Swapping Arguments and Results of Recursive Functions

Akimasa Morihata[†], Kazuhiko Kakehi[‡], Zhenjiang Hu[†], and Masato Takeichi[†]

[†] Graduate School of Information Science and Technology, University of Tokyo
[‡] Division of University Corporate Relations, University of Tokyo
7-3-1 Hongo, Bunkyo-ku, 113-8656 Tokyo, JAPAN
{morihata,kaz}@ipl.t.u-tokyo.ac.jp,
{hu,takeichi}@mist.i.u-tokyo.ac.jp

Abstract. Many useful calculation rules, such as fusion and tupling, rely on well-structured functions, especially in terms of inputs and outputs. For instance, fusion requires that well-produced outputs should be connected to well-consumed inputs, so that unnecessary intermediate data structures can be eliminated. These calculation rules generally fail to work unless functions are well-structured. In this paper, we propose a new calculation rule called *IO swapping.* IO swapping exchanges call-time computations (occurring in the arguments) and return-time computations (occurring in the results) of a function, while guaranteeing that the original and resulting function compute the same value. IO swapping enables us to rearrange inputs and outputs so that the existing calculation rules can be applied. We present new systematic derivations of efficient programs for detecting palindromes, and a method of higher-order removal that can be applied to defunctionalize function arguments, as two concrete applications.

1 Introduction

Calculational programming [1] is a methodology for constructing programs, where we first write down a program that may be terribly inefficient but certainly correct, then we improve its efficiency by applying calculation rules, such as fusion [2, 3] and tupling [4–7]. As an example, consider the problem of checking whether a list is a palindrome or not. A straightforward solution `pld0` is given as follows.

```
pld0 x = eqlist(x,reverse x)

eqlist([],[]) = True
eqlist(a:x,b:y) = a==b && eqlist(x,y)

reverse x = rev x []
  where rev [] h = h
        rev (a:x) h = rev x (a:h)
```

The function `reverse` reverses the order of a list, and the function `eqlist` checks whether two lists of the same length are equal. This program is accurate but inefficient on account of multiple traversals over the input list `x`; both `eqlist` and `reverse` iterate their computation over `x`. Tupling enables us to eliminate such multiple traversals [4–7]. For example, Bird [4] derived

```
pldBird x = let (r1,r2) = aux x r2 [] in r1
  where aux [] [] h = (True, h)
        aux (a:x) (b:y) h = let (r1,r2) = aux x y (a:h)
                            in (a==b && r1, r2)
```

Alternatively, Pettorossi and Proietti [6] derived

```
pldPettorossi x = let (r1,r2) = aux x [] in r1 r2
  where aux [] h = (\y->y==[], h)
        aux (a:x) h = let (r1,r2) = aux x (a:h)
                      in (\(b:y)->a==b && r1 y, r2)
```

Both involve a single traversal of `x`, and tupling plays an important role.

As can be seen in this palindrome detecting problem, calculation rules are useful for developing various kinds of programs if functions are well-structured. For instance, tupling calculation eliminates multiple traversals if two functions have the same structure for the recursion; in facts we succeeded in eliminating multiple traversals in the palindrome detecting problem, because `reverse` and `eqlist` certainly have the same recursion structure. However, These calculation rules generally fail to work unless functions are well-structured.

Let us turn to another improvement to solutions for the palindrome detecting problem. The previous two solutions, namely `pldBird` and `pldPettorossi`, construct intermediate lists in the accumulative arguments (denoted by `h`). The intermediate lists originate from the function `reverse`, and they are another source of inefficiency. In other words, the intermediate list produced by `reverse` is consumed by `eqlist` as follows.

```
pld1 = eqlist · (id △ reverse)
```

A question that naturally arises is: “Can we derive an efficient palindrome detecting program without an intermediate list?”. One obvious idea is to fuse `eqlist` with `(id △ reverse)`, however, applying the fusion rule is not easy, because there are unsuitable connections between `eqlist` and `(id △ reverse)`. Fusion requires that well-produced outputs should be connected to well-consumed inputs so that the intermediate data structure can be eliminated. However, `eqlist` consumes two lists simultaneously while `(id △ reverse)` produces two lists differently: `id` produces a list in its *results* while `reverse` produces a list in its accumulative *arguments*.

In this paper, we introduce a novel program transformation called *IO swapping*. IO swapping exchanges call-time computations (occurring in the arguments) and return-time computations (occurring in the results) of a function, while guaranteeing that the original and resulting function compute the same value. IO swapping enables us to rearrange inputs and outputs so that existing calculation rules can be applied. For example, we can derive the following program, `rev_n`, from `reverse` defined above using IO swapping.

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

In contrast to `reverse`, function `rev_n` constructs the reversed list at return-time. The production structure of `rev_n` is now the same as `id`, and fusion with `eqlist` successfully derives the program

```
pld1 x = snd (aux x)
  where aux [] = (x,True)
        aux (b:y) = let (a:z,r') = aux y in (z,a==b&&r')
```

This function `pld1` is slightly more efficient than `pldBird` and `pldPettorossi`. Although all three functions `pld1`, `pldBird` and `pldPettorossi` require two traversals, the derived function `pld1` does not require an intermediate list.

