MATHEMATICAL ENGINEERING
TECHNICAL REPORTS

Reversing Iterations: IO Swapping Leads You There And Back Again

Akimasa Morihata, Kazuhiko Kakehi,
Zhenjiang Hu, and Masato Takeichi

METR 2005–11 May 2005

DEPARTMENT OF MATHEMATICAL INFORMATICS
GRADUATE SCHOOL OF INFORMATION SCIENCE AND TECHNOLOGY
THE UNIVERSITY OF TOKYO
BUNKYO-KU, TOKYO 113-8656, JAPAN

WWW page: http://www.i.u-tokyo.ac.jp/mi/mi-e.htm

Reversing Iterations: IO Swapping Leads You There And Back Again

Akimasa Morihata, Kazuhiko Kakehi, Zhenjiang Hu, and Masato Takeichi

Department of Mathematical Informatics
Graduate School of Information Science and Technology
University of Tokyo
{Akimasa_Morihata, kaz, hu, takechi}@mist.i.u-tokyo.ac.jp

May, 2005

Abstract

TABA (There And Back Again) [DG02], proposed by Danvy and Goldberg, is a special but powerful programming pattern where a recursive function traverses lists at return time. They showed the TABA programs, but neither derivation nor manipulation of the TABA programs were presented. We propose a novel program transformation rule called *IO swapping*. The rule swaps input and output values of functions and introduces iteration at return times. Using this rule three stories of TABA are played out in this paper: systematic derivation of a basic TABA program; manipulation of TABA programs including palindrome through function fusion; and extension of the rule to deal with circular dependency of inputs and outputs like `repmin` function, introduced by Bird [Bir84], and with structures other than lists like binary trees.

1 Introduction

TABA ("There And Back Again") [DG02], proposed by Danvy and Goldberg, is a special but powerful programming pattern where a recursive function traverses lists at return time. Their idea is that the recursive calls get us there (typically to a empty list) and the returns get us back again while traversing the list. A typical example is the symbolic convolution function `cnv` which accepts two lists, $[x_0, x_1, \ldots, x_n]$ and $[y_0, y_1, \ldots, y_n]$, and computes a new list $[(x_0, y_n), (x_1, y_{n-1}), \ldots, (x_n, y_0)]$. This can be naively specified as follows.

```
cnv :: [a] -> [b] -> [(a,b)]
cnv x y = zip x (reverse y)
```

Being straightforward, this definition is not satisfactory: The list `y` is traversed by `reverse` to produce an intermediate list which will be again traversed by `zip`. A clever TABA program, which avoids generation of the intermediate list, is as follows.

```
cnv x y = let (r,[]) = walk x in r
    where walk [] = ([],y)
          walk (a:x') = let (r,b:y') = walk x'
                        in ((a,b):r,y')
```

This program uses an auxiliary function `walk` which exhibits a bit unusual behavior. When the input `x` is empty, `walk` uses the input `y` directly as a return value, and his return value will be traversed together while traversing `x`. Indeed this program is much different from the initial specification, but it actually computes symbolic convolution without the need of extra memory other than resulting data. Other examples such as the palindrome program in Section 4 may be more complex.

TABA is truly tricky. It would be interesting to see whether there is a systematic way that may *lead* us to write such TABA programs. A set of clever TABA programs Danvy and Goldberg [DG02] gave is not satisfactory: They showed the TABA programs, but neither derivation nor manipulation of the TABA programs were presented. It seems like a *still picture*; we enjoy it, but we could not know *where it came from, what it is, and where it will go*. One may wish to use TABA-like computations and to manipulate such a new kind of iteration, i.e., iteration over some return values.

In this paper, we show that we can make this still picture to a *motion picture* by program calculation [BdM96], an transformational approach to carrying programs from naive definitions to efficient ones. Our *TABA Trilogy* has the following three *plots*, which are the main contributions of this paper.

- We propose a novel program transformation rule called *IO swapping*, which swaps input and output values of a function and introduces iteration at return times. We show it enables systematic derivation of a function `cnv` by program calculation.

- We demonstrate how to manipulate TABA programs. Function `cnv` is a generic function and we can get variety of functions by fusing other functions to `cnv`. We demonstrate the derivation of efficient palindrome-detecting function and confirm this.

- We extend IO swapping rule and give more and more power. Extended IO swapping gives a constraint to be able to manipulate functions which has circular dependency of inputs and outputs. IO swapping jumps out of lists: We show that IO swapping can deal not only lists, but also other data structures, for example binary tree.

Along with this introduction, the next Section 2, which briefly reviews notations and explains a known fusion law used in our derivation, forms the *prologue*. Section 3 figures out *where TABA came from*, by introducing a novel program transformation rule named IO swapping for deriving the standard TABA form. Section 4 portrays *what TABA is* through manipulation of TABA programs. Section 5 foresees *where TABA will go*, by generalizing the IO swapping rule and show manipulations of more confusing functions. As the *epilogue*, we discuss about related work in Section 6, and conclude the TABA story in Section 7.

2 Preliminary

In this section, we briefly explain our notational conventions and an important and general law for program derivation.

2.1 Notations

Throughout the paper we use the notation of the functional programming language Haskell [Bir98]. Some syntactic notations we use in this paper are as follows. The symbol \ is used instead of λ for λ-expressions, and the identity function, for example, is written as `(\x -> x)`. The symbol `.` denotes function composition, i.e., `(f.g) x = f(g x)`. In this paper, we use many standard Haskell functions, whose informal definitions are given in Figure 1. We also assume that the structured data we are treating are finite.

