)
 moreover assume $f j = \text{Suc } n$
 with fi and f have $f (\text{Suc } (\text{Suc } n)) = f i$ by simp
 ultimately show False ..
 next
 from i-nz and iSSn and j
 have $f i \neq f j$ by (rule 0)
 moreover assume $f j \neq \text{Suc } n$
 with fi and f have $f i = f j$ by simp
 ultimately show False ..

```

    qed
  qed
  qed
  moreover have  $\bigwedge i. i \leq \text{Suc } n \implies ?f i \leq n$ 
  proof -
    fix i assume  $i \leq \text{Suc } n$ 
    hence  $i: i < \text{Suc } (\text{Suc } n)$  by simp
    have  $f (\text{Suc } (\text{Suc } n)) \neq f i$ 
      by (rule 0) (simp-all add: i)
    moreover have  $f (\text{Suc } (\text{Suc } n)) \leq \text{Suc } n$ 
      by (rule Suc) simp
    moreover from i have  $i \leq \text{Suc } (\text{Suc } n)$  by simp
    hence  $f i \leq \text{Suc } n$  by (rule Suc)
    ultimately show ?thesis i
      by simp
  qed
  hence  $\exists i j. i \leq \text{Suc } n \wedge j < i \wedge ?f i = ?f j$ 
    by (rule Suc)
  ultimately show ?case ..
next
case (Suc k)
from search [OF nat-eq-dec] show ?case
proof
  assume  $\exists j < \text{Suc } k. f (\text{Suc } k) = f j$ 
  thus ?case by (iprover intro: le-refl)
next
assume nex:  $\neg (\exists j < \text{Suc } k. f (\text{Suc } k) = f j)$ 
have  $\exists i j. i \leq k \wedge j < i \wedge f i = f j$ 
proof (rule Suc)
  from Suc show  $k \leq \text{Suc } (\text{Suc } n)$  by simp
  fix i j assume  $k: \text{Suc } k \leq i$  and  $i: i \leq \text{Suc } (\text{Suc } n)$ 
    and  $j: j < i$ 
  show  $f i \neq f j$ 
  proof cases
    assume eq:  $i = \text{Suc } k$ 
    show ?thesis
    proof
      assume  $f i = f j$ 
      hence  $f (\text{Suc } k) = f j$  by (simp add: eq)
      with nex and j and eq show False by iprover
    qed
  qed
next
assume  $i \neq \text{Suc } k$ 
with k have  $\text{Suc } (\text{Suc } k) \leq i$  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:
   $\bigwedge f. (\bigwedge i. i \leq Suc\ n \implies f\ i \leq n) \implies \exists i\ j. i \leq Suc\ n \wedge j < i \wedge f\ i = f\ j$ 
proof (induct  $n$ )
  case 0
  have  $Suc\ 0 \leq Suc\ 0 ..$ 
  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 [OF nat-eq-dec] show  $?case$ 
proof
  assume  $\exists j < Suc\ (Suc\ n). f\ (Suc\ (Suc\ n)) = f\ j$ 
  thus  $?case$  by (iprover intro: le-refl)
next
  assume  $\neg (\exists j < Suc\ (Suc\ n). f\ (Suc\ (Suc\ n)) = f\ j)$ 
  hence nex:  $\forall j < Suc\ (Suc\ n). f\ (Suc\ (Suc\ n)) \neq f\ j$  by iprover
  let  $?f = \lambda i. if\ f\ i = Suc\ n\ then\ f\ (Suc\ (Suc\ n))\ else\ f\ i$ 
  have  $\bigwedge i. i \leq Suc\ n \implies ?f\ i \leq n$ 
  proof -
    fix  $i$  assume  $i: i \leq Suc\ n$ 
    show  $?thesis\ i$ 
    proof (cases  $f\ i = Suc\ n$ )
      case True
      from  $i$  and nex have  $f\ (Suc\ (Suc\ n)) \neq f\ i$  by simp
      with True have  $f\ (Suc\ (Suc\ n)) \neq Suc\ n$  by simp
      moreover from  $Suc$  have  $f\ (Suc\ (Suc\ n)) \leq Suc\ n$  by simp
      ultimately have  $f\ (Suc\ (Suc\ n)) \leq n$  by simp
      with True show  $?thesis$  by simp
    next
      case False
      from  $Suc$  and  $i$  have  $f\ i \leq Suc\ n$  by simp
      with False show  $?thesis$  by simp
    qed
  qed
  hence  $\exists i\ j. i \leq Suc\ n \wedge j < i \wedge ?f\ i = ?f\ j$  by (rule Suc)
  then obtain  $i\ j$  where  $i: i \leq Suc\ n$  and  $j: j < i$  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 j i 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 j i by iprover
qed
qed