The remainder of this paper is organized as follows. We introduce IO swapping in Section 3. We then give two concrete applications of IO swapping in the two sections that follow. The first, in Section 4, is new systematic derivations of efficient programs for detecting palindromes. The second, in Section 5, is a derivation of the transformation of higher-order removal that can be applied to defunctionalize function arguments. Finally, we discuss related work in Section 6 and conclude the paper in Section 7.

2 Preliminaries

2.1 Notations

Throughout the paper, we have mostly used the notation in functional programming language Haskell [8]. Some syntactic notations we have used in this paper are as follows. The backslash \ is used instead of λ for λ-abstraction, and the identity function is written as `(\x -> x)`. The symbol $\cdot$ denotes function composition, i.e., `(f·g) x = f(g x)`. The underscore _ stands for the "don't care" value. We have used the special symbols $\times$ and $\vartriangle$ to express tupled functions for notational convenience: `(f × g) (x,y) = (f x, g y)` and `(f △ g) x = (f x, g x)`. Many basic Haskell functions have been used in this paper; their informal definitions are given in Fig. 1. We have assumed that evaluation is based on *lazy evaluation*, data structures are finite, and all patterns are irrefutable except for those of recursion parameters.

2.2 Fusion and Tupling

Functional programming languages provide a compositional way of programming; larger programs are developed through the composition of smaller and

```
id x = x
fst (a,_) = a
snd (_,b) = b
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]
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
div n m = ⌊n/m⌋
```

Fig. 1. Informal definitions of basic functions

simpler functions. Fusion and tupling play an important role in improving the efficiency of compositional programs. Fusion combines the composition of two functions into one and eliminates the intermediate data structure between them. Tupling eliminates multiple traversals of the same data if two functions share the same recursion scheme. We will later make use of the following fusion rule [2] and tupling rule [7].

Theorem 1 (Fold Promotion).

$$\texttt{f} \cdot \texttt{foldr} \ (\oplus) \ \texttt{e} = \texttt{foldr} \ (\otimes) \ \texttt{e'}$$

provided that $\otimes$ and `e'` are such that `f (a` $\oplus$ `y)` $=$ `a` $\otimes$ `(f y)` and `f e` $=$ `e'` hold for any `a` and `y`. □

Theorem 2 (Simple Tupling).

```
(f1 △ f2) = foldr (\a (r1,r2)->(k1 a r1, k2 a r2)) (z1,z2)
```

where `f1 = foldr k1 z1` and `f2 = foldr k2 z2`. □

Both Theorems 1 and 2 can be generalized to be polytypic [3, 9, 7].

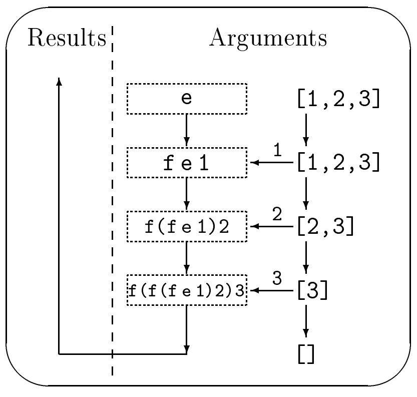
3 IO Swapping

3.1 IO Swapping for `foldl`

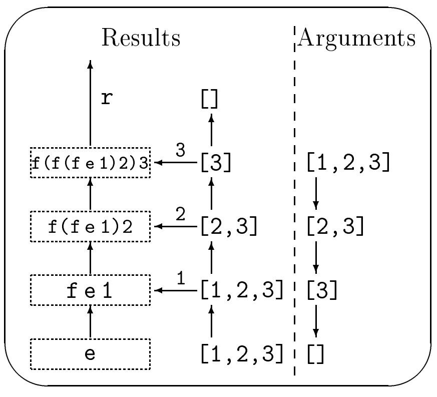
IO swapping is the new transformation that is used to change the view of recursive functions through the swapping of input (arguments) and output (results). The following theorem shows the IO swapping rule for a typical function, `foldl`. Before going into the general framework, let us illustrate the basic idea behind IO swapping using this theorem.

Theorem 3 (IO Swapping for `foldl`). The functions `foldl` and `foldl_n` defined below are equivalent.

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



foldl



foldl_n

Fig. 2. Models of the computation processes of `foldl` and `foldl_n`

```
foldl_n f e x = let ([],r) = foldl' x in r
  where
   {- foldl' y = (drop (length y) x, foldl f e (take (length y) x)) -}
    foldl' [] = (x,e)
    foldl' (_:y) = let (a:z,r) = foldl' y
                   in (z,f r a)
```

Proof. This is the direct consequence of Theorem 4, which we are going to introduce in Section 3.2. Applying Theorem 4 to `foldl` and removing unnecessary variables yields `foldl_n`. □

Notice how the result is computed using function parameter `f` of `foldl` and `foldl_n`. While `f` is applied to the accumulation parameter in function `foldl`, it does the computation of the result in `foldl'`. This is because IO swapping is a rule to swap the call-time computation (occurring in the arguments) and the return-time computation (occurring in the results) of the original function.

Fig. 2 illustrates the recursion stacks with the value flows for `foldl` and `foldl_n` (`foldl'`). If we ignore the argument of `foldl'`, we can easily see that `foldl` and `foldl_n` compute exactly the same value, except that the computation is done at different times, call time or return time; moreover, inverting the figure for `foldl` makes it look almost the same figure as that for `foldl_n`. IO swapping swaps call-time and return-time computation without changing the whole process of computation by 'turning the recursion stack upside down', because call-time and return-time computation correspond to top-to-bottom and bottom-to-top computation in the figure, respectively.

