Program Transformation Using PVS[a]

N. Shankar
shankar@csl.sri.com
URL: http://www.csl.sri.com/~shankar/

Computer Science Laboratory
SRI International
Menlo Park, CA

[a] Supported by NSF Grant CCR-930044 and ARPA Contract PR8556.

Overview

- Many clever algorithmic tricks can be systematized as program transformations.
- Do we need special-purpose programs to carry out program transformations?
- Or can these be formalized within a general-purpose deductive framework?
- We show how two basic transformations can be formalized and mechanized using PVS.
  - Bird's Fusion theorem fuses a catamorphism with an anamorphism to eliminate the intermediate data structure.
  - Wand's continuation-based program transformation eliminates tail-recursion in a systematic manner.

Bird's Fusion Theorem

- Complex operations can frequently be defined as the composition $c \circ a$ of two operations c and a.
- Operation a constructs an intermediate data structure.
- Operation c extracts the relevant information by traversing the intermediate data structure.

$$\begin{aligned} \mathit{sort} &= \mathit{flatten} \circ \mathit{mktree} \\ \mathit{isfreevar?}(\mathit{var}) &= \mathit{member}(\mathit{var}) \circ \mathit{freevars} \\ \mathit{squareodds?} &= \mathit{map}(\mathit{square}) \circ \mathit{filter}(\mathit{odd?}) \end{aligned}$$

- Bird's fusion transformation eliminates the intermediate data structure by fusing the composition of a catamorphism with an anamorphism.

What is a Catamorphism?

Recall the PVS declaration of a binary tree.

```
binary_tree[T : TYPE] : DATATYPE
  BEGIN
    leaf : leaf?
    node(val : T,
         left : binary_tree,
         right : binary_tree) : node?
END binary_tree
```

This declaration when typechecked generates various extensionality, accessor/constructor, and induction axioms, and a number of useful recursive combinators.

A *catamorphism* is an operation that can be defined using the generic recursion combinator `reduce`, e.g., *flatten*, *every*.

The Generic Recursion Combinator

```
binary_tree_adt_reduce[T: TYPE, range: TYPE]: THEORY
  BEGIN
  IMPORTING binary_tree_adt[T]

  reduce(leaf?_fun: range,
         node?_fun: [[T, range, range] -> range]):
      [binary_tree[T] -> range] =
    LAMBDA (binary_tree_var: binary_tree[T]):
      CASES binary_tree_var OF
        leaf: leaf?_fun,
        node(node1_var, node2_var, node3_var):
            node?_fun(node1_var,
                      reduce(leaf?_fun,
                             node?_fun)(node2_var),
                      reduce(leaf?_fun,
                             node?_fun)(node3_var))
        ENDCASES
  END binary_tree_adt_reduce
```


What is an Anamorphism?

An anamorphism is a recursive function that produces an instance of a recursive datatype, e.g., *mktree*.

An anamorphism can be defined as unfold which takes a predicate p and a function f:

$$\begin{aligned} & \mathit{unfold}(p, f)(x) \\ & \quad = \quad \mathit{leaf} \text{ if } p(x) \\ & \mathit{unfold}(p, f)(x) \\ & \quad = \quad \mathit{node}(a, \mathit{unfold}(p, f)(y), \mathit{unfold}(p, f)(z)) \\ & \qquad\quad \text{otherwise} \\ & \qquad\quad \text{where } (a, y, z) = f(x) \end{aligned}$$

Bird: *the recursion is not well-defined unless* f *is 'well-founded' in a suitable sense that we will not make precise.*

The `unfold` Anamorphism

```
unfold   [ T, B:  TYPE, size : [B -> nat] ]
                : THEORY
  BEGIN
  IMPORTING binary_tree[T]
  p : VAR PRED[B]
  x, y, z : VAR B
  a : VAR T

   smaller(x) : TYPE = { y | size(y) < size(x)}

   well_fnd(p) : TYPE =
     [x : (p) -> [T, smaller(x), smaller(x)]]

   unfold(p, (f: well_fnd(p)))(x) :
     RECURSIVE  binary_tree  =
    (IF p(x)
      THEN (LET (a, y, z) = f(x)
          IN node(a, unfold(p, f)(y),
                     unfold(p, f)(z)))
       ELSE leaf ENDIF)
   MEASURE size(x)
 END unfold
```

Two TCCs: both are proved automatically.