```

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

```

(Λi. i ≤ Suc n ⇒ f i ≤ n) ⇒
fst (pigeonhole n f) ≤ Suc n ∧
snd (pigeonhole n f) < fst (pigeonhole n f) ∧
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 ..

end

instantiation * :: (default, default) default
begin

definition default = (default, default)

instance ..

end

definition
  test n u = pigeonhole n (λm. m - 1)
definition
  test' n u = pigeonhole-slow n (λm. m - 1)
definition
  test'' u = pigeonhole 8 (op ! [0, 1, 2, 3, 4, 5, 6, 3, 7, 8])

ML timeit (@{code test} 10)
ML timeit (@{code test'} 10)
ML timeit (@{code test} 20)
ML timeit (@{code test'} 20)
ML timeit (@{code test} 25)
ML timeit (@{code test'} 25)
ML timeit (@{code test} 500)
ML timeit @{code test''}

```

```

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

code-module PH
contains
  test = test
  test' = test'
  test'' = test''

ML timeit (PH.test 10)
ML timeit (PH.test' 10)
ML timeit (PH.test 20)
ML timeit (PH.test' 20)
ML timeit (PH.test 25)
ML timeit (PH.test' 25)
ML timeit (PH.test 500)
ML timeit PH.test''

end

```

8 Euclid's theorem

```

theory Euclid
imports ~~/src/HOL/Number-Theory/UniqueFactorization Util Efficient-Nat
begin

```

A constructive version of the proof of Euclid's theorem by Markus Wenzel and Freek Wiedijk [4].

```

lemma factor-greater-one1:  $n = m * k \implies m < n \implies k < n \implies \text{Suc } 0 < m$ 
  by (induct m) auto

```

```

lemma factor-greater-one2:  $n = m * k \implies m < n \implies k < n \implies \text{Suc } 0 < k$ 
  by (induct k) auto

```

```

lemma prod-mn-less-k:
   $(0::nat) < n \implies 0 < k \implies \text{Suc } 0 < m \implies m * n = k \implies n < k$ 
  by (induct m) auto

```

```

lemma prime-eq:  $\text{prime } (p::nat) = (1 < p \wedge (\forall m. m \text{ dvd } p \longrightarrow 1 < m \longrightarrow m = p))$ 
  apply (simp add: prime-nat-def)
  apply (rule iffI)
  apply blast
  apply (erule conjE)
  apply (rule conjI)
  apply assumption

```

```

apply (rule allI impI)+
apply (erule allE)
apply (erule impE)
apply assumption
apply (case-tac m=0)
apply simp
apply (case-tac m=Suc 0)
apply simp
apply simp
done

```

lemma *prime-eq'*: $\text{prime } (p::\text{nat}) = (1 < p \wedge (\forall m\ k. p = m * k \longrightarrow 1 < m \longrightarrow m = p))$
by (simp add: prime-eq dvd-def all-simps [symmetric] del: all-simps)