2.2 Fusion Law

Functional programming languages provide a constructive way of programming, namely development of larger programs through composition of smaller and simpler functions. To improve efficiency of such compositional programming style, function fusion plays an important role, which fuses function composition into a single function and eliminates intermediate data structures passed between them. In this paper we will make an intensive use of the following fusion law [Bir89].

```
id x = x
fst (a,b) = a
snd (a,b) = b
head [x0,x1,...,xn] = x0
tail [x0,x1,...,xn] = [x1,x2,...,xn]
[x0,x1,...,xm,...,xn] !!  m = xm
take m [x0,x1,...,xm,...,xn] = [x0,x1,...,xm−1]
drop m [x0,x1,...,xm,...,xn] = [xm,xm+1,...,xn]
length [x0,x1,...,xn] = n+1
reverse [x0,x1,...,xn] = [xn,xn−1,...,x0]
map f [x0,x1,...,xn] = [f x0,f x1,...,f xn]
zip [x0,x1,...,xn] [y0,y1,...,yn] = [(x0,y0),(x1,y1),...,(xn,yn)]
foldr f e [x0,x1,...,xn] = f x0 (f x1 (··· (f xn e)···))
foldl f e [x0,x1,...,xn] = f (···(f (f e x0) x1)···) xn
and x = foldr (&&) True x
div n m = ⌊n/m⌋
```

Figure 1: Informal definitions of standard functions

theorem 1 (Fold Promotion)
`f . foldr (⊕) e = foldr (⊗) e'`
provided that `f e = e'` and `f (a ⊕ y) = a ⊗ (f y)` hold for all `a` and `y`.
□

This theorem indicates that finding a proper operator $\otimes$ is enough for fusing programs. Such calculation over programs, which is often referred as *calculational programming* [BdM96] (or *program calculation*) is a powerful tool, as we later see TABA-styled functions can be formally derived using calculational programming.

3 Deriving TABA Programs

We shall show that the TABA program for the symbolic convolution function `cnv` can be systematically derived, based on the standard fusion transformation together with a new transformation called *IO swapping*.

3.1 IO Swapping for `foldl`

The new and effective transformation rule proposed in this paper is *IO swapping*, which changes the view of functions; literally "thinking upside down" about treatments of data structure. We start by seeing how the familiar accumulative function `foldl` defined by

```
foldl f e [] = e
foldl f e (a:x) = foldl f (f e a) x
```

can be turned into an accumulation-free variant by IO swapping. Though it is often the case that elimination of accumulation involves changes in semantics and complexity, the obtained program through IO swapping keeps the meaning while avoiding degradation of computational complexity.

Our idea is to move some information from the input to the output so that accumulative computation on the input can be replaced by reconstruction of the output. To do so, we introduce the following function `foldl'`

```
foldl' f e y x = (drop #y x, foldl f e (take #y x))
```

which uses an additional parameter `y` to control how much of the input `x` of `foldl` to be moved to the output (the first component of the result of `foldl'`). Here `#y` denotes the length of `y`. It is easy to recognize the relation between `foldl` and `foldl'`.

```
foldl f e x = let ([],r) = foldl' f e x x in r
```

Now we can derive an efficient program for `foldl'` by the following calculation.

```
foldl' f e [] x ⇒ (x,e)

foldl' f e (a:y) x
     = (drop #(a:y) x, foldl f e (take #(a:y) x))
     ⇒   { by definition of take, drop and (#) }
       (tail (drop #y x), foldl f e ((take #y x)++[x!!#y]))
     ⇒   { by a lemma: foldl f e (y++[b]) = f (foldl f e y) b }
       (tail (drop #y x), f (foldl f e (take #y x)) (x!!#y))
     ⇒   { by a lemma: head (drop n x) = x !! n for 0 ≤ n < #x }
       (tail (drop #y x), f (foldl f e (take #y x)) (head (drop #y x)))
     ⇒   { by definition: foldl' f e y x = (drop #y x, foldl f e (take #y x)) }
       let (b:z,r) = foldl' f e y x in (z, f r b)
```

We exploit some lemmas about `take`, `drop`, `(#)`, `foldl` during the above calculation. These properties are easily proved by induction (therefore their proofs are omitted). Now that a new definition of `foldl` is in hand, we call the new `foldl` as `foldl_n` to be distinguishable from ordinary `foldl`, which is summarized as follows.

```
foldl_n f e x = let ([],r) = foldl' f e x x in r
    where foldl' f e [] x = (x, e)
          foldl' f e (a:y) x = let ((b:z), r) = foldl' f e y x
                               in (z, f r b)
```

This implementation of `foldl'` has redundant variables. We can see that the first, the second and the forth arguments (namely `f`, `e` and `x`, respectively) are kept unchanging during all computation steps. The third argument is necessary for pattern-matching, though its value is not reflected for computing the resulting value. Deleting these unchanged arguments and renaming some variables gives the following concise definition.

```
foldl_n f e x = let ([],r) = foldl' x in r
    where foldl' [] = (x, e)
          foldl' (b:y) = let ((a:x'), h) = foldl' y
                         in (x', f h a)
```

We summarize these results of derivation as the following theorem.

theorem 2 (IO Swapping for `foldl`)
The functions `foldl` and `foldl_n` are defined as follows.

```
foldl f e [] = e
foldl f e (a:x) = foldl f (f e a) x

foldl_n f e x = let ([],r) = foldl' x in r
    where foldl' [] = (x, e)
          foldl' (b:y) = let ((a:x'), r') = foldl' y
                         in (x', f r' a)
```