Fusing Cata- and Ana- Morphisms

```
fusion   [ T, S, Range:  TYPE, size : [S -> nat] ]
  : THEORY
  BEGIN
   IMPORTING unfold[T, S, size]
   p : VAR PRED[S]
   x, y, z : VAR S
   c : VAR Range
   g: VAR [T, Range, Range -> Range]
   a : VAR T

   fun(p, (f : well_fnd(p)), c, g)(x) :
     RECURSIVE  Range =
   (IF p(x)
    THEN (LET (a, y, z) = f(x)
          IN g(a, fun(p, f, c, g)(y),
                  fun(p, f, c, g)(z)))
    ELSE c ENDIF)
   MEASURE size(x)
   ⋮
  END fusion
```

Note dependent type `well_fnd(p)`.

The Fusion Theorem

```
fusion: THEOREM
 (FORALL (p, (f: well_fnd(p)), c, g, x):
    reduce(c, g)(unfold(p, f)(x))
  = fun(p, f, c, g)(x))
```

Proved by

```
(""
 (MEASURE-INDUCT-AND-SIMPLIFY "size(x)" ("x")
  :EXPAND "unfold"
  :IF-MATCH ALL))
```


Fusion Applied to Quicksort

The parameterization for theory `treesort` is the same as for the theory `obt` of ordered binary trees.

The `flatten` operation is defined as a catamorphism.

```
treesort [T: TYPE, <= : (total_order?[T])]: THEORY
 BEGIN
  IMPORTING fusion[T, list[T], list[T], length[T]],
            obt[T, <=]
  A, B, C : VAR binary_tree[T]
  x, y, z : VAR list[T]
  a, b, c : VAR T
  p, q    : VAR PRED[T]

  flatten(A) : list[T] =
             reduce(null[T],
                    (LAMBDA a, x, y:
                       append(x, cons(a, y))))(A)
  ⋮
 END treesort
```


Quicksort : `mktree`

The anamorphism `unjoin` captures the operation of making an ordered binary tree by filtering a given list.

```
below(a)(b): bool = (b <= a)
above(a)(b): bool = NOT (b <= a)

length_filter: LEMMA
   (FORALL (p: PRED[T]):
      length(filter(x, p)) <= length(x))

unjoin: well_fnd(cons?[T]) =
 (LAMBDA (x: (cons?[T])):
    (LET a = car(x),
         y = cdr(x)
      IN
    (a, filter(y, below(a)), filter(y, above(a)))))

mktree(x) : binary_tree[T] =
   unfold(cons?, unjoin)(x)
```


Quicksort by Fusion

The fusion theorem can be used to show

```
flatten(mktree(x)) =
  fun(cons?, unjoin, null[T],
      (LAMBDA a, x, y:
         append(x, cons(a, y))))(x)
```

Quicksort by Fusion

Measure induction can be used to show that the above right-hand side can be given the following recursive definition.

```
quicksort(x): RECURSIVE list[T] =
  (CASES x OF
    null : null,
    cons(a, y) :
      append(quicksort(filter(y, below(a))),
             cons(a, quicksort(filter(y, above(a)))))
   ENDCASES)
 MEASURE length(x)

quicksort_by_fusion: THEOREM
  quicksort(x) = flatten(mktree(x))
```


Fusion: Summary

- We have seen how Bird's fusion theorem can be formalized abstractly and elegantly using subtype and dependent type constructions.
- The related mechanical proofs are trivial.
- We cannot prove the fusion theorem generically for all recursive datatypes: this would be a metatheorem for PVS.