Note also that to do swapping we need to estimate the recursion depth from which we should start the computation, because `foldl'` should finish its whole computation exactly at the top of the recursion. We can use the input list to estimate the recursion depth and indeed `foldl'` does this, because there is no difference in the recursion depth between `foldl'` and `foldl`.

3.2 IO Swapping for List Catamorphisms

The idea behind Theorem 3 can be generalized so that it can be applied to higher-order list catamorphisms [3], known to be a generalized form of `foldr` and `foldl`. The following theorem describes the IO swapping rule for higher-order list catamorphisms with circularity [4].

Theorem 4 (IO Swapping for List Catamorphisms). For any suitably-typed `g0`, `g1`, `g2`, and `g3`, the following two functions, `f1` and `f2`, are equivalent.

```
  {- g0::r->h, g1::h->r, g2::a->r->h->h, g3::a->r->h->r -}

f1 :: [a] -> r
f1 x = let r = f1' x (g0 r) in r
  where {- f1' :: [a] -> h -> r -}
        f1' [] h = g1 h
        f1' (a:z) h = let r = f1' z (g2 a r h)
                      in  g3 a r h

f2 :: [a] -> r
f2 x = let ([],h,r') = f2' x (g1 h) in r'
  where {- f2' :: [a] -> r -> ([a],h,r) -}
        f2' [] r = (x, g0 r, r)
        f2' (_:y) r = let (a:z,h,r') = f2' y (g3 a r h)
                      in (z, g2 a r h, r')
```

Proof Sketch
Here we will provide a proof sketch. The full proof can be found in [10].

To prove Theorem 4, we need to assume that all computations terminate with a unique solution. We call the outside (top) of the recursion of the auxiliary function (`f1'` or `f2'`) the 0-th recursive call and the first call of the auxiliary functions the 1-st recursive call.

Now we can inductively prove that, for all k such that $0 \leq k \leq n$, the first argument, the second argument, and the return value of the k-th recursive call of `f1'` will be the first element of the return value, the second element of the return value, and the second argument of the $(n-k)$-th recursive call of `f2'`, respectively, without any conflict between recursions. Consequently the values of `f1` and `f1'` determine one solution for `f2` and `f2'`, and, from the assumption, it is the only solution for `f2` and `f2'`. Then the result for the whole computation of `f1` is the same as the second argument of `f2'` at the bottom of the recursion. The second argument of `f2'` at the bottom of the recursion is propagated to the top of the recursion without any updating and eventually becomes the result for the whole recursion of `f2`. Therefore the results for `f1` and `f2` are the same. □

As the same as Theorem 3, Theorem 4 swaps the call-time computation and the return-time computation of the auxiliary function. In the definition for `f1`, `g3` does the return-time computation, but in the definition for `f2` it does the call-time computation. In contrast, `g2` manages the call-time computation in

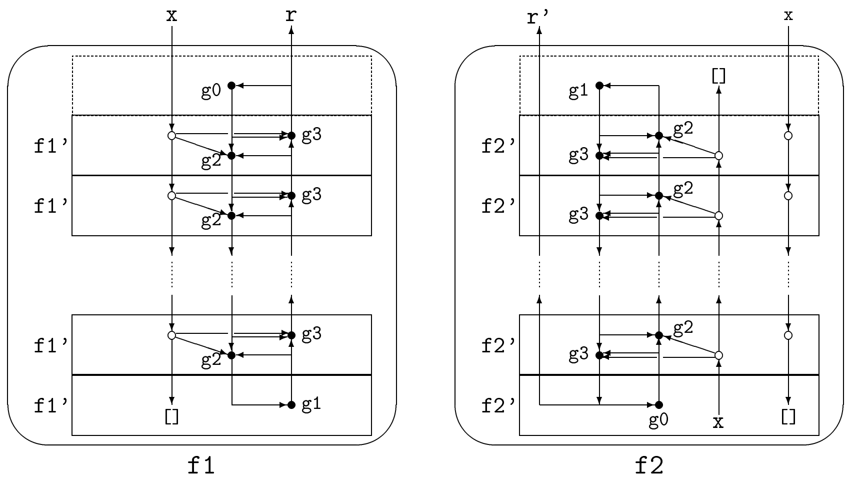


Fig. 3. Models of the computation processes of `f1` and `f2` in Theorem 4

the function `f1`, but under `f2` it does the return-time computation. The auxiliary function of `f2`, namely `f2'`, only uses its first argument for estimating the recursion depth, then does the same computation of `f1'` in the IO-swapped manner. It finally returns the result for whole computation from the bottom of the recursion by the third element of the result of `f2'`. Fig. 3 outlines the computation process for `f1` and `f2`. We can easily see that inverting the figure to compute `f1` yields almost the same figure as `f2`, which reflects the fact that swapping the input and output of `f1` yields `f2`. We can generalize Theorem 4 further, so that it can deal with almost every linear recursive function [10].

Note that we have assumed that data structures are finite. We succeed in estimating the recursion depth because of the finiteness of the input list. In other words, `f2` never returns if the input list is infinite, because estimating the recursion depth needs an infinite recursion. Also note that this does not matter for Theorem 3, because `foldl` never returns anyway for an infinite input.