Then for all `x`, `f`, and `e`, `foldl` and `foldl_n` are equivalent.
□

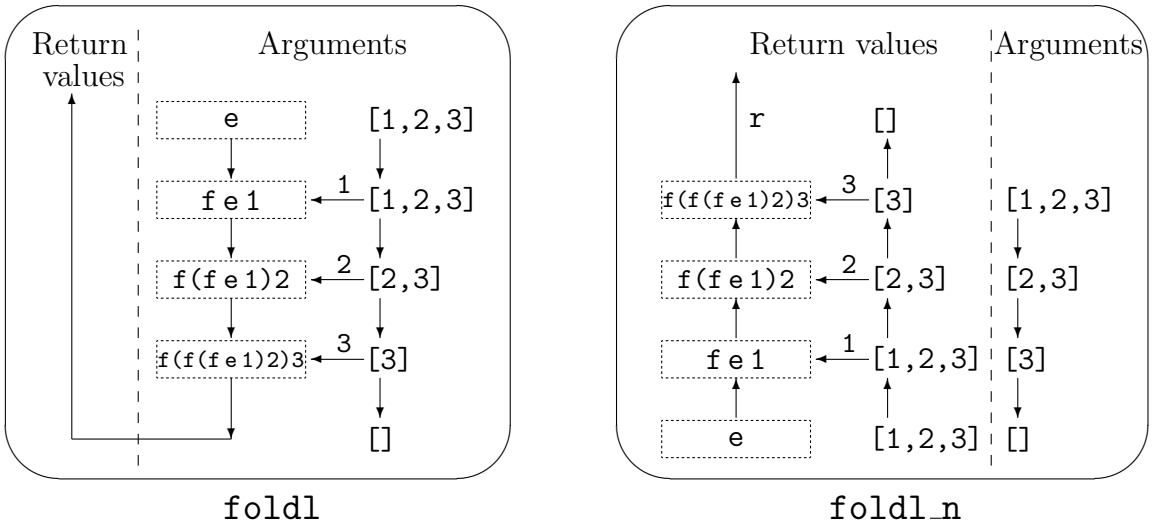

Figure 2: The models of computation processes of `foldl` and `foldl_n`

In function `foldl'` the initial input list `x` of `foldl_n` is passed directly as the return value of the termination condition, and it performs its destruction through pattern-matching in its recursive call. It is worth noting that it shares with TABA the way to manipulate the given list, namely in a reversed manner; indeed we are ready derive TABA functions and this will be demonstrated later.

Several remarks are worth making on the implications of this theorem. First pay attention to how the result is computed using the function parameter `f`. While `f` is applied to the accumulation parameter in the function `foldl`, it comes to the surface to compute the return value of `foldl_n`. Here arises a hypothesis: That IO swapping is a rule that produces a kind of inverted function which swaps the inputs (arguments) and the outputs (return values) of the original function. This is exactly the essence of IO swapping. If the input list comes syntactically to the position as the output, consumption of lists in the return value is a natural consequence.

Figure 2 illustrates the computation process of `foldl` and `foldl_n`. Comparing two figures carefully, the idea of IO swapping becomes much more obvious: Turning over the the figure of `foldl` looks almost the same as that of `foldl_n`! It is possible to liken the input list as a tower where each floor stores one value except for ground floor holding the special value `[]`. The King lives at the top of this tower; according to the King's command to gather values, a servant goes from the top downward to the ground floor (like the argument) or another servant goes from the ground floor upward to the top (like the return value). If the King rearranges the values upside-down behind the curtain, what servants gather up as the result are secretly exchanged. Such a rearrangement of values can take place by transferring consumption of the list from the argument to the return value: A return value arranges the values in the list, from the ground floor to the top, providing the reversed, upside-down order of values.

Have a look again into the definition of `foldl_n`. We can see that `foldl_n` (and its auxiliary function `foldl'`) has no accumulation parameters. To resolve this mystery recall the behavior of IO swapping, namely to flip the inputs and the outputs. Usual `foldl` has computation on accumulation parameter and has no computation on its return value. Swapping the argument and the return value of the ordinary `foldl` we get an unusual form of `foldl` (that is `foldl_n`) which has computation on the return value and has no accumulation parameters. Since what is taking place is just flipping, the computational complexity of `foldl_n` and `foldl` is the same.

3.2 Deriving TABA Programs by IO Swapping and Fusion

Now we are going to show how to systematically derive the TABA program for `cnv` in the introduction, starting from the following straightforward specification:

```
cnv x y = zip x (reverse y)
    where reverse = foldl (\y a -> a:y) []
```

We assume that `x` and `y` have the same length.

Our derivation strategy basically consists of two steps: (1) deriving a TABA program for smaller functions used in the specification, and (2) deriving bigger a TABA program by promotion a function into a smaller TABA program. For our case of `cnv`, we first derive a TABA program for `reverse`. Because function `reverse` is a instance of `foldl` we can apply IO swapping to `reverse` and get the following program:

```
rev_n x = let ([],r) = rev' x in r
    where rev' [] = (x,[])
          rev' (b:y) = let (a:x',r') = rev' y
                       in (x',a:r')
```

which can be described in terms of `foldr` for being suitable for later fusion transformation.

```
rev_n x = snd (foldr (\b (a:x',r')->(x',a:r')) (x,[]) x)
```

Now we calculate a TABA program for `cnv` by promoting the functions into `rev_n`.

```
cnv x y = zip x (rev_n y)
        ⇒ zip x (snd (foldr (\b (a:x',r')->(x',a:r')) (y,[]) y))
        ⇒ snd (id_zip (foldr (\b (a:x',r')->(x',a:r')) (y,[]) y) x)
              where id_zip (a,y) x = (a, zip x y)
```

To promote `id_zip` into `foldr` in the above, we check the following two conditions to apply the fusion law (Theorem 1).

```
id_zip (y,[]) x ⇒ (y,[])
id_zip ((\b (a:x',r')->(x',a:r')) b (a:x',r')) x
     ⇒ (x', (head x,a):zip (tail x) r')
     ⇒ step b (id_zip (a:x',r')) x
           where step b r x = let (a:x', r') = r (tail x)
                              in (x', (head x,a):r')
```

Therefore, the fusion transformation gives

```
cnv x y = snd (foldr step (\x->(y,[])) y x)
```

which is actually the following program after unfolding the `foldr`.

```
cnv x y = snd (cnv' y x)
    where cnv' [] = \x->(y,[])
          cnv' (b:y) = \x->let (a:x',r') = cnv' y (tail x)
                           in (x',(head x,a):r')
```

Finally, we make the program more concise with some known calculations. First, applying eta-expansion to remove function values and using the assumption, length of `x` and `y` are same, yields the following program.

```
cnv x y = let ([],r) = cnv' y x in r
    where cnv' [] [] = (y,[])
          cnv' (b:y) (d:z) = let (a:x',r') = cnv' y z
                             in (x',(d,a):r')
```

Then, we eliminate constant propagation; during all computation steps, `cnv'` does not use the first argument and always succeeds in pattern-matching, and we can eliminate it.

```
cnv x y = let ([],r) = cnv' x in r
    where cnv' [] = (y,[])
          cnv' (d:z) = let (a:x',r') = cnv' z
                       in (x',(d,a):r')
```

This is what we have seen in the introduction, the efficient TABA program for `cnv`.

