Relationship between Arguments and Results of Recursive Functions

（再帰関数の引数と返値の関係に関する研究）

Akimasa Morihata（森畑 明昌）
Supervisor: Professor Masato Takeichi

1. BACKGROUND

Since computers were invented, they have made remarkable progress. Their number, power, application, and theory have grown every year. Along with increasing importance of computers, the importance of programs, especially correct and efficient ones, have been increasing. But here we have a serious dilemma. On one hand, we need to produce involved programs that are efficient. On the other hand, we need to produce simple programs to confirm correctness. We cannot produce correct and efficient programs in naive ways.

To solve this dilemma, *calculational programming* [2] (or *program calculation*) is proposed. In calculational programming, we first construct a correct program that may be terribly inefficient, and after that we improve its efficiency with program manipulation technique. The methodology of calculational programming is a guidepost of program construction; calculational programming gives a global map, while program manipulation methods work as road signs. Calculational programming has succeeded in developing various kinds of programs, and what we need are more road signs to show a proper route.

2. ASYMMETRY BETWEEN ARGUMENTS AND RESULTS

In functional programming, arguments (inputs of functions) and results (outputs of functions) are not symmetric. Things being natural for arguments may not be natural for results, and vice versa. Here we give three examples of such asymmetry that are not suitable for program construction and program manipulation.

(1) Asymmetry of program elements

Usually programs iterate their computations over arguments. For example, the function `reverse`, which reverses an input list, is programed as follows.

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

The function `reverse` iterates its auxiliary function `rev` over its input list. It is a quite usual description of recursive functions, and many theories and techniques have been introduced to recognize and manipulate such programs [6][7][5]. In contrast, Danvy and Goldberg proposed a program pattern *There And Back Again* [3] (or in short, *TABA*) where programs iterate their computations over its results. For example, we can program `rev_n` that is actually `reverse` of TABA pattern as follows.

```
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')
```

The function `rev_n` iterates the computation of `rev'` over its results. Though TABA programs have nothing strange except for iteration over results, they are interesting but puzzling. It is not clear that how to recognize, how to use, how to analyze, and how to manipulate such programs.

(2) Asymmetry of computation dependencies

Usually a program computes its results from its arguments. In contrast, *circular programs* [1], use their results as their arguments as follows:

```
repmin t = let (r,m) = aux t m in r
  where
    aux (Node l r) m = let (lr, lm) = aux l m
                           (rr, rm) = aux r m
                       in (Node lr rr, min lm rm)
    aux (Leaf n) m = (Leaf m, n)
```

where `repmin` takes a tree and replaces the values of leaves by the minimum value in the tree. In this program, the variable `m` is computed by a function call `aux t m` where `m` is also used as an input of `aux`. To be precise, the `aux` computes its arguments from its results. It is not intuitive, and raises similar problems with TABA.

(3) Asymmetry of program manipulations:

In many cases, program manipulation methods that naturally fit to results are not directly applicable to arguments and vice versa. For example, consider the problem of higher-order removal. It is well known that η-expansion effectively achieves higher-order removal of results. For the following function `sumH`, whose auxiliary function `sum'` returns a function value,

```
sumH x = let r = sum' x in r 0
  where sum' [] = id
        sum' (a:x) = \h->a+(sum' x h)
```

η-expansion immediately gives a first-order definition as follows.

```
sumH' x = let r = sum' x 0 in r
  where sum' [] h = h
        sum' (a:x) h = a+(sum' x h)
```

In spite of such an effective use for higher-order removal of results, η-expansion can do nothing for accumulative arguments that produce function values. For example, it cannot work for the following `sumCP` function.

```
sumCP x = sum' x id
  where sum' [] r = r 0
        sum' (a:x) r = sum' x (\h->a+(r h))
```

In short, if we define one program transformation, we might have to prepare two versions, one for arguments and the other for results.

Such kinds of asymmetry disturb construction and manipulation programs. We have been suffering from them. What we need is a criterion that will be a guidepost so that we would not lose our way by these kinds of asymmetry.

