Master Thesis

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

Department of Mathematical Informatics
University of Tokyo

January 27, 2006

Abstract

In functional programming, arguments (inputs of functions) and results (outputs of functions) are not symmetric. Things being natural and suitable for arguments may not be natural for results, and vice versa. Such asymmetry is not suitable for program construction and program manipulation. We need some guideposts so that we would not lose our way by these kinds of asymmetry.

In this thesis, we introduce a novel program transformation called *IO swapping*, which 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. IO swapping therefore introduces symmetry of arguments and results at the level of the program elements. Moreover, IO swapping is easy to combine with other program manipulation techniques and it enable to derive the IO-swapped manipulation of an existing one. Using IO swapping, we do not suffer from the asymmetry of program manipulations anymore.

Contents

Chapter 1

Introduction

1.1 Background

Since computers were invented, they have made remarkable progress. Their number, power, application, and theory have grown every year. Today they are indispensable to our lives. Large and small computers support our lives everywhere. No system works properly without computers.

Along with increasing importance of computers, the importance of programs, especially correct and efficient ones, have been increasing. We hope that our programs work correctly but use a small amount of computational resources. 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* [BdM96] (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, which shows where we start and where we want to go, while program manipulation methods work as road signs, which show a prospect of improving efficiency. Calculational programming has succeeded in developing various kinds of programs, and what we need are more road signs to show a proper route.

1.2 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 takes a lists $[a_1, a_2, \ldots, a_n]$ and reverses it as $[a_n, a_{n-1}, \ldots, a_1]$, 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 [Bir84a][MFP91][HIT99][CDPR99][Voi04]. In contrast, Danvy and Goldberg proposed a program pattern *There And Back Again* [DG02] (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* [Bir84b], 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. Actually, there is little research about manipulation of circular programs though they are introduced two decades ago.

(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 = = sum' x 0
  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 of programs. We have been suffering from them. But there seem some criteria to translate between something of arguments and that of results. What we need is a criterion that will be a guidepost so that we would not lose our way by these kinds of asymmetry.

In this thesis, 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. For example, we can derive `rev_n` above from usual `reverse` using IO swapping, and vice versa. 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. As we will see later in Section 6.2, 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.

1.3 Related Work

Our work is very closely related to TABA work of Danvy and Goldberg [DG02]. We found the IO swapping rule for deriving TABA programs systematically [MKHT05a][MKHT05b]; later we found that its effect was not restricted to derivation of TABA programs. The detailed discussions are going to be given in Chapter 5.

It is well known in functional community that manipulation of accumulative programs is troublesome. In contrast, attribute grammars (in short, AGs) [Knu68] give a good abstraction for accumulative programs. Many AG-based program transformation methods for accumulative functions have been proposed [Küh98][Küh99][CDPR99][Voi04], and part of a composition method of AGs was translated into the functional programming world as a higher-order removal method [Nis04]. AGs also have benefits on the treatments of circular programs [Joh87][Sar99]. However, AGs are not functional programs; expressiveness is different and the translation between AGs and functional programs is troublesome. After all, AG-based methods are hard to be combined with other functional-programming-based methods. One of our aims is to bridge the gap between the functional world and the world of AGs so that we can enjoy the benefits of AGs within the functional programming world, without jumping into the world of AGs. The reason why AGs make manipulations of accumulative functions easy is the symmetric treatments over arguments and results. This is what IO swapping aims for, and why our approach is able to enjoy the benefits of AGs. We are going to address about relationship among AGs, functional programing, and our works throughout this thesis, in particular in Chapters 2 and 3.

IO swapping is also related with circular programs [Bir84b]. There have been few researches about their application and transformation in functional community, since circularity is not intuitive and its existence disturbs manipulation of programs. IO swapping use circular programs intensively, and show that circularities are nothing but IO-swapped variants of accumulations. We are going to discuss the manipulation of circular programs in Chapter 7.

IO swapping has a relationship with logic programming or relation-based programs. From the viewpoint of logic programming, IO swapping manages to change the order to construct the proof tree. If the original program constructs a proof tree from its root to its leaf, IO-swapped one constructs it from its leaf to its root, but the resulting trees are the same. The relationship will be explained using the notation of relational AGs [DM93], in Section 4.1.

It looks possible to cast another view from inversion of the evaluation order [Boi92]. Yet our work has little relationship with it because IO swapping requires no inverse function. What IO swapping does is not to change the order of evaluations, but to change dependencies of computations: IO-swapped functions usually compute arguments after results, in contrast to ordinary functions which compute results after arguments.