lemma *not-prime-ex-mk*:
assumes $n: \text{Suc } 0 < n$
shows $(\exists m\ k. \text{Suc } 0 < m \wedge \text{Suc } 0 < k \wedge m < n \wedge k < n \wedge n = m * k) \vee$
prime n
proof –
{
 fix k
 from *nat-eq-dec*
 have $(\exists m < n. n = m * k) \vee \neg (\exists m < n. n = m * k)$
 by (rule search)
}
hence $(\exists k < n. \exists m < n. n = m * k) \vee \neg (\exists k < n. \exists m < n. n = m * k)$
by (rule search)
thus ?thesis
proof
 assume $\exists k < n. \exists m < n. n = m * k$
 then obtain $k\ m$ **where** $k: k < n$ **and** $m: m < n$ **and** $nmk: n = m * k$
 by *iprover*
 from $nmk\ m\ k$ **have** $\text{Suc } 0 < m$ **by** (rule factor-greater-one1)
 moreover from $nmk\ m\ k$ **have** $\text{Suc } 0 < k$ **by** (rule factor-greater-one2)
 ultimately show ?thesis **using** $k\ m\ nmk$ **by** *iprover*
next
 assume $\neg (\exists k < n. \exists m < n. n = m * k)$
 hence $A: \forall k < n. \forall m < n. n \neq m * k$ **by** *iprover*
 have $\forall m\ k. n = m * k \longrightarrow \text{Suc } 0 < m \longrightarrow m = n$
 proof (intro allI impI)
 fix $m\ k$
 assume $nmk: n = m * k$
 assume $m: \text{Suc } 0 < m$
 from $n\ m\ nmk$ **have** $k: 0 < k$
 by (cases k) *auto*
 moreover from n **have** $n: 0 < n$ **by** *simp*
 moreover note m
 moreover from nmk **have** $m * k = n$ **by** *simp*


```

ultimately have kn:  $k < n$  by (rule prod-mn-less-k)
show  $m = n$ 
proof (cases  $k = \text{Suc } 0$ )
  case True
    with nmk show ?thesis by (simp only: mult-Suc-right)
  next
    case False
    from m have  $0 < m$  by simp
    moreover note n
    moreover from False n nmk k have  $\text{Suc } 0 < k$  by auto
    moreover from nmk have  $k * m = n$  by (simp only: mult-ac)
    ultimately have mn:  $m < n$  by (rule prod-mn-less-k)
    with kn A nmk show ?thesis by iprover
  qed
qed
with n have prime n
  by (simp only: prime-eq' One-nat-def simp-thms)
thus ?thesis ..
qed
qed

lemma dvd-factorial:  $0 < m \implies m \leq n \implies m \text{ dvd fact } (n::\text{nat})$ 
proof (induct n rule: nat-induct)
  case 0
    then show ?case by simp
  next
    case (Suc n)
    from  $\langle m \leq \text{Suc } n \rangle$  show ?case
    proof (rule le-SucE)
      assume  $m \leq n$ 
      with  $\langle 0 < m \rangle$  have  $m \text{ dvd fact } n$  by (rule Suc)
      then have  $m \text{ dvd } (\text{fact } n * \text{Suc } n)$  by (rule dvd-mult2)
      then show ?thesis by (simp add: mult-commute)
    next
      assume  $m = \text{Suc } n$ 
      then have  $m \text{ dvd } (\text{fact } n * \text{Suc } n)$ 
        by (auto intro: dvdI simp: mult-ac)
      then show ?thesis by (simp add: mult-commute)
    qed
  qed
qed

lemma dvd-prod [iff]:  $n \text{ dvd } (\text{PROD } m::\text{nat}:\#\text{multiset-of } (n \# ns). m)$ 
  by (simp add: msetprod-Un msetprod-singleton)

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

lemma all-prime-simps:
  all-prime []

```

$all\text{-}prime\ (p \# ps) \longleftrightarrow prime\ p \wedge all\text{-}prime\ ps$
by (*simp-all add: all-prime-def*)

lemma *all-prime-append*:

$all\text{-}prime\ (ps @ qs) \longleftrightarrow all\text{-}prime\ ps \wedge all\text{-}prime\ qs$
by (*simp add: all-prime-def ball-Un*)

lemma *split-all-prime*:

assumes *all-prime ms and all-prime ns*
shows $\exists qs. all\text{-}prime\ qs \wedge (PROD\ m::nat:\#multiset\text{-}of\ qs. m) =$
 $(PROD\ m::nat:\#multiset\text{-}of\ ms. m) * (PROD\ m::nat:\#multiset\text{-}of\ ns. m)$ (**is**
 $\exists qs. ?P\ qs \wedge ?Q\ qs$)
proof –
from *assms* **have** $all\text{-}prime\ (ms @ ns)$
by (*simp add: all-prime-append*)
moreover from *assms* **have** $(PROD\ m::nat:\#multiset\text{-}of\ (ms @ ns). m) =$
 $(PROD\ m::nat:\#multiset\text{-}of\ ms. m) * (PROD\ m::nat:\#multiset\text{-}of\ ns. m)$
by (*simp add: msetprod-Un*)
ultimately have $?P\ (ms @ ns) \wedge ?Q\ (ms @ ns) ..$
then show *?thesis* ..
qed

lemma *all-prime-nempty-g-one*:

assumes *all-prime ps and* $ps \neq []$
shows $Suc\ 0 < (PROD\ m::nat:\#multiset\text{-}of\ ps. m)$
using $\langle ps \neq [] \rangle$ *all-prime ps* **unfolding** *One-nat-def [symmetric]* **by** (*induct ps*
rule: list-nonempty-induct)
(simp-all add: all-prime-simps msetprod-singleton msetprod-Un prime-gt-1-nat
less-1-mult del: One-nat-def)

lemma *factor-exists*: $Suc\ 0 < n \implies (\exists ps. all\text{-}prime\ ps \wedge (PROD\ m::nat:\#multiset\text{-}of\ ps. m) = n)$

proof (*induct n rule: nat-wf-ind*)

case $(1\ n)$

from $\langle Suc\ 0 < n \rangle$

have $(\exists m\ k. Suc\ 0 < m \wedge Suc\ 0 < k \wedge m < n \wedge k < n \wedge n = m * k) \vee prime$

n

by (*rule not-prime-ex-mk*)