4 Manipulating TABA Programs

In this section, we illustrate more on manipulability of TABA programs, by deriving the efficient palindrome program as proposed by Danvy and Goldberg [DG02]. We show that `cnv`, a typical TABA program, can be fused with other functions in many ways resulting in a more involved TABA program. This shows that TABA programs are of high generality, and that our method can be of useful guidance for developing TABA programs.

4.1 A Small Example

In Section 3.2, we have shown the derivation of a TABA program for `cnv`. In fact, `cnv` captures a typical form of TABA programs, and can be used to write various kinds of functions. It should be interesting to see more about how to derive a new TABA programs from smaller ones. Our strategy of manipulation of the TABA function is to exploit function fusion—fusing functions to `cnv` to produce new TABA programs.

Recall the function `rev_n`, which is obtained by IO swapping.

```
rev_n x = let ([],r) = rev' x in r
    where rev' [] = (x,[])
          rev' (b:y) = let (a:x',r') = rev' y
                       in (x',a:r')
```

We may choose another way to get the above definition, with an assumption that we have in hand the TABA program for `cnv`. Let us derive `rev_n` from `cnv`. From the definition

```
cnv x y = zip x (reverse y)
```

we extract `reverse x` as follows.

```
reverse x ⇒ map snd (zip x (reverse x))
          ⇒ map snd (cnv x x)
```

We use this equation as our specification of `reverse`, and try to fuse `map` with `cnv` to obtain `rev_n`.

```
reverse x = map snd (cnv x x)
          ⇒ map snd (snd (foldr (\a (b:y',r)->(y',(a,b):r)) (x,[]) x))
          ⇒ snd (id_map snd (foldr (\a (b:y',r)->(y',(a,b):r)) (x,[]) x))
                where id_map f (a,b) = (a, map f b)
```

Now we apply Theorem 1 to fuse the above `id_map` with `foldr`. With checking the following conditions

```
reverse x ⇒ snd (foldr(\a (b:y',r)->(y',b:r)) (x,[]) x)

  (id_map snd) (x,[]) ⇒ (x,[])
  (id_map snd) ((\a (b:y',r)->(y',(a,b):r)) a (b:y r)
      ⇒ (y', b:map snd r)
      ⇒ (\a (b:y',r)->(y',b:r)) ((id_map snd) ((b:y'),r))
```

we get

```
reverse x = snd (foldr (\a (b:y',r)->(y',b:r)) (x,[]) x)
```

which is exactly `rev_n` in the form of `foldr`.

This process indicates that derivation of TABA programs through specification using `cnv` and function fusion is a successful approach. In the next subsection we develop a much more complicated example of palindrome.

4.2 Palindrome

Danvy and Goldberg [DG02] riddle in the beginning of their paper: "Given a list of length n, where n is not known in advance, determine whether this list is palindrome in $\lceil n/2 \rceil$ recursive calls and with no auxiliary list." Let us see this riddle is solved with calculation.

We may start by solving the problem in a straightforward way without being concerned with its efficiency. We check whether a list is palindrome or not by turning up from the latter half center of the list, zipping it with the first half, and checking whether all elements are the same.

```
palindrome x
    = and (map (\(a,b)->a==b) (zip (take (div (length x) 2) x)
                                   (reverse (drop (div (length x) 2) x))))
```

Here, for simplicity we assume the length of the list is even. Replacing the `zip-reverse` pattern with `cnv` gives

```
palindrome x
    = and (map (\(a,b)->a==b) (cnv (take (div (length x) 2) x)
                                   (drop (div (length x) 2) x)))
```

Now the problem is how to manipulate `cnv`, which is not trivial because we have to fuse functions from both front and back of `cnv`. To manipulate `cnv`, Danvy and Goldberg [DG02] proposed a theorem similar to the warm fusion law [LS95], but their theorem cannot cope with this problem. It is nice to see later that the existing program calculation techniques are enough here.

Our derivation of an efficient program for `palindrome` consists of the following three main steps.

1. Define the following functions to extract subexpressions in the definition of `palindrome`:

   ```
   alleq = and.(map (\(a,b)->a==b))
   takehalf x = take (div (length x) 2) x
   drophalf x = drop (div (length x) 2) x
   ```

   and derive efficient definitions for them by fusion transformation. Since this derivation is not special, we give the results only. If you want to know the detailed calculation, see Appendix.

   ```
   alleq ⇒ foldr (\(a,b) r->a==b && r) True
   takehalf x ⇒ foldr' (\a r x->head x:r (tail x)) (\x->[]) x x
   drophalf x ⇒ foldr' (\a r x->r (tail x)) id x x
   ```

   Here `foldr'` is defined below, being equipped with the same fusion law as `foldr` [HIT96].

   ```
   foldr' f e [] = e
   foldr' f e (a:b:x) = f (a,b) (foldr' f e x)
   ```

2. Apply fusion transformation to merge functions with `cnv` from both front and back. Here gives a big picture of the fusion calculation.

```
palindrome x
      = alleq (cnv (takehalf x) (drophalf x))
      ⇒  { TABA form for cnv }
        alleq (snd (foldr (\a (b:y',r)->(y',(a,b):r))
                          (drophalf x, []) (takehalf x)))
      ⇒  { swap alleq and snd by defining id_alleq (a,x) = (a, alleq x) }
        snd (id_alleq (foldr (\a (b:y',r)->(y',(a,b):r))
                             (drophalf x, []) (takehalf x)))
      ⇒  { fuse the underlined part, similar to what we did in Section 4.1 }
        snd (foldr (\a (b:y',r')->(y',a==b && r'))
                   (drophalf x, True) (takehalf x)))
      ⇒  { define alleqcnv x y = foldr (\a (b:y',r')->(y',a==b && r')) (x,True) y }
        snd (alleqcnv (takehalf x) (drophalf x))
      ⇒  { by the efficient definition for takehalf }
        snd (alleqcnv (foldr' (\a r x->head x:r (tail x)) (\x->[]) x x)
                      (drophalf x))
      ⇒  { fuse the underlined composition }
        snd (foldr' (\a r x y-> let (b:y,r') = r (tail x) y
                                in (y', head x==b && r'))
                    (\x y->(y,True)) x x (drophalf x))
      ⇒  { by the efficient definition for drophalf }
        snd (foldr' (\a r x y-> let (b:y,r') = r (tail x) y
                                in (y', head x==b && r'))
                    (\x y->(y,True)) x x (foldr' (\a r x->r (tail x)) id x x))
```

