IO Swapping Leads You There And Back Again
(Extended Abstract) ⋆

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

Department of Mathematical Informatics, University of Tokyo
{Akimasa_Morihata,kaz,hu,takeichi}@mist.i.u-tokyo.ac.jp

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

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 solution, 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 (b:y',r) = walk x'
                        in (y',(a,b):r)
```

This program uses a bit unusual auxiliary function `walk`. When the input `x` is empty, `walk` uses the input `y` directly as a return value, and its 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 the resulting output.

TABA is truly tricky. It would be interesting to see whether there is a systematic way that may *lead us* to construct such TABA programs. One may wish to use and manipulate TABA-like computations for constructing a new kind of such iterations, i.e., iterations over some return values. In [DG02] Danvy and Goldberg gave a set of clever TABA programs, but neither derivation nor manipulation of them were presented sufficiently. In a recent paper [DG05], they showed how to derive TABA programs based on the two known transformations:

⋆ This is an extended abstract of the technical report [MKHT05]: A. Morihata, K. Kakehi, Z. Hu, and M. Takeichi. Reversing iterations: IO swapping leads you there and back again. *Technical Report METR* 2005-11, Department of Mathematical Informatics, University of Tokyo, May 2005. Available from `http://www.keisu.t.u-tokyo.ac.jp/Research/METR/2005/METR05-11.pdf`

```
fst (a,b) = a
snd (a,b) = b
head [x₀,x₁,...,xₙ] = x₀
tail [x₀,x₁,...,xₙ] = [x₁,x₂,...,xₙ]
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
```

Fig. 1. Informal definitions of standard functions

CPS transformation and defunctionalization [DN01]. Though being systematic, the method is not constructive; that is to say, it is not incremental.

In this paper, we show a new and clear derivation of TABA programs by a novel program transformation rule called *IO swapping*. It swaps input and output values of a function, introduces iteration at return times and immediately derives TABA programs. We also demonstrate manipulations of TABA. We can incrementally construct bigger TABA programs from simpler TABA programs by program calculation [BdM96], a transformational approach to carrying programs from their naive definition to their efficient equivalent. The ability to manipulate TABA programs proves the effectiveness of program calculation.

Throughout the paper we use the notation of the functional programming language Haskell [Bir98]. The symbol \ is used instead of λ for λ-expressions. The symbol `.` denotes function composition. We use many standard Haskell functions, whose informal definitions are given in Figure 1. We also assume that the size of structured data we are treating is finite.

2 Calculational Programming and IO Swapping

2.1 Calculational Programming

Functional programming languages provide a constructive way of programming, namely development of involved 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 intensively use the following fusion (promotion) law [Bir89].

Theorem 1 (Fold Fusion).

`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 programs can be manipulated using calculational programming.

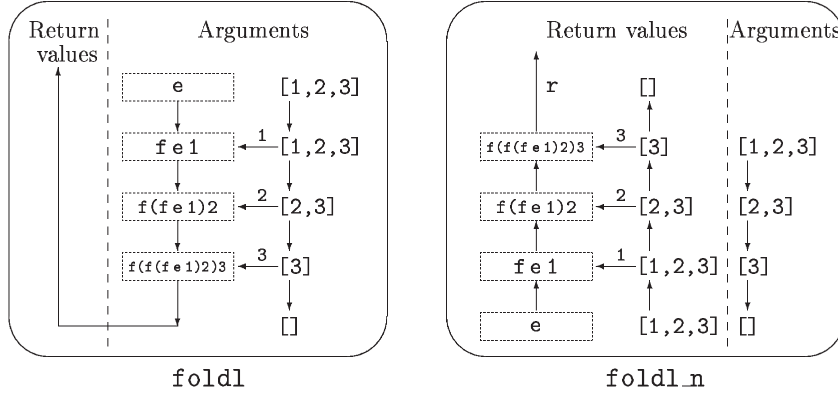

Fig. 2. The models of computation processes of `foldl` and `foldl_n`

2.2 IO Swapping for `foldl`

The new and effective transformation rule proposed in this paper is *IO swapping*, which changes the view of functions: It "thinks upside down" about treatments of data structures through literally swapping the input and the output. Before going into the general framework, we explain the essence of the proposed transformation using a typical function `foldl` in the following theorem.

Theorem 2 (IO Swapping for `foldl`).
The following two functions `foldl` and `foldl_n` are equivalent.