then show *?case*

proof

assume $\exists m\ k. Suc\ 0 < m \wedge Suc\ 0 < k \wedge m < n \wedge k < n \wedge n = m * k$

then obtain *m k* **where** *m*: $Suc\ 0 < m$ **and** *k*: $Suc\ 0 < k$ **and** *mn*: $m < n$

and *kn*: $k < n$ **and** *nmk*: $n = m * k$ **by** *iprover*

from *mn* **and** *m* **have** $\exists ps. all\text{-}prime\ ps \wedge (PROD\ m::nat:\#multiset\text{-}of\ ps. m) = m$ **by** (*rule 1*)

then obtain *ps1* **where** *all-prime ps1* **and** *prod-ps1-m*: $(PROD\ m::nat:\#multiset\text{-}of\ ps1. m) = m$

by *iprover*

from *kn* **and** *k* **have** $\exists ps. all\text{-}prime\ ps \wedge (PROD\ m::nat:\#multiset\text{-}of\ ps. m)$

```

= k by (rule 1)
  then obtain ps2 where all-prime ps2 and prod-ps2-k: (PROD m::nat:#multiset-of
ps2. m) = k
    by iprover
  from ⟨all-prime ps1⟩ ⟨all-prime ps2⟩
  have ∃ ps. all-prime ps ∧ (PROD m::nat:#multiset-of ps. m) =
    (PROD m::nat:#multiset-of ps1. m) * (PROD m::nat:#multiset-of ps2. m)
    by (rule split-all-prime)
  with prod-ps1-m prod-ps2-k nmk show ?thesis by simp
next
  assume prime n then have all-prime [n] by (simp add: all-prime-simps)
  moreover have (PROD m::nat:#multiset-of [n]. m) = n by (simp add:
msetprod-singleton)
  ultimately have all-prime [n] ∧ (PROD m::nat:#multiset-of [n]. m) = n ..
  then show ?thesis ..
qed
qed

```

```

lemma prime-factor-exists:
  assumes N: (1::nat) < n
  shows ∃ p. prime p ∧ p dvd n
proof -
  from N obtain ps where all-prime ps
  and prod-ps: n = (PROD m::nat:#multiset-of ps. m) using factor-exists
  by simp iprover
  with N have ps ≠ []
  by (auto simp add: all-prime-nempty-g-one msetprod-empty)
  then obtain p qs where ps: ps = p # qs by (cases ps) simp
  with ⟨all-prime ps⟩ have prime p by (simp add: all-prime-simps)
  moreover from ⟨all-prime ps⟩ ps prod-ps
  have p dvd n by (simp only: dvd-prod)
  ultimately show ?thesis by iprover
qed

```

Euclid's theorem: there are infinitely many primes.

```

lemma Euclid: ∃ p::nat. prime p ∧ n < p
proof -
  let ?k = fact n + 1
  have 1 < fact n + 1 by simp
  then obtain p where prime: prime p and dvd: p dvd ?k using prime-factor-exists
  by iprover
  have n < p
  proof -
    have ¬ p ≤ n
    proof
      assume pn: p ≤ n
      from ⟨prime p⟩ have 0 < p by (rule prime-gt-0-nat)
      then have p dvd fact n using pn by (rule dvd-factorial)
      with dvd have p dvd ?k - fact n by (rule dvd-diff-nat)

```

```

    then have  $p \text{ dvd } 1$  by simp
    with prime show False by auto
  qed
  then show ?thesis by simp
  qed
  with prime show ?thesis by iprover
  qed

```

```

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  $\equiv$ 
 $\lambda x. \text{nat-wf-ind-}P\ x$ 
  ( $\lambda x\ H2.$ 
    case not-prime-ex-mk  $x$  of None  $\Rightarrow [x]$ 
    | 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 ..

```

```

end

```

```

instantiation list :: (type) default
begin

```

```

  definition default = []

```

```

instance ..

```

```

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] by eval
lemma factor-exists 567 = [7, 3, 3, 3, 3] by eval
lemma factor-exists 345 = [23, 5, 3] by eval
lemma factor-exists 999 = [37, 3, 3, 3] by eval
lemma factor-exists 876 = [73, 3, 2, 2] by eval

```

```

lemma iterate 4 Euclid 0 = [2, 3, 7, 71] by eval

consts-code
  default ((error default))

lemma factor-exists 1007 = [53, 19] by evaluation
lemma factor-exists 567 = [7, 3, 3, 3, 3] by evaluation
lemma factor-exists 345 = [23, 5, 3] by evaluation
lemma factor-exists 999 = [37, 3, 3, 3] by evaluation
lemma factor-exists 876 = [73, 3, 2, 2] by evaluation

lemma iterate 4 Euclid 0 = [2, 3, 7, 71] by evaluation

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.