1.4 Organization of This Thesis

The rest of this paper is organized as follows. Chapter 2 gives brief introduction to functional programming and attribute grammars, their notations and basic theories. We also explain about accumulative programs and circular programs that will be used intensively in this thesis. In Chapter 3, we explain the known fusion technique, which is one of the most important kinds of program transformation. We introduce many fusion methods and discuss their power for manipulating accumulative and circular programs. In Chapter 4, we propose our novel program transformation IO swapping. We explain its main idea and discuss its properties. In Chapter 5, we demonstrate the effect of IO swapping by deriving and manipulating TABA programs, and discuss relationship between IO swapping and TABA programs. In Chapter 6, we explain the use of IO swapping as a meta program transformation, which makes manipulations of arguments and results symmetrical. In Chapter 7, we discuss manipulation of circular programs. Finally we conclude our thesis in Chapter 8.

Chapter 2

Preliminaries

In this chapter, we will give a brief introduction of functional programming and attribute grammars. Each framework gives a theoretical foundation of programs, and leads program manipulation methods that we will see in Chapter 3. We are also going to explain about accumulative programs and circular programs whose manipulations are our purposes. We will show two views of them, one is from functional programming and another is from attribute grammars.

2.1 Functional Programming

In this section, we are going to explain about a taste of functional programming. Starting from the brief introduction of lambda calculus, the underlying model of computation of functional programming, we will introduce the notion of Church encoding and structural recursions. Structural recursions play the central role for program construction and program manipulation. Furthermore they also show the correspondence between functional programming and attribute grammars that we are going to explain in Section 2.2. It may be better to refer some textbooks such as [Bar84][Bir98], because we will give only a brief introduction.

2.1.1 Lambda Calculus

Functional programming is based on lambda calculus. Lambda calculus is a framework where computation is expressed by reduction over lambda terms.

Definition 2.1.1 (Lambda terms)**.**
From a set of variables $\mathcal{V}$, the set of lambda terms Λ is defined recursively as follows.

$$\begin{array}{ll} v \in \Lambda & \text{if } v \in \mathcal{V} \\ MN \in \Lambda & \text{if } M, N \in \Lambda \\ \lambda v.M \in \Lambda & \text{if } v \in \mathcal{V} \text{ and } M \in \Lambda \end{array}$$

We call the situation of lambda terms of the second rule above as an *application of lambda terms*, and that of the third rule as a *lambda abstraction*.

The definition of lambda terms describes its syntactic representation only. Its semantics, that is to say its correspondence to computation, is come from a reduction rule over them, called β-reduction. Before defining β-reduction, we define some words for expressing the circumstance of lambda terms and variables.

Definition 2.1.2 (Occurrences of a lambda term)**.**
An occurrence of a lambda term is a sequence of $\{1, 2\}$. A set of occurrences of a lambda term M,

denoted by $\mathcal{O}(M)$, is defined as follows.

$$\begin{aligned}\mathcal{O}(x) &= \{\epsilon\}\\ \mathcal{O}(MN) &= \{\epsilon\} \cup \{1 \cdot u_M \mid u_M \in \mathcal{O}(M)\} \cup \{2 \cdot u_N \mid u_N \in \mathcal{O}(N)\}\\ \mathcal{O}(\lambda x.M) &= \{\epsilon\} \cup \{1 \cdot u_M \mid u_M \in \mathcal{O}(M)\}\end{aligned}$$

Definition 2.1.3 (Subterms of a lambda term)**.** *A subterm of a lambda term M, denoted by M/u, $u \in \mathcal{O}(M)$, is defined as follows.*

$$\begin{aligned}M/\epsilon &= M\\ MN/1 \cdot u &= M/u\\ MN/2 \cdot u &= N/u\\ \lambda x.M/1 \cdot u &= M/u\end{aligned}$$

Definition 2.1.4 (Free variables and bound variables)**.**
A variable x of a subterm of a lambda term M such that $x = M/u, u \in \mathcal{O}(M)$, is said to be free *if $M/v \neq \lambda x.(M/v \cdot 1)$ for all $v < u$. A set of all free variables in a lambda term M is denoted by $\mathcal{FV}(M)$.*

A variable x of a subterm of a lambda term M such that $x = M/u, u \in \mathcal{O}(M)$, is said to be bound *if it is not free. A lambda abstraction $\lambda x.(M/v \cdot 1)$ is said to be the* binder *of x if $v < u$ and there is no v' such that $\lambda x.(M/v' \cdot 1)$ and $v < v' < u$.*

Definition 2.1.5 (Closed lambda terms)**.**
A lambda term M is said to be closed *if $\mathcal{FV}(M) = \emptyset$.*

Now we are ready for defining β-reduction.