Higher-order list catamorphisms are known as `foldr` functions with higher-order results, and functions `f1` and `f2` are certainly instances of `foldr` with higher-order results as follows.

```
f1 x = let r = foldr k1 g1 x (g0 r) in r
  where  k1 a p = \h -> let r = p (g2 a r h) in g3 a r h
f2 x = let ([],h,r') = foldr k2 z2 x (g1 h) in r'
  where z2 = \r -> (x, g0 r, r)
        k2 _ p = \r -> let (a:z,h,r') = p (g3 a r h)
                           in (z, g2 a r h, r')
```

So Theorem 4 indicates that applying IO swapping to higher-order list catamorphisms results in higher-order list catamorphisms with a projection function. Moreover, applying IO swapping twice produces the original function after constant propagation is removed. It is well known that catamorphisms are suitable for manipulation, and many transformation rules for them have been developed [3, 1, 7]. Theorem 4 therefore allows us to combine IO swapping with other program manipulation techniques.

The list reversing functions provide an example. From Theorem 4, the following function, `reverse2`, is equivalent to `reverse` defined in Section 1.

```
reverse2 x = let ([],h,r') = rev2 x h in r'
  where rev2 [] r = (x, [], r)
        rev2 (_:y) r = let (a:z,h,r') = rev2' y r
                       in (z, a:h, r')
```

Function `reverse2` produces a resulting list in the result of `rev2`, in contrast to `reverse`, which produces a resulting list in the accumulative argument of its auxiliary function `rev`.

It is worth noting that variable `h` at the top of the recursion of `reverse2` describes a circularity [4], i.e., computational dependency from a result to an argument. This circularity is the IO-swapped appearance of computational dependency from an argument to a result; the auxiliary function of `reverse`, namely `rev`, passes its accumulative argument to its result at the bottom of the recursion, and this corresponds to the circularity that passes a result to an argument at the top of the recursion. In general, IO swapping introduces circularities whenever the original function uses its arguments to compute its results. In other words, `f1`, `f1'`, `f2`, and `f2'` are defined using circularities to capture accumulations. In the case of `foldl`, we do not need circularities as can be seen in Theorem 3, because the dependency from arguments to results is unnecessary in `foldl`. In fact, we can remove the circularity in `reverse2` by removing the second argument and the third element of the result of `rev2`, because they just propagate constants. Removing the circularity results in the function `rev_n` that we discussed in the introduction.

4 Detecting Palindromes

To find out how useful IO swapping is in program development, let us demonstrate the derivation of two efficient palindrome detecting programs that have no intermediate lists. The role of IO swapping is to rearrange the structure of functions in order to enable convenient manipulation. We will first derive a simple palindrome detecting program, `pld1`, to show how IO swapping works, and after that we will derive a more involved but efficient one, `pld2`, that only recurses through half the length of the input list.

4.1 Detecting Palindromes without Intermediate Data

Let us start from the following specification for a palindrome detecting function.

```
pld1 = eqlist · (id △ reverse)
```

The definition for `pld1` has an intermediate list produced by `(id △ reverse)`, but Theorem 1 is not sufficient to eliminate it. As explained in the introduction, the production/consumption structure of the intermediate list does not form a suitable connection for the fusion. More concretely, `eqlist` consumes two lists simultaneously while `(id △ reverse)` produces two lists differently: `id` produces a list in its *results* while `reverse` produces a list in its accumulative *arguments*. Let us show how IO swapping can solve this problem.

First of all, we apply IO swapping to `reverse`. Function `reverse` is an instance of `foldl` as follows:

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

Theorem 3 yields the following program.

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

Note how `rev_n` produces the resulting list at return-time, in the same manner as `id`. Successful fusion of `eqlist` with `(id △ rev_n)` is consequently expected because of suitable connection of the production/consumption structure; `(id △ rev_n)` produces its resulting lists simultaneously, and `eqlist` consumes its input lists simultaneously. Let us confirm it through the following calculation.

We write `id` and `rev_n` in terms of `foldr` as follows, because the `foldr` form is appropriate for the later calculation.

```
id x = foldr (:) [] x
rev_n x = snd (foldr (\_ (a:z,r)->(z,a:r)) (x,[]) x)
```

Tupling (Theorem 2) of `id` and `rev_n` yields the following program.

```
(id △ rev_n) x = snd (foldr (\b (a:z,(r1,r2))->(z,(b:r1,a:r2)))
                            (x,([],[])) x)
```

We now calculate an efficient palindrome detecting function as follows.

```
pld1 x
  = eqlist ((id △ rev_n) x)
  =  {- foldr form of (id △ rev_n) -}
    eqlist (snd (foldr (\b (a:z,(r1,r2))->(z,(b:r1,a:r2)))
                       (x,([],[])) x))
  =  {- Swapping snd with eqlist -}
    snd ((id×eqlist)(foldr (\b (a:z,(r1,r2))->(z,(b:r1,a:r2)))
                            (x,([],[])) x))
  =  {- Fusion (Theorem 1):                                      -}
     {- (id×eqlist)(x,([],[])) = (x,True)                        -}
     {- (id×eqlist)((\b (a:z,(r1,r2))->(z,(b:r1,a:r2))) b r)  -}
     {-    = (z,b==a && eqlist (r1,r2))                          -}
     {-    = (\b (a:z,r')->(z,b==a&&r')) b ((id×eqlist) r)     -}
    snd (foldr (\b (a:z,r')->(z,b==a&&r'))(x,True) x)
```

The resulting function is the one following after `foldr` is unfolded.

```
pld1 x = snd (aux x)
  where aux [] = (x,True)
        aux (b:y) = let (a:z,r') = aux y in (z,a==b&&r')
```

This function, `pld1`, has no intermediate list. IO swapping creates matching connections between the production/consumption structures, and enables successful fusion.