3. Apply the tupling transformation [Bir84] [HITT97] to avoid twice traversals of the same data structure `x` by `foldr'` as underlined above.

```
palindrome x
    = let
      (y',([],r)) = foldr' ((\a r x' x y->
                               let (y',(b:y,r')) = r (tail x') (tail x) y
                               in (y',(y, head x==b && r'))
                            (\y' x y->(y',(y,True))) x x) y'
      in r
```

To enhance readability, we unfold the definition of `foldr'`.

```
palindrome x = let (y',([],r)) = pld x x x y' in r
   where
      pld (a1:b1:x1) (a2:x2) (a3:x3) y = let (y',(b:y,r')) = pld x1 x2 x3 y
                                         in (y',(y, a3==b && r'))
      pld [] y' x' y = (y',(y,True))
```

This program has circular data dependency between output and input as underlined above. But this is not a problem; we can eliminate the first element of return value and the forth argument of `pld` because they do not change during the whole computation steps of palindrome, and this pseudo-dependency is thus eliminated.

```
palindrome x = let ([],r) = pld x x x in r
    where pld (a1:b1:x1) (a2:x2) (a3:x3) = let (b:y,r') = pld x1 x2 x3
                                           in (y, a3==b && r')
          pld [] y' x' = (y',True)
```

Noticing that the fact that the second and the third arguments of `pld` are always the same, we get the final program.

```
palindrome x = let ([],r) = pld x x in r
    where pld (a1:b1:x1) (a2:x2) = let (b:y,r') = pld x1 x2
                                   in (y, a2==b && r')
          pld [] y' = (y',True)
```

Our final program is essentially the same as the efficient palindrome detecting function of Danvy and Goldberg [DG02]. Herewith we have solved their riddle.

5 TABA, and Further More!

So far, we have proposed IO swapping for `foldl` in Section 3.1 and illustrate its use in derivation of TABA programs in Section 3.2. In this section, we reinforce the power and generality of IO swapping. A more general IO swapping will be given to deal with wider class of functions, which may even have circular dependency between inputs and outputs. We demonstrate the power of IO swapping by manipulating the known `repmin` function [Bir84]. As far as we are aware, no useful method has been proposed to manipulate circular programs like `repmin`. And furthermore, we generalize IO swapping from lists to other data structures such as binary trees.

5.1 IO Swapping

The following theorem is a generalization of Theorem 2, being able to deal with a more general and more powerful recursive function `f1` that not only uses an accumulation parameter but also may contain complicated circular data dependency between the output and input.

theorem 3 (IO Swapping)
The following two functions `f1` and `f2` are equivalent.

```
f1 x h0 = let r = f1' x (g3 r h0) in r
    where f1' [] h = g0 h
          f1' (a:x') h = let r = f1' x' (g2 a r h)
                         in g1 a r h

f2 x h0 = let ([], h, r') = f2' (x, g0 h) in r'
    where f2' ([], r) = (x, g3 r h0, r)
          f2' (b:y, r) = let (a:x', h, r') = f2' (y, g1 a r h)
                         in (x', g2 a r h, r')
```

proof sketch
To prove this theorem, we define following function `f1_`.

```
f1_ x h0 = let ([], r) = f1_' (x, g3 r h0) in r
    where f1_' ([], h) = (x, g0 h)
          f1_' (a:x', h) = let (b:y,r) = f1_' (x', g2 a r h)
                           in (y, g1 a r h)
```

It is obvious that function `f1_` computes the same value as `f1`.

We define the set $S_{\{\texttt{f1\_,x,h0}\}}$. $S_{\{\texttt{f1,x,h0}\}}$ contains all pair of arguments and return values of `f1_'` which appears in the whole reduction step of `f1_ x h0`. Similarly, we define the set $S_{\{\texttt{f2,x,h0}\}}$ which contains all pair of arguments and return values of `f2'` which appears in the whole reduction step of `f2 x h0`. Then following three lemmas hold.

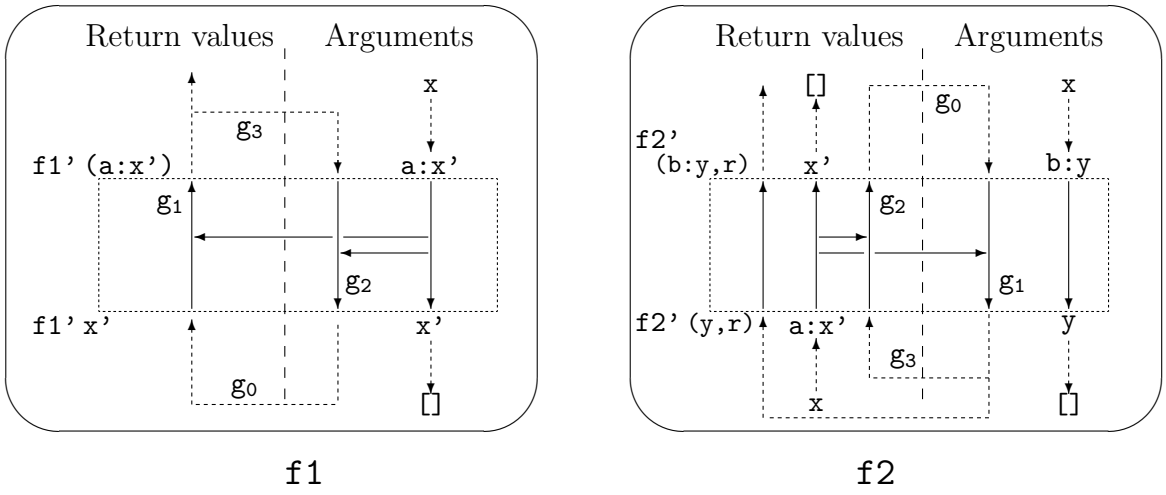

Figure 3: The model of computation process of `f1` and `f2`

1. For all `(a1,b1)`, `(a2,b2)` $\in S_{\{\texttt{f1\_,x,h}_0\}}$, `a1`=`a2` iff `b1`=`b2`.
2. `((x',h),(y,r))` $\in S_{\{\texttt{f1\_,x,h}_0\}}$ iff `((y,r),(x',h,r'))` $\in S_{\{\texttt{f2,x,h}_0\}}$.
3. If `((x,h`$_0$`),([],r))` $\in S_{\{\texttt{f1\_,x,h}_0\}}$, for all `((y,r''),(x',h,r'))` $\in S_{\{\texttt{f2,x,h}_0\}}$, `r'`=`r`.