3. IO SWAPPING

Here we introduce a novel program transformation called *IO swapping*. IO swapping makes a new recursive function whose call-time computations (computations managed in arguments) and return-time computations (computations managed in results) are the return-time computations and call-time computations of the old one, respectively, yet guarantees that the old and new recursive functions compute the same value.

THEOREM 1 (IO SWAPPING).
Assume that `g0`, `g1`, `g2`, and `g3` are given functions. Then the following two functions `f1` and `f2` are equivalent.

```
f1 x h0 = let r = f1' (x, g0 r h0) in r
  where
    f1' (x',h)
       = if p x' then g1 x' h
         else let r = f1' (k x', g2 x' r h)
              in g3 x' r h

f2 x h0 = let ((x',h),r') = f2' (x, g1 x' h)
           in r'
  where
    f2' (y,r)
       = if p y then ((x, g0 r h0),r)
         else
         let ((x',h),r') = f2' (k y, g3 x' r h)
         in ((k x', g2 x' r h),r')
```

□

Theorem 1 swaps the call-time computations and the return-time computations of the auxiliary function. In the definition of the function `f1`, `g3` performs the return-time computation, but in the definition of the function `f2` it does the call-time computation. In contrast, `g2` manages the call-time computation in the function `f1`, but under `f2` it does the return-time computation. Actually, we can derive `rev_n` above from usual `reverse` using IO swapping for example. Now we never suffer from the asymmetry of both program elements and computation dependencies, because we can exchange arguments with results by IO swapping.

IO swapping is easy to be combined with other program manipulation techniques. Moreover, IO swapping works as a *meta transformation* with other program manipulation techniques. A program manipulation with IO swapping becomes a new program manipulation which is IO-swapped manipulation of the old one. For example, we can derive a higher-order removal method for accumulative arguments from η-expansion with IO swapping. This higher-order removal method transforms the `sumCP` function above into the following usual first-order definition.

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

Now we do not suffer from the asymmetry of program manipulations anymore.

4. CONCLUSION

We introduced a novel program transformation namely IO swapping. It gives symmetry of arguments and results to recursive functions. With IO swapping, we could symmetrize not only program elements or computational dependencies, but also program manipulations. We confirmed its effectiveness through many examples: We demonstrated TABA program derivations and manipulations. We derived higher-order removal of accumulative arguments and fusion for accumulative programs from these of the results, and discussed its extension to the case of non-linear recursions. We proposed a guideline of manipulating circular programs and clarified the difficulty to manipulate them.

Now we are trying to clarify the relationship between IO swapping and theories of structural recursions. The theories of structured recursions are researched in terms of *constructive algorithmics* [6][4], where programs are abstracted using the theory of categories in mathematics. The framework of constructive algorithmics is very powerful so that many program transformaton methods are actually described [6][7][5]. But to the best of our knowledge, no research gives a proper abstraction to accumulative functions, and in fact we have not succeeded in describing the IO swapping rule in terms of constructive algorithmics yet.

We also consider that IO swapping is related with synthesis of data structures. IO swapping for structural recursions on lists produces a new function scanning a list from tail to head. In other words, it produces a new function that scans a queue-fashion data structure from ordinary lists iterating function. It is known that manipulation of queues is difficult in purely functional setting. We hope that IO swapping makes a room for the synthesis of data structures, for example a synthesis of list-like data structures such as queues, doubly linked lists, circular lists, etc.

5. REFERENCES

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

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

[3] O. Danvy and M. Goldberg. There and back again. In *Proceedings of ICFP'02*, pages 230–234. 2002.

[4] M. M. Fokkinga. *Law and Order in Algorithmics.* PhD thesis, University of Twente, Dept INF. 1992.

[5] Z. Hu, H. Iwasaki, M. Takeichi, and A. Takano. Tupling calculation eliminates multiple data traversals. In *Proceedings of ICFP'97*, pages 164–175. 1997.

[6] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Proceedings of FPCA'91*, pages 124–144. 1991.

[7] A. Takano and E. Meijer. Shortcut deforestation in calculational form. In *Proceedings of FPCA'95* pages 306–313. 1995.