4.2 Detecting Palindromes without Intermediate Data and Using Half the Recursion Depth

To check whether a list is a palindrome or not, we do not need to traverse the whole list; half of it is sufficient. This insight yields a more efficient specification as follows.

```
pld2 = eqlist · (takehalf △ revdrophf)
  where takehalf x = take (div (length x) 2) x
        revdrophf x = reverse (drophalf x)
        drophalf x = drop (div (length x) 2) x
```

For simplicity, we have assumed that the length of the input list is even.

First, let us derive efficient definitions for `takehalf` and `drophalf` using fusion. We omit the details.

```
takehalf x =  foldr' (\_ r (b:y)->b:r y) (\_->[]) x x
drophalf x =  foldr' (\_ r (_:y)->r y) id x x
```

Function `foldr'` is defined below, having the similar fusion and tupling rules to `foldr` [3, 9, 7].

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

Note that `takehalf` produces its resulting list in its return-time computation. This indicates that the combination of `takehalf` and `reverse` is not suitable to be fused with `eqlist`. Here, IO swapping has an effect. We adopted the IO-swapped variant `rev_n` instead of `reverse`, because its production scheme is the same as that for `takehalf`.

```
pld2 = eqlist · (takehalf △ revdrophf2)
  where revdrophf2 x = rev_n (drophalf x)
```

We will next calculate an efficient definition for `revdrophf2`. Here tupling is appropriate, because `drophalf` does not produce a new list and fusion is not suitable in such a situation. Note that `rev_n` and `drophalf` have the same recursion scheme; `rev_n` and `drophalf` have the same recursion depth, and `rev_n` does not use its recursion parameter except for estimating the depth of the recursion. Tupling now yields the following program.

```
revdrophf2 x
  = let ([],r1,dphf)
             = foldr' (\_ r (_:y)->let (a:z,r1,r2) = r y
                                     in (z,a:r1,r2))
                      (\y->(dphf,[],y)) x x
     in r1
```

This definition has an uncomfortable dependency denoted by the variable `dphf`; `dphf` is computed by the recursion of `foldr'` and is used at the bottom of the recursion. We can eliminate this uncomfortable dependency because the third element of the result (denoted by `r2`) remains unchanged throughout the recursion. We can thus obtain the following program.

```
revdrophf2 x
  = let ([],r1) = foldr' (\_ r (_:y)->let (a:z,r1) = r y
                                        in (z,a:r1))
                           (\y->(y,[])) x x
     in r1
```

Finally, we fuse `eqlist` with `(takehalf △ revdrophf2)`. It is almost the same as that discussed in the previous section. Note that `takehalf` and `revdrophf2` have the same recursion scheme and the same production scheme, and tupling `takehalf` with `revdrophalf2` and fusing it with `eqlist` is not difficult. We omit the details.

Tupling yields the following definition for `(takehalf △ revdrophf2)`.

```
(takehalf △ revdrophf2)
  =  {- Tupling (Theorem 2) -}
     let ([],r) = foldr' (\_ r (b:y)->let (a:z,(r2,r1)) = r y
                                       in (z,(b:r2,a:r1)))
                          (\y->(y,([],[]))) x x in r
```

Fusion gives the following definition for `pld2`.

```
pld2 x
  = eqlist ((takehalf △ revdrophf) x)
  = let ([],r) = foldr' (\_ r (b:y)->let (a:z,(r2,r1)) = r y
                                       in (z,(b:r2,a:r1)))
                          (\y->(y,([],[]))) x  x
     in eqlist r

  =  {- Fusion (Theorem 1) -}
     let ([],r) = foldr' (\_ r (b:y)->let (a:z,r') = r y
                                       in (z,b==a&&r'))
                          (\y->(y,True)) x  x
     in r
```

The resulting function is as follows, after `foldr'` is unfolded.

```
pld2 x = let ([],r) = aux x x in r
  where aux [] y = (y,True)
        aux (_:_:x) (b:y) = let (a:z,r') = aux x y
                            in (z, b==a&&r')
```

This program has no intermediate list and its recursion depth is half the length of the input list.

5 Reinforce the Power of Transformations by IO Swapping

In Section 4, we presented an application of IO swapping as a program transformation. This section demonstrates an application of IO swapping as a *metatransformation*; IO swapping can take a program transformation and return one that is an IO-swapped transformation of the old one. We will present a derivation of a higher-order removal transformation that can be applied to defunctionalize function arguments.