Proving lemma 1 is easy. We can see that not only first element of argument but also or return value of `f1_'` specifies the depth of recursive calls.

Proving lemma 2 is a bit difficult. Lemma 1 implies function `f1_'` has its inverse function. Inverse function of `f1_'` is almost the same as `f2'`, except for the third element of return value of `f2'`. Induction with a care of this fact lets lemma 2 proved.

Lemma 2 prove lemma 3 straightforwardly. Lemma 3 descries almost the same specification of Theorem 3.

□

Theorem 3 swaps the outputs and inputs of the auxiliary functions. In the definition of function `f1`, $\texttt{g}_1$ manages the computation of return value (i.e., output), but it manages the updating computation of accumulation parameter (i.e., input) in the definition of function `f2`. In contrast, $\texttt{g}_2$ manages the computation of the accumulation parameter in function `f1` but it manages the computation of return value in the definition of function `f2`. These facts reflect swapping of inputs and outputs. We depict computation process of `f1` and `f2` in Figure 3.

5.2 Manipulating Circular Programs by IO Swapping

Bird [Bir84] discussed circular programs whose return value is the same as (used as) an input. A well known example is the `repmin` function that replaces all element values in a data structure by the minimum element value. He showed the tree version of `repmin` but for simplicity we discuss the list version of `repmin`.

```
repmin x = let (r, m) = repmin' x m in r
    where repmin' (a:x) m = let (r', m') = repmin' x m
                            in (m:r', min a m')
          repmin' [] m = ([], +∞)
```

The underlined variable `m` above expresses circularity, that is, the output value is used as the input value. Function `repmin'` returns the minimum value of the list as the second component of the return

value, while the minimum value is also the second argument of `repmin'` and used for construction of the output list.

The wonder of this function is "the output value is used as the input value". Notice that IO swapping is capable of swapping the output and the input. So think literally about swapping of output and input in the above phrase, "the input value is used as the output value." This new phrase describes behavior of usual functions with no wonder now. To see this, let us apply IO swapping to `repmin`.

```
repmin2 x = let ([], r', m) = repmin2' x ([], +∞) in r'
    where repmin2' (b:y) (r, m') = let (a:x, r', m)= repmin2' y (m:r, min a m')
                                   in (x, r', m)
          repmin2' [] (r, m)= (x, r, m)
```

We would have expected a result without circularity. The circularity existing in `repmin` did disappear as expected, but a new circular is produced inside the definition of `repmin2` (as in the underlined place). The reason is simple; IO swapping swaps inputs and outputs, then any use of input value as output value in `repmin` will introduce circularity in `repmin2`.

Even with circularity, function `repmin2` has a nice property `repmin` does not have. That is, the circularity in `repmin2` is local in the sense it has no circularity on top level of function call as `repmin`. This makes `repmin2` more suitable for manipulation than `repmin`. Consider, for instance, that we want to fuse a function with `repmin`.

```
g (repmin x)
```

The top level circularity of `repmin` prevents `g` from going into `repmin'`, because `repmin` is difficult to be defined in terms of `foldr`. In contrast, fusion law applies to

```
g (repmin2 x)
```

because function `repmin2` has no top level circularity and can be defined in terms of `foldr`.

Although it does not always hold that IO swapping eliminates top level circularity, the following theorem indicates that a wide class of functions having circularity at the top are applicable.

theorem 4 (Removing Top Level Circularity)
The following two functions `f1` and `f2` are equivalent.

```
f1 x h0 = let r = f1' x (g3 r h0)
    where f1' [] h = e
          f1' (a:x') h = let r = f1' x' (g2 a r h)
                         in g1 a r h

f2 x h0 = let ([], h, r') = f2' (x, e) in r'
    where f2' ([], r) = (x, g3 r h0, r)
          f2' (b:y, r) = let (a:x', h, r') = f2' (y, g1 a r h)
                        in (x', g2 a r h, r')
```

proof
It immediately holds from Theorem 3.
□

5.3 IO Swapping on Trees

We shall generalize IO swapping from lists to other data structures such as trees. Recalling the IO swapping on lists, we turn over an input list to swap input and output. But what does it mean by "turning over" a binary tree:

```
data Tree a = Node a (Tree a) (Tree a) | Leaf a
```

and will the result be still a binary trees? Our answer is "Yes, we can turn over a binary tree and get a binary tree". To reach the point, let us think more about what we want. What we should turn over is not a data structure itself but the order of iteration, that is, function calls. In case of lists, the order of iteration is the same as that of the input list and we can reverse the order of iteration by turning over the input list. But, in case of binary trees, the order of iteration is not the same as the input tree. To resolve this problem, we make use of the fact (in compiler) that the order and circumstances of function calls are generally captured by the stack frame, and turn over the stack frame to reverse the order of iteration on data structures.

theorem 5 (IO Swapping on Tree)
The following three functions are equivalent.

```
f1 (Node n l r) = g1 n (f1 l) (f1 r)
f1 (Leaf n) = g0 n

f2 t = let [r] = f2' [t] in r
    where f2 ((Node n l r):x) = let (lv:rv:xv) = f2' (l:r:x)
                                in g1 n lv rv : xv
          f2 ((Leaf n):x) = g0 n : f2' x
          f2 [] = []

f3 t = let ([r],[]) = f3' [t] [] in r
     where f3' (b:y) h = let (rs, a:x) = f3' (pushChildren b y) (f3'' a h)
                         in (rs, pushChildren a x)
           f3' [] h = (h,[t])
           f3'' (Node n l r) (lv:rv:xv) = g1 n lv rv : xv
           f3'' (Leaf n) xv = g0 n : xv
           pushChildren (Node n l r) x = l:r:x
           pushChildren (Leaf n) x = x
```

proof sketch
Simple induction proves the equivalence of function `f1` and `f2`. We can think that function `f2` iterates the list which is the result of flattening of the input tree. Applying Theorem 3 to the function iterating the result of flattening of tree gives function `f3`.
□

Starting from function `f1`, we determine the order of iteration in left-most depth-first fashion and get function `f2`, before we turn over the order of iteration to get function `f3`.