Definition 2.1.6 (β-reduction)**.**
From a variable x and lambda terms M and N, β-reduction is defined by the following procedure beta.

$$\begin{array}{lcll}(\lambda x.M)N \Rightarrow_\beta beta(M,N,x) & & & \\ beta(x,N,x) & = & N & \\ beta(y,N,x) & = & y & y \neq x\\ beta(M_1M_2,N,x) & = & beta(M_1,N,x)beta(M_2,N,x) & \\ beta(\lambda x.M,N,x) & = & \lambda x.M & \\ beta(\lambda y.M,N,x) & = & \lambda y.beta(M,N,x) & y \neq x,\ y \notin \mathcal{FV}(N)\\ beta(\lambda y.M,N,x) & = & \lambda z.beta(M[z/y],N,x) & y \neq x,\ z \notin \mathcal{FV}(N)\end{array}$$

The transitive closure of $\Rightarrow_\beta$ is denoted by $\Rightarrow_\beta^*$.

Roughly speaking, β-reduction for lambda application $(\lambda x.M)N$ is rewriting where all free variables x in M, that is to say all variables whose binder is $\lambda x.M$, are substituted by N. Note that β-reduction affect nothing to closed subterms of the applied term.

β-reduction rule shows a correspondence between lambda calculus and computation. Lambda terms are functions and β-reductions are computation of function applications. We will explain the correspondence through examples. Assume that natural numbers and arithmetic $(+)$ are defined in terms of closed lambda terms, for we will show the representation of them in the next subsection. Then, a function *one*, which is a constant function and returns 1 for any argument, is expressed as the following lambda term,

$$one = \lambda x.1$$

where we read it as follows: $\lambda x.$ means "this term takes an argument x" and 1 means "and returns 1". β-reduction rule shows that it certainly returns 1 for any argument v as follows.

$$\begin{array}{rcl} one(v) & = & (\lambda x.1)\ v \\ & \Rightarrow_\beta & beta(1, v, x) \\ & = & 1 \end{array}$$

Note that the procedure *beta* do not rewrite 1 because it is a closed lambda term.

And a function *double*, which takes a natural number n and computes its double namely $n + n$, is expressed as the following lambda term:

$$double = \lambda x.x + x$$

because of the following reduction.

$$\begin{array}{rcl} double(n) & = & (\lambda x.x + x)\ n \\ & \Rightarrow_\beta & beta(x + x, n, x) \\ & \Rightarrow & \quad \{\text{- } (+) \text{ is a closed lambda term -}\} \\ & & beta(x, n, x) + beta(x, n, x) \\ & = & n + n \end{array}$$

Furthermore, a function *plus*, which takes two natural number n and m and computes their sum, namely $n + m$, is expressed as the following lambda term:

$$plus = \lambda x.\lambda y.x + y$$

because of the following reduction.

$$\begin{array}{rcl} plus(n)(m) & = & ((\lambda x.\lambda y.x + y)\ n)\ m \\ & \Rightarrow_\beta & beta(\lambda y.x + y, n, x)\ m \\ & = & (\lambda y.n + y)m \\ & \Rightarrow_\beta & beta(n + y, m, y) \\ & = & n + m \end{array}$$

These examples show that lambda abstractions are able to describe function values: The number of lambda abstractions corresponds to the number of arguments. Applications of lambda terms correspond to function applications. β-reduction corresponds to computation of function applications. Now we can regard lambda terms with β-reduction as a computation model where function values are a first class citizen. It is called *lambda calculus*, which is the underlying computation model of functional programming. In lambda calculus, lambda terms are called *lambda expressions*.

Usual lambda calculus has two more rules, called α-renaming and η-expansion.