5.1 IO Swapping as a Metatransformation

Consider the higher-order removal problem [11]. It is well known that η-expansion effectively defunctionalizes higher-order results. For example, think about the following function, `sumTC`, whose auxiliary function `sum'` returns a function value.

```
sumTC x = let r = sum' x in r 0
  where sum' [] = id
        sum' (a:x) = (sum' x) · (a+)
```

η-expansion yields the usual first-order definition for `sumTC` as follows.

```
sumTC x = sum' x 0
  where sum' [] h = h
        sum' (a:x) h = sum' x (a+h)
```

Despite such effective defunctionalization of higher-order *results*, η-expansion cannot remove higher-order accumulative *arguments*. That is, it cannot work for the following `sumCPS` function, whose auxiliary function constructs a higher-order accumulative argument.

```
sumCPS x = sum' x id
  where sum' [] k = k 0
        sum' (a:x) k = sum' x (\v->k(a+v))
```

We have to find another rule to remove higher-order accumulations. It is inefficient to start from scratch. In Section 5.2, we will derive a new method from η-expansion with IO swapping. Here, let us explain the general idea.

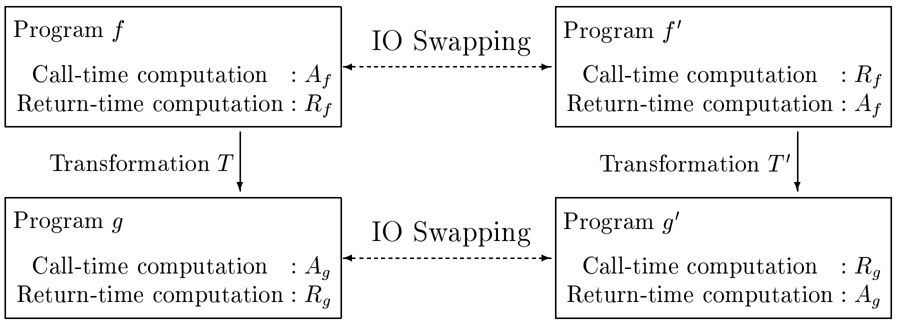

Program f
Call-time computation : A_f
Return-time computation : R_f
IO Swapping
Program f'
Call-time computation : R_f
Return-time computation : A_f
Transformation T
Transformation T'
Program g
Call-time computation : A_g
Return-time computation : R_g
IO Swapping
Program g'
Call-time computation : R_g
Return-time computation : A_g

Fig. 4. Framework provided by IO swapping

The problem is the mismatch between the purpose and the rule; we want to manipulate *arguments*, but η-expansion only defunctionalizes *results*. IO swapping enables us to rearrange the arguments and results to suit manipulation, as seen in Section 4. Applying IO swapping to `sumCPS` above yields the following function.

```
sumCPS' x = let ([], k) = sum_n' x in k 0
  where sum_n' [] = (x, id)
        sum_n' (_:y) = let (a:z,k) = sum_n' y
                       in  (z,\v->k(a+v))
```

The auxiliary function `sum_n'` of `sumCPS'` constructs a higher-order *result*, and seems suitable for the application of η-expansion. The metatransformation use of IO swapping is derived by generalizing this process, and the idea is summarized in Fig. 4. Assume that there are programs f, g and a program transformation T such that $T[\![f]\!] = g$. IO swapping gives functions equivalent to f and g, namely f' and g'. We can now define a program transformation T' by the relation between f' and g', and T' works as an IO-swapped transformation of T. Note that T' is specified by the sequence of program transformations, that is, applying IO swapping after applying T, after applying IO swapping. Consequently, IO swapping provides the relationship between the manipulation of arguments and that of results for recursive functions.

5.2 Higher-order Removal for Accumulative Arguments

Let us turn to removing the higher-order accumulation of `sumCPS` in the previous subsection using IO swapping and η-expansion. First, we apply IO swapping to `sumCPS`. Because `sumCPS` is an instance of `foldl`,

```
sumCPS x = foldl (\k a v->k(a+v)) id x 0
```

we use Theorem 3 and obtain the following program.

```
sumCPS' x = let ([], k) = sum_n' x in k 0
  where sum_n' [] = (x, id)
        sum_n' (_:y) = let (a:z,k) = sum_n' y
                       in  (z,\v->k(a+v))
```

In `sumCPS'`, higher-order values only appear in the results, and applying η-expansion is sufficient for higher-order removal. Recall that η-expansion is the rule to pass an extra argument to the higher-order result. We define a function `sum_n''` that passes an extra argument to the second element of the result of `sum_n'` as follows.

```
sum_n'' y v = let (x',k) = sum_n' y in (x',k v)
```

Replacing `sum_n'` in the definition of `sumCPS'` with `sum_n''` yields the following program.

```
sumCPS' x = let ([], k) = sum_n'' x 0 in k
  where sum_n'' [] v = (x, v)
        sum_n'' (_:y) v = let (a:z,k) = sum_n'' y (a+v)
                          in  (z,k)
```

Higher-order removal is achieved.

We may go further. Since the effect of IO swapping is no longer needed, we eliminate it. Applying Theorem 4 backwards yields the following program.

```
sumCPS x = sum'' x
  where sum'' [] = 0
        sum'' (a:x') = let v = sum'' x'
                       in a+v
```

This is the usual definition of the function summing up all elements of a list. Our strategy, namely applying IO swapping after η-expansion after IO swapping, works successfully.

Let us summarize the transformation above as a formal rule. The point of derivation of an IO-swapped transformation is the step where we apply the original transformation (in this case η-expansion) to the result of IO swapping. Recall that the result for Theorem 4 is the following function.