If the input data structures are neither binary trees nor lists, we can similarly construct corresponding variation of IO swapping, by first determining the iteration order, describing the iteration order using stack, and swapping inputs and outputs. This generic IO Swapping sounds interesting, though we have not yet found practical use of it.

6 Related Work

We have already argued in the previous sections the relation between our research and TABA. To observe other aspects of our results, it is worth reviewing the foundation of our research, technique of attribute grammars [Knu68] [Küh98] [Küh99]. Recall the definition of symbolic convolution function `cnv`:

```
cnv x y = zip x (reverse y)
```

This function involves two kinds of technically challenging compositions: fusion of functions with existence of accumulation parameters, and fusion of `zip`-like functions with two recursion parameters. Several fusion methods for accumulative functions have been proposed [CDPR99] [Voi02] [Sve02] [Nis04], and a majority of them employ, as their underlying techniques, attribute grammars. The reason is that the view point in attribute grammars makes it easy to manipulate accumulation parameters. Our results also indicate an approach toward fusing `zip`-like functions, though we have not pursued in this direction here.

We exploited, during the transformation, the symmetry between the input and the output. An important characteristic of attribute grammars is symmetry between synthesized attributes and inherited attributes, which correspond to return values and accumulation (or context) parameters, respectively. This symmetry enables us to treat two classes of values equally, and it plays an important role for IO swapping in this paper as well as function fusion with existence of accumulation parameters. As another advantage, it is also known that circularity in programs can be easily captured in the world of attribute grammars [Joh87].

Our motivation behind the scene is to construct another bridge between the world of functional programing and the one of attribute grammars. Existing researches which tried to apply attribute grammars techniques to functional programming have not fully utilized the benefits of attribute grammars so that they just focused only on one problem domain. Researches about the relationship between functional programming and attribute grammars in a broader sense also exist [DPRJ96] [FJMM91]; they stayed as generic frameworks where it is often hard to manipulate concrete programs and apply program transformation. Our result successfully hoists the essence of attribute grammars and imports it into the functional world as IO swapping.

When we turn our eyes again to TABA, IO swapping is to structurize the production of `reverse`-like functions. Under functional treatments, functions producing output data structures in the return values are considered as better producers than functions producing output data structures in accumulation parameters. IO swapping is therefore of great help as Theorem 2 has shown. The power of IO swapping, however, is limited for structurizing data flows: We cannot structurize functions which are not expressed by `foldl`. Kühnemann et al. proposed more generic method [KGK01] [GKV03], called *deaccumulation*. Generality sometimes brings about the price to pay, and in this case they have to live with an untyped world, introduction of new data type during transformations, and degradation of efficiency. Once functions fit in the form of `foldl` IO swapping guarantees transformation without these troubles.

As the final remark, Danvy and Goldberg [DG02] mention that defunctionalization [DN01] transforms `rev_n` function in Section 3.2, that is one instance of TABA, into usual accumulative `reverse` function. Their transformation is somehow accidental. They haven't figured out what occurred by that transformation. They haven't formalized a transformation method which translate from `reverse` to `rev_n`, either. From a still picture taken by them, we wrote off in this paper the vivid *trilogy* where TABA comes from, what it is and where it will go.

7 Conclusion and Future Work

This paper showed a transformational approach to derive TABA using a new technique called IO swapping. This novel transformation rule swaps the outputs and inputs of functions. We show not only the derivation of TABA, but also the manipulation of TABA and the promising future of TABA by calculational method of which we reinforce the transformation power with IO swapping. Our approach was to confirm the competence of calculational programming for deriving efficient program from naive definition through these transformations of programs.

IO swapping truly makes one region of functional programming world clear. We have some knowledge about manipulation of iteration in the return value, circular data dependency of inputs and outputs, and relationship between lists and other data structures. But here are still some dim region. For example, the perfect solution of fusion for functions traversing plural data structure, having circular data dependency, having irregular iteration or production of data structures. More investiga-

tion on fusion and exploitation of other transformation techniques are required. We hope that more algorithms are derived in a systematic manner.

Acknowledgement

We are very grateful to Shin-Cheng Mu and Keisuke Nakano for their inspiring discussions at the laboratory seminars, and to Olivier Danvy for introducing us the TABA work as well as the fusion problem in it.

References

[BdM96] Richard Bird and Oege de Moor. *Algebras of Programming*. Prentice Hall, 1996.

[Bir84] Richard Bird. Using circular programs to eliminate multiple traversals of data. *Acta Informatica*, 21:239–250, 1984.

[Bir89] Richard Bird. Algebraic identities for program calculation. *Computer Journal*, 32(2):122–126, 1989.

[Bir98] Richard Bird. *Introduction to Functional Programming using Haskell*. Series in Computer Science. Prentice Hall, 1998.

[CDPR99] Loic Correnson, Etienne Duris, Didier Parigot, and Gilles Roussel. Declarative program transformation: A deforestation case-study. In *Proceedings of the International Conference on Principles and Practice of Declarative Programming*, pages 360–377, 1999.

[DG02] Olivier Danvy and Mayer Goldberg. There and back again. In *Proceedings of the 7th ACM SIGPLAN International Conference on Functional programming, ICFP'02*, pages 230–234. ACM Press, 2002.

[DN01] Olivier Danvy and Lasse R. Nielsen. Defunctionalization at work. In *Proceedings of the 3rd ACM SIGPLAN international conference on Principles and practice of declarative programming, PPDP'01*, pages 162–174, New York, NY, USA, 2001. ACM Press.

[DPRJ96] Etienne Duris, Didier Parigot, Gilles Roussel, and Martin Jourdan. Attribute grammars and folds: Generic control operators. Technical Report 2957, INRIA, 1996.

[FJMM91] Maarten M. Fokkinga, Johan Jeuring, Lambert Meertens, and Erik Meijer. A translation from attribute grammars to catamorphisms. *The Squiggolist*, 2(1):20–26, 1991.