Wand's Continuation Based Program Transformation

How do you systematically transform the quadratic non-tail-recursive

```
reverse(l): RECURSIVE list[T] =
  CASES l OF
   null: l,
   cons(x, y): append(reverse(y), cons(x, null))
  ENDCASES
  MEASURE length
```

to the linear tail-recursive

```
reva(l, w): RECURSIVE list[T] =
  CASES l OF
    null: w
    cons(x, y):
      reva(y, cons(x, w))
  ENDCASES
  MEASURE length
```


Using Continuations

The non-tail-recursive computation can be captured as a *continuation* argument.

```
revc(l, f): RECURSIVE list[T] =
  CASES l OF
    null: f(l),
    cons(x, y):
      revc(y, (LAMBDA u:
                 f(append(u, cons(x, null)))))
  ENDCASES
  MEASURE length
```

Associative continuation-builder `append` can be replaced by an accumulator to derive `reva`.

Note

$$\begin{aligned} \mathtt{revc}(\mathtt{l},\mathtt{f}) &= \mathtt{f}(\mathtt{rev}(\mathtt{l})) \\ \mathtt{reva}(\mathtt{l},\mathtt{w}) &= \mathtt{revc}(\mathtt{l},(\mathtt{LAMBDA\ u:\ append}(\mathtt{u},\mathtt{w}))) \end{aligned}$$

Continuation-based Transformation in PVS

The theory parameters schematically capture a linear non-tail-recursive definition.

```
wand [dom, rng: TYPE,      %function domain, range
      a: [dom -> rng],     %base case function
      d: [dom-> rng],      %recursion parameter
      b: [rng, rng -> rng],%continuation builder
      c: [dom -> dom],     %recursion destructor
      p: PRED[dom],        %branch predicate
      m: [dom -> nat],     %termination measure
      F : [dom -> rng]]    %tail-recursive function
  : THEORY
  BEGIN
  :
  END wand
```


Assumptions on `wand` Parameters

The parameters should include a well-founded termination measure `m`, an associative continuation builder `b`, and a recursive scheme defining `F`.

```
ASSUMING  %3 assumptions: b associative,
          % c decreases measure, and
          % F defined recursively
          %  using p, a, b, c, d.
  u, v, w: VAR rng
 assoc: ASSUMPTION b(b(u, v), w) = b(u, b(v, w))

 x, y, z: VAR dom

 wf : ASSUMPTION NOT p(x) IMPLIES m(c(x)) < m(x)

 F_def: ASSUMPTION
  F(x) =
 (IF p(x) THEN a(x) ELSE b(F(c(x)), d(x)) ENDIF)
ENDASSUMING
```


`wand` Definitions/Theorems

`FC` turns the continuation builder `b` in `F_def` into a continuation argument by λ-abstracting out the recursive call.

```
f: VAR [rng -> rng]

FC(x, f) : RECURSIVE rng =
  (IF p(x)
     THEN f(a(x))
    ELSE FC(c(x), (LAMBDA u: f(b(u, d(x)))))
   ENDIF)
MEASURE m(x)

FFC: LEMMA FC(x, f) = f(F(x))
```

Proved automatically by measure induction.

`wand` Definitions/Theorems

`FA` turns the continuation `f` into an accumulator `u` by applying the associativity of `b`.

```
FA(x, u): RECURSIVE rng =
 (IF p(x)
    THEN b(a(x), u)
   ELSE FA(c(x), b(d(x), u)) ENDIF)
 MEASURE m(x)

FAFC: LEMMA
  FA(x, u) = FC(x, (LAMBDA w: b(w, u)))
```

Also proved by measure induction.