Definition 2.1.7 (α-renaming)**.**
From variables x, y and a lambda term M, α-renaming is defined as follows.

$$\lambda x.M \ \Rightarrow_\alpha \ \lambda y.beta(M, x, y)$$

Definition 2.1.8 (η-expansion)**.**
From a variable x and a lambda term M, where x is not free in M, then η-expansion is defined as follows.

$$M \Rightarrow_\eta \lambda x.Mx$$

α-renaming and η-expansion rules do not change the semantics of lambda terms given by β-reduction. Define an equivalence relation $=_{\alpha\eta}$ by the reflective transitive closure of α-renaming and η-expansion. Then $M =_{\alpha\eta} M'$ and $N =_{\alpha\eta} N'$ implies that the result of β-reduction over the application MN is $=_{\alpha\eta}$-equivalent to the result of β-reduction of $M'N'$. We can recognize that $=_{\alpha\eta}$ gives a semantic equivalence of lambda terms.

From now on we use some abbreviations for lambda terms. Nested lambda abstractions are merged into one; for example $\lambda x\ y\ z.M$ is equivalent to $\lambda x.\lambda y.\lambda z.M$. We consider that application of lambda terms are left associative and give no parenthesis for a sequence of applications; for example PQR is equivalent to $(PQ)R$.

Though we have explained that β-reduction gives a semantics of lambda terms, there are many choices of which application we reduce first. For example, think about the following lambda term:

$$(\lambda a.\ (\lambda d.\ d)\ ((\lambda b.\ a\ b)\ (\lambda c.\ c\ a)))$$

There are two applications of lambda terms in this term. Without determining which one we reduce first, we cannot give a complete semantics of them. There are two well-known strategies of reduction, called *strict evaluation* and *lazy evaluation*.

In strict evaluation, we should reduce the outermost application of lambda terms such that it is not inside of any lambda abstractions and the later one of lambda terms which compose a lambda application is reduced to be a term whose root is not a lambda application. If we choose strict evaluation strategy, then we compute the example above as follows.

$$\begin{aligned}
(\lambda a.\ a)\ ((\lambda b.\ b\ b)\ (\lambda c.\ (\lambda d.\ d)\ c)) &\Rightarrow_\beta (\lambda a.\ a)\ ((\lambda c.\ (\lambda d.\ d)\ c)\ (\lambda c.\ (\lambda d.\ d)\ c)) \\
&\Rightarrow_\beta (\lambda a.\ a)\ ((\lambda d.\ d)\ (\lambda c.\ (\lambda d.\ d)\ c)) \\
&\Rightarrow_\beta (\lambda a.\ a)\ (\lambda c.\ (\lambda d.\ d)\ c) \\
&\Rightarrow_\beta (\lambda c.\ (\lambda d.\ d)\ c)
\end{aligned}$$

In lazy evaluation, we should reduce outermost application of lambda terms such that it is not inside of any lambda abstractions. If we choose lazy evaluation strategy, then we compute the example above as follows.

$$\begin{aligned}
(\lambda a.\ a)\ ((\lambda b.\ b\ b)\ (\lambda c.\ (\lambda d.\ d)\ c)) &\Rightarrow_\beta (\lambda b.\ b\ b)\ (\lambda c.\ (\lambda d.\ d)\ c) \\
&\Rightarrow_\beta (\lambda c.\ (\lambda d.\ d)\ c)\ (\lambda c.\ (\lambda d.\ d)\ c) \\
&\Rightarrow_\beta (\lambda d.\ d)\ (\lambda c.\ (\lambda d.\ d)\ c) \\
&\Rightarrow_\beta (\lambda c.\ (\lambda d.\ d)\ c)
\end{aligned}$$

2.1.2 Church Encoding and Structural Recursion

We have shown that lambda terms and β-reduction give a computational model where functions are first class citizens. In other words everything is a function in lambda calculus. It seems too restrictive to use it for practical programming because there are neither primitive values nor data structures. We are going to show that we can encode both primitive values and data structures by lambda expressions.

First think about boolean values, namely *True* and *False*. In many systems or programming languages boolean values are encoded by integers. For example zero means *False* and non-zero means *True* in programming language C. This is because the most important point is not implementations of

boolean values but the fact that we can distinguish *True* from *False* easily. This fact implies that we can encode boolean values in terms of lambda expressions by defining correspondence between lambda terms and boolean values appropriately such that we can distinguish *True* from *False*. We propose an appropriate implementation of boolean values by lambda expressions. *True* takes two branches and selects the first one. *False* also takes two branch, and selects the second one.

$$\begin{array}{lcl} \mathit{True} & = & \lambda x\ y.\ x \\ \mathit{False} & = & \lambda x\ y.\ y \end{array}$$