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

If we can define the rule for η-expansion for this function, we can then obtain a higher-order removal rule for accumulative arguments. Although it is not so obvious, we can achieve this by clarifying the intersection for the range of IO swapping and the domain of η-expansion. We then obtain the following lemma.

Lemma 1. For suitably typed functions `g0`, `g1`, `g2`, `g3`, and `g4`, the following two functions, `f2a` and `f2b`, are equivalent.

```
 {- g0::r->v->h, g1::h->r, g2::a->r->v->h->h -}
 {- g3::a->r->r, g4::a->r->v->v              -}

f2a :: [a] -> v -> r
f2a x v0 = let ([],h,r') = fa' x (g1 (h v0)) in r'
  where  {- fa' :: [a] -> r -> ([a], v->h, r) -}
    fa' [] r = (x, \v->g0 r v, r)
    fa' (_:y) r = let (a:z,h,r') = fa' y (g3 a r)
                  in (z, \v->g2 a r v (h (g4 a r v)), r')

f2a :: [a] -> v -> r
f2b x v0 = let ([],h,r') = fb' x (g1 h) v0 in r'
  where  {- fb' :: [a] -> r -> v -> ([a], h, r) -}
    fb' [] r v = (x, g0 r v, r)
    fb' (_:y) r v = let (a:z,h,r') = fb' y (g3 a r) (g4 a r v)
                    in (z, g2 a r v h, r')
```

Proof. This is proved by η-expansion, similar to the case of `sumCPS'` above. Starting from `f2a`, we define `fb'` as follows.

```
fb' y r v = let (z,h,r') = fa' y r in (z,h v,r')
```

Then we replace `fa'` with `fb'`. We then obtain `f2b`. □

We are ready to derive the higher-order removal rule for function arguments. Applying IO swapping to both `f2a` and `f2b`, we obtain the following theorem.

Theorem 5 (Higher-order Removal for Function Arguments). For suitably typed functions g0, g1, g2, g3, and g4, the following two functions, `f1a` and `f1b`, are equivalent.

```
 {- g0::r->v->h, g1::h->r, g2::a->r->v->h->h -}
 {- g3::a->r->r, g4::a->r->v->v              -}

f1a :: [a] -> v -> r
f1a x v0 = let r = fa x (\v->g0 r v) in r
  where  {- fa :: [a] -> (v->h) -> r -}
    fa [] h = g1 (h v0)
    fa (a:z) h = let r = fa z (\v->g2 a r v (h (g4 a r v)))
                 in  g3 a r

f1b :: [a] -> v -> r
f1b x v0 = let (r,v) = fb x (g0  v) in r
  where   {- fb :: [a] -> h -> (r,v) -}
    fb [] h = (g1 h, v0)
    fb (a:z) h = let (r,v) = fb z (g2 a r v h)
                 in  (g3 a r, g4 a r v)
```

Proof. From Lemma 1, currying the arguments of `fb'` to create a triple, and applying IO swapping backwards to both `f2a` and `f2b`, we obtain `f1a` and `f1b` respectively. □

We can use our strategy for other program transformations such as fusion, as discussed in [10].

6 Related Work

We demonstrated the derivation of two palindrome detecting functions, `pld1` and `pld2`, in Section 4. These palindrome detecting functions were given by Danvy and Goldberg [12] as an application of the *There And Back Again* (TABA) pattern. What we demonstrated in Section 4 is, therefore, a derivation of TABA programs based on IO swapping. IO swapping derives TABA programs, on the one hand, because IO swapping turns an iteration over arguments into an iteration over results [13, 14]. IO swapping, on the other hand, is itself an application of TABA pattern; the TABA pattern is necessary for expressing the IO swapping rule. It is worth mentioning that another method based on *defunctionalization* [15] was proposed [16] to derive TABA programs. Although this defunctionalization-based method certainly derives `pld1`, it is not obvious whether it can cope with `pld2`.

While it is well known in the functional community that it is difficult to manipulate accumulative programs, we demonstrated in Section 5 a derivation of a manipulation method that could deal with accumulative programs. We found that a combination of the derived method (Theorem 5) and η-expansion works in a similar fashion to the higher-order removal method proposed by Nishimura [17] on the basis of a composition method [18] of attribute grammars [19]. Attribute grammars give a good abstraction of accumulative programs, and many attribute-grammar-based program transformation methods for accumulative functions have been proposed [20–23]. The reason attribute grammars make manipulations of accumulative functions easy is the symmetric treatments over arguments and results, and this is also what IO swapping aims at.

IO swapping is related to circular programs [4]. There have not been many studies on the application and transformation of circular programs in functional area, since circularities are not intuitive and disturb program manipulation. IO swapping offers the view that circularities, i.e., computational dependencies from results to arguments, are IO-swapped variants of accumulations, which expresses computational dependencies from arguments to results.

IO swapping is related to logic programming or relation-based programs to some extent. From the viewpoint of logic programming, what IO swapping does is to change the order in which a proof tree is constructed. If the original program constructs the proof tree from its root to its leaves, the IO swapped program constructs it from its leaves to its root, but the resulting tree is the same.