A Transformation Instance: `reverse`

Theory `wand` generates a tail-recursive variant of `reverse`. All TCCs can be trivially proved using PVS.

```
reverse [T : TYPE]: THEORY
 BEGIN
  CONVERSION
   extend[list[T], (cons?[T]), list[T], null[T]]

   reverse_wand: THEORY =
      wand[list[T], list[T], id[list[T]],
            (LAMBDA (x: (cons?[T])):
               cons(car(x), null)),
             append[T], cdr[T],
             null?[T], length[T], reverse[T]]

  u, x, y, z: VAR list[T]

%invariant relating tail-recursive FA and reverse
 tail_reverse: LEMMA
    FA(x, u) = append(reverse(x), u)
 END reverse
```


Transforming Binary Recursions

The same results can be transferred to tree-like recursion schemes.

```
binary [dom, rng: TYPE,       %function domain, range
        a: [dom -> rng],      %base case function
        b: [rng, rng -> rng], %continuation builder
        l: [dom -> dom],      %recursion destructor
        r: [dom -> dom],      %recursion destructor
        p: PRED[dom],         %branch predicate
        m: [dom -> nat],      %termination measure
        F : [dom -> rng]]     %non-tail-recursive input
  : THEORY
  BEGIN
 ASSUMING
  :
  ENDASSUMING
  :
  END binary
```


Assumptions on Binary Recursion Scheme

The assumptions are similar to `wand`.

```
  u, v, w: VAR rng

  assoc: ASSUMPTION b(b(u, v), w) = b(u, b(v, w))

  x, y, z: VAR dom

  wfl : ASSUMPTION NOT p(x) IMPLIES m(l(x)) < m(x)

  wfr : ASSUMPTION NOT p(x) IMPLIES m(r(x)) < m(x)

  mpos: ASSUMPTION m(x) > 0

  m_left_right: ASSUMPTION
    NOT p(x) IMPLIES m(x) > m(l(x)) + m(r(x))

  F_def: ASSUMPTION
   F(x) =
   (IF p(x) THEN a(x) ELSE b(F(l(x)), F(r(x))) ENDIF)
```


Introducing the Continuation

The `FFC` lemma here has an extra hypothesis which is easily satisfied by the identity continuation.

```
  f: VAR [rng -> rng]

  FC(x, f) : RECURSIVE rng =
    (IF p(x)
       THEN f(a(x))
      ELSE FC(r(x), (LAMBDA u: b(FC(l(x), f), u)))
     ENDIF)
  MEASURE m(x)

  FFC: LEMMA
   (FORALL u, v: f(b(u, v)) = b(f(u), v))
      IMPLIES FC(x, f) = f(F(x))
```

Introducing the Accumulator

FA transforms the continuation in FC into an accumulator.

The proofs are straightforward.

```
FA(x, v): RECURSIVE rng =
 (IF p(x)
    THEN b(v, a(x))
   ELSE FA(r(x), FA(l(x), v)) ENDIF)
 MEASURE m(x)

FAFC: LEMMA
  FA(x, v) = FC(x, (LAMBDA w: b(v, w)))
```

FA is still not tail-recursive; this can be fixed by adding a stack argument for the right recursive calls.

Eliminating Nested Recursion

```
X, Y: VAR list[dom]

mlist(X): RECURSIVE nat =
  (IF cons?(X)
     THEN m(car(X)) + mlist(cdr(X))
    ELSE 0 ENDIF)
 MEASURE length(X)

FI(x, v, Y): RECURSIVE rng =
 (IF p(x)
   THEN (IF cons?(Y)
           THEN FI(car(Y), b(v, a(x)), cdr(Y))
          ELSE b(v, a(x))
         ENDIF)
  ELSE FI(l(x), v, cons(r(x), Y))
  ENDIF)
 MEASURE (m(x) + mlist(Y))

FIFA: LEMMA
   FI(x, v, Y) =
    (IF cons?(Y)
       THEN FI(car(Y), FA(x, v), cdr(Y))
      ELSE FA(x, v)
     ENDIF)
```


Conclusions

- There is nothing syntactic about program transformation.
- A suitably expressive specification language can capture these transformations at a fairly abstract level.
- A general-purpose proof checker can easily handle the deductive aspects of program transformation.
- Many “difficult” induction proofs for tail-recursion optimization are actually trivial instances of continuation-based transformation.
- No fancy induction heuristics are required for dealing with these proofs.