```
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)
```
□

In `foldl'` the initial input list `x` of `foldl_n` is passed directly as the return value of the termination condition, and destructed in its recursive call. In short, this theorem achieves the transformation from `foldl` to its TABA form.

We make some remarks on the implications of this theorem. 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 return value of `foldl_n`. This indicates the fact that IO swapping is a rule to swap the inputs (arguments) and the outputs (return values) of the original function. 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 a 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 a value. When the King, living at the top of this tower, commands servants to gather the values, some go downward from the top to the ground floor (like arguments) or others go upward from the ground floor to the top (like return values), as the phrase "`cdr` down, `cons` up" indicates. If the values in the tower are secretly rearranged upside-down, what these servants gather up are exchanged. Instead of such rearrangement, the equivalent effect can take place by transferring consumption of the list from the position of the argument to that of 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.

2.3 IO Swapping

It is possible to reinforce Theorem 2 so that it can deal with many more functions.

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')
```
□

Theorem 3 swaps the inputs and outputs of the auxiliary functions. In the definition of function `f1`, g_1 manages the computation of return value, but it manages that of accumulation parameter in `f2`. In contrast, g_2 manages the computation of the accumulation parameter in `f1` but it manages that of return value in `f2`. Applications of Theorem 3 are in [MKHT05].

3 Deriving TABA Programs by IO Swapping and Fusion

Now we show our derivation of TABA programs. We propose two methods: IO swapping and program calculation. The former directly derives a TABA program and the latter incrementally derives a bigger TABA program by promoting functions into a smaller TABA program.

3.1 List Reversal

We start by demonstrating a derivation of TABA-style `reverse` by the first method. Function `reverse` is defined by using `foldl` as follows.

```
reverse = foldl (\y a -> a:y) []
```

Applying Theorem 2 to `reverse`, we instantly get the following function `rev_n`, which is the TABA program for `reverse`.

```
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')
```

3.2 Symbolic Convolution

Next we show a systematic derivation of the TABA program for `cnv` in the introduction, starting from the following straightforward specification:

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

where we assume that `x` and `y` have the same length.

To derive TABA-style `cnv` we use the second method, program calculation, for we already get the TABA program for `reverse` namely `rev_n` in Section 3.1. We derive the TABA program for `cnv` by promoting `zip` into `rev_n`.

The function `rev_n` 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 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, η-expansion to remove function values yields the following program, where the case `cnv' [] []` is obtained from the assumption that length of `x` and `y` are same.

```
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)
```

Next we eliminate the unnecessary parameter: The first argument of `cnv'` is not used at all for producing results. It derives the efficient TABA program for `cnv`.

```
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)
```

3.3 List Reversal Revisited

As we have seen in successful derivation of `cnv`, program calculation works effectively if we have some TABA programs in hand. Making sure of it, we choose `rev_n`, which is obtained by IO swapping in Section 3.1, as a next example.

From the definition of `cnv`, we extract `rev_n` as follows.

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

Starting from this equation, we obtain `rev_n` by fusing `map` with `cnv`.

```
rev_n 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

```
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)) a (id_map snd (b:y',r))
```

we get

```
rev_n 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 program calculation can be useful guidance for developing TABA programs. We can also derive more involved programs, such as the efficient palindrome detecting program. See [MKHT05].

4 Conclusion and Future Work

This paper presented a new approach to derive TABA programs systematically using a novel technique called IO swapping, which swaps the outputs and inputs of functions. Our approach confirms the competence of calculational programming for deriving efficient program from naive definition through these transformations of programs.

Our belief is that the effect of IO swapping is not limited to derivation of TABA. Investigation of some further applications still remains.

Acknowledgement

We are very grateful to Olivier Danvy for introducing us the TABA work and its relation to defunctionalization and CPS transformation, and to Shin-Cheng Mu and Keisuke Nakano for their inspiring discussions at the laboratory seminars.

References

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

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

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

[DG02] O. Danvy and M. Goldberg. There and back again. In *Proc. of the 7th Int. Conf. on Functional programming,* pages 230–234, 2002.

[DG05] O. Danvy and M. Goldberg. There and back again. Technical report, *BRICS Research Series RS-02-12.* Extended version of an article to appear in Fundamenta Informaticae, 2005.

[DN01] O. Danvy and L. R. Nielsen. Defunctionalization at work. In *Proc. of the 3rd Int. Conf. on Principles and practice of declarative programming*, pages 162–174, 2001.

[MKHT05] A. Morihata, K. Kakehi, Z. Hu, and M. Takeichi. Reversing iterations: IO swapping leads you there and back again. *Technical Report METR* 2005-11, Department of Mathematical Informatics, University of Tokyo, 2005.