Although IO swapping seems related to the inversion of evaluation order [24], our work bears little relationship to it. IO swapping does not change the order of evaluations, but changes the dependency of computation: IO-swapped functions usually compute arguments after results, while ordinary functions compute results after arguments.

7 Conclusion

In this paper, we introduced a novel program transformation, namely IO swapping. IO swapping enables us to rearrange arguments and results to be suitable for manipulation. We demonstrated its effectiveness through two examples, the derivations of efficient palindrome detecting functions, and a higher-order removal transformation to defunctionalize function arguments.

We are currently attempting to extend IO swapping so that it can deal with non-linear recursions. Although many calculational rules have been extended to non-linear recursions using a framework of *constructive algorithmics* [3], we have not yet succeeded in describing the IO swapping rule in terms of constructive algorithmics.

We also consider that IO swapping is related to the synthesis of data structures. IO swapping for list catamorphisms produces a new function, scanning a list from tail to head. In general, IO swapping produces a new function that scans a queue-fashion data structure from an ordinary list-iterating function. It is much more difficult to manipulate queues than lists in a purely functional setting. We hope that IO swapping will enable data structures to be synthesised, e.g., the synthesis of list-like data structure such as queues, doubly linked lists, and circular lists.

Acknowledgements

We are very grateful to Olivier Danvy for introducing us to the TABA work and explaining its relation to defunctionalization and CPS transformation, and to Shin-Cheng Mu and Keisuke Nakano for their inspiring discussions. We are also very grateful to Jeremy Gibbons for editing this paper and giving helpful advice.

References

1. Bird, R., de Moor, O.: Algebra of Programming. Prentice Hall (1996)
2. Bird, R.: Algebraic identities for program calculation. Computer Journal **32**(2) (1989) 122–126
3. Meijer, E., Fokkinga, M., Paterson, R.: Functional programming with bananas, lenses, envelopes and barbed wire. In: Proceedings of the 5th ACM International Conference on Functional Programming Languages and Computer Architecture, FPCA'91. (1991) 124–144
4. Bird, R.: Using circular programs to eliminate multiple traversals of data. Acta Informatica **21** (1984) 239–250
5. Chin, W.N.: Towards an automated tupling strategy. In: Proceedings of the ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation, PEPM'93. 119–132
6. Pettorossi, A., Proietti, M.: Rules and strategies for transforming functional and logic programs. ACM Computing Surveys **28**(2) (1996) 360–414
7. Hu, Z., Iwasaki, H., Takeichi, M., Takano, A.: Tupling calculation eliminates multiple data traversals. In: Proceedings of the 2nd ACM SIGPLAN International Conference on Functional Programming ICFP'97. (1997) 164–175
8. Bird, R.: Introduction to Functional Programming using Haskell. Series in Computer Science. Prentice Hall (1998)

9. Hu, Z., Iwasaki, H., Takeichi, M.: Deriving structural hylomorphisms from recursive definitions. In: Proceedings of the 1st ACM SIGPLAN International Conference on Functional Programming, ICFP'96. (1996) 73–82
10. Morihata, A.: Relationship between arguments and results of recursive functions. Master's thesis, University of Tokyo (2006)
11. Chin, W.N.: Fully lazy higher-order removal. In: Proceedings of the ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation, PEPM'92. (1992) 38–47
12. Danvy, O., Goldberg, M.: There and back again. In: Proceedings of the 7th ACM SIGPLAN International Conference on Functional programming, ICFP'02. (2002) 230–234
13. Morihata, A., Kakehi, K., Hu, Z., Takeichi, M.: IO swapping leads you there and back again. In: Proceedings of the 7th Generative Programming and Component Engineering Young Researchers Workshop, GPCE-YRW'05. (2005) 7–13 Extended abstract of [14].
14. Morihata, A., Kakehi, K., Hu, Z., Takeichi, M.: Reversing iterations: IO swapping leads you there and back again. *Technical Report METR* 2005-11, Department of Mathematical Informatics, University of Tokyo. (2005)
15. Reynolds, J.C.: Definitional interpreters for higher-order programming languages. Higher-Order and Symbolic Computation **11**(4) (1998) 363–397 Reprinted from the proceedings of the 25th ACM National Conference (1972), with a foreword.
16. Danvy, O., Goldberg, M.: There and back again. Fundamenta Informaticae **66**(4) (2005) 397–413
17. Nishimura, S.: Fusion with stacks and accumulating parameters. In: Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM'04. (2004) 101–112
18. Ganzinger, H., Giegerich, R.: Attribute coupled grammars. In: Proceedings of the of the 1984 SIGPLAN symposium on Compiler construction. (1984) 157–170
19. Knuth, D.E.: Semantics of context-free languages. Mathematical Systems Theory **2**(2) (1968) 127–145
20. Kühnemann, A.: 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. (1998) 146–157
21. Kühnemann, A.: 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. (1999) 114–130
22. Correnson, L., Duris, E., Parigot, D., Roussel, G.: Declarative program transformation: A deforestation case-study. In: Proceedings of the 1st International Conference on Principles and Practice of Declarative Programming, PPDP'99. (1999) 360–377
23. Voigtländer, J.: Using circular programs to deforest in accumulating parameters. Higher-Order and Symbolic Computation **17**(1-2) (2004) 129–163
24. Boiten, E.A.: Improving recursive functions by inverting the order of evaluation. Science of Computer Programming **18**(2) (1992) 139–179