[GKV03] Jürgen Giesl, Armin Kühnemann, and Janis Voigtländer. Deaccumulation — Improving provability. In Vijay A. Saraswat, editor, *Proceedings of the Eighth Asian Computing Science Conference, Mumbai, India*, volume 2896 of *LNCS*, pages 146–160. Springer-Verlag, 2003.

[HIT96] Zhenjiang Hu, Hideya Iwasaki, and Masato Takeichi. Deriving structural hylomorphisms from recursive definitions. In *Proceedings of the 1st ACM SIGPLAN International Conference on Functional Programming, ICFP'96, Philadelphia, PA, USA*, volume 31(6), pages 73–82. ACM Press, New York, 1996.

[HITT97] Zhenjiang Hu, Hideya Iwasaki, Masato Takeichi, and Akihiko Takano. Tupling calculation eliminates multiple data traversals. In *Proceedings of the 2nd ACM SIGPLAN International Conference on Functional Programming ICFP'97, Amsterdam, The Netherlands*, pages 164–175. ACM Press, 1997.

[Joh87] Thomas Johnsson. Attribute grammars as a functional programming paradigm. In *Proceedings of the 3rd International Conference on Functional Programming Languages and Computer Architecture, FPCA'87, Portland, Oregon, USA*, pages 154–173, 1987.

[KGK01] Armin Kühnemann, Robert Glück, and Kazuhiko Kakehi. Relating accumulative and non-accumulative functional programs. In *Proceedings of the 12th International Conference on Rewriting Techniques and Applications ,RTA'01*, pages 154–168. Springer-Verlag, 2001.

[Knu68] Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

[Küh98] Armin Kühnemann. Benefits of tree transducers for optimizing functional programs. In *Proceedings of the 18th Conference on Foundations of Software Technology & Theoretical computer Science, FST&TCS'98*, pages 146–157, 1998.

[Küh99] Armin Kühnemann. Comparison of deforestation techniques for functional programs and for tree transducers. In *Proceedings of the 4th Fuji International Symposium on Functional and Logic Programming, FLOPS'99*, pages 114–130, 1999.

[LS95] John Launchbury and Tim Sheard. Warm fusion: Deriving build-catas from recursive definitions. In *Proceedings of the 7th ACM SIGPLAN/SIGARCH International Conference on Functional Programming Languages and Computer Architecture, FPCA'95, La Jolla, San Diego, CA, USA*, pages 314–323. ACM Press, New York, 1995.

[Nis04] Susumu Nishimura. Fusion with stacks and accumulating parameters. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM'04*, pages 101–112. ACM Press, 2004.

[Sve02] Josef Svenningsson. Shortcut fusion for accumulating parameters & zip-like functions. In *Proceedings of the 7th ACM SIGPLAN International Conference on Functional programming, ICFP'02*, pages 124–132. ACM Press, 2002.

[Voi02] Janis Voigtländer. Using circular programs to deforest in accumulating parameters. In Kenichi Asai and Wei-Ngan Chin, editors, *Proceedings of the Asian Symposium on Partial Evaluation and Semantics-Based Program Manipulation, Aizu, Japan*, pages 126–137. ACM Press, 2002. Extended version appeared in *Higher-Order and Symbolic Computation*, volume 17(1–2), pages 129–163, 2004.

Appendix

We show the derivation of `alleq`, `takehalf` and `drophalf`, which we introduce in Section 4.2. These functions are defined as follows.

```
alleq x = and (map (\(a,b)->a==b) x)
takehalf x = take (div (length x) 2) x
drophalf x = drop (div (length x) 2) x
```

Deriving `alleq` is trivial. Theorem 1 immediately gives the efficient definition.

```
map (\(a,b)->a==b) ⇒ foldr (\(a,b) r->(a==b):r) []
alleq x = and (map (\(a,b)->a==b) x)
        ⇒   { foldr form of map }
          and (foldr (\(a,b) r->(a==b):r) [] x)
        ⇒   { Theorem 1 with checking the following conditions
                and [] ⇒ True
                and ((\(a,b) r->(a==b):r) (a,b) r) ⇒ (a==b) && and r }
          foldr (\(a,b) r->(a==b) && r) True x
```

Derivations of `takehalf` and `drophalf` are somewhat technical. We only show about `drophalf` because both are almost the same.

First we define `drop`, `length` and `div` by using constructor `S` corresponding successor and `Z` corresponding zero as follows.

```
--data N = S N | Z
drop (S v) (a:x) = drop v x
drop Z x = x
length (a:x) = S (length x)
length [] = Z
--div v 2 = div2 v
div2 (S(S v)) = S (div2 v)
div2 (S Z) = Z
div2 Z = Z
```

Here we use a special sequence of symbols `--` as one line comment.

Fusing `div2 (length x)` is easy. We can get the result immediately.

```
--halflen x = div2 (length x)
halflen (a:b:x) = S (halflen x)
halflen (a:[]) = Z
halflen [] = Z
```

Recognize each two elements of the list as a pair form head of it and the list itself as list of pairwise elements . Then function `halflen` is just a instance of `foldr` on list of pair. We program this tow-elements-iterating `foldr`, named `foldr'`, as follows.

```
foldr' f e [] = e
foldr' f e (a:b:x) = f (a,b) (foldr' f e x)
```

Where `foldr'` is a partial function, on the even-length list. And `halflen` is programed by using `foldr'`.

```
halflen = foldr' (\a r->S r) Z
```

Function `foldr'` is not `foldr`. But don't worry. Function `foldr'` is equipped with the same fusion law as `foldr`[HIT96]. So we can proceed to the next step of derivation.

Next, we fuse `drop (halflen x) x`.

```
drophalf x = drop (halflen x) x
           ⇒   { foldr' form of halflen }
             drop (foldr' (\a r->S r) Z x) x
           ⇒   { Theorem 1 with checking the following conditions
                 drop Z x ⇒ x
                 drop ((\a r->S r) a r) x ⇒ drop r (tail x) }
             foldr' (\a r x->r (tail x)) id x x
```

This is what we want, the efficient `foldr'` form of `drophalf`.