It is proper because it fits the use of boolean values. Boolean values are necessary for the *if* expression that branches off a computation according to a boolean value. This implementation gives a natural definition of the *if* expression as follows.

$$\mathit{if} \quad = \quad \lambda b\ x\ y.\ b\ x\ y$$

We can also encode other useful operations manipulating boolean values such as *and*, *or* and *not* by lambda expressions.

$$\begin{array}{lcl} \mathit{and} & = & \lambda a\ b.\ a\ b\ \mathit{False} \\ \mathit{or} & = & \lambda a\ b.\ a\ \mathit{True}\ b \\ \mathit{not} & = & \lambda a.\ (\lambda x\ y.\ a\ y\ x) \end{array}$$

We have given an encoding of boolean values as their manipulations. To say concrete, in spite of implementing the algebraic structure boolean values, we use homomorphisms on it. There are many way to construct the algebraic structure of boolean values. One of the simplest ways is to regard it as a structure where we have only two elements *True* and *False*, and have no operation. Then we can represent homomorphisms on it by a pair of functions, one corresponding to *True* and another corresponding to *False*. We use homomorphisms on their algebraic structure explicitly to express boolean values, while they in usual come out implicitly from the *if* expression.

We can also encode natural numbers in the similar way. The algebraic structure of natural number has one special value *Zero* and one operation *Succ* which takes a value and returns its successor. *Zero* and *Succ* are enough to construct it, because we can generate all natural numbers from them. Then we can represent homomorphisms by pair of functions, one for *Zero* and another for *Succ*. Now we encode *Zero* and *Succ*:

$$\begin{array}{lcl} \mathit{Zero} & = & \lambda x\ y.\ y \\ \mathit{Succ} & = & \lambda n.\ (\lambda x\ y.\ x\ (n\ x\ y)) \end{array}$$

From this definition, natural numbers are realized as follows:

$$\begin{array}{lcll} 1 & \equiv & \mathit{Succ}\ \mathit{Zero} & = \lambda x\ y.\ x\ y \\ 2 & \equiv & \mathit{Succ}(\mathit{Succ}\ \mathit{Zero}) & = \lambda x\ y.\ x(x\ y) \\ 3 & \equiv & \mathit{Succ}(\mathit{Succ}(\mathit{Succ}\ \mathit{Zero})) & = \lambda x\ y.\ x(x(x\ y)) \end{array}$$

A natural number n is implemented as the function that takes two functions and compute n times using the first one after an initialization by the second one. We can naturally implement *for* expression, because *for* corresponds to homomorphisms on natural numbers.

$$\mathit{for} = \lambda n\ x\ y.\ n\ x\ y$$

We can also apply the same method to encode data structures by lambda expressions. For example, think about lists. We construct lists from *Nil* corresponding to empty list, and *Cons* corresponding

to addition of an element to the head of a list. *Nil* and *Cons* define the algebraic structure of lists. Now we encode lists using homomorphisms on their algebraic structure as follows.

$$\begin{array}{lcl} \mathit{Nil} & = & \lambda x\ y.\ y \\ \mathit{Cons} & = & \lambda a\ r.\ (\lambda x\ y.\ x\ a(r\ x\ y)) \end{array}$$

For example, a list $[1, 2, 3]$ is represented as follows.

$$\begin{array}{lcl} [1,2,3] & \equiv & \mathit{Cons}\ 1\ (\mathit{Cons}\ 2\ (\mathit{Cons}\ 3\ \mathit{Nil})) \\ & = & \lambda x\ y.\ x\ 1(x\ 2(x\ 3\ y)) \end{array}$$

Such homomorphisms on an algebraic structure of a data structure are called *structural recursions* or *catamorphisms* [MFP91], and encoding of values or data structures using structural recursions is called *Church encoding*. Using Church encoding, we can introduce primitive values and data structures to lambda calculus where we can manipulate them by structural recursions. Church encoding and structural recursions enable lambda calculus to be modeling practical programs. While we do not use Church encoding in practical functional programming languages for efficiency, it gives a theoretical foundation.

Structural recursions are also important for program manipulation because of the following two reasons: First they are easy to manipulate because we can use the underlying algebraic structure. Secondly they are expressive because they come from the semantics of the underlying data structures. Later we will show how structural recursions work for program manipulation in Chapter 3.

2.1.3 Basic Notations for Functional Programs

Throughout this thesis, we use the notation of the functional programming language Haskell [Bir98] to express functional programs. Some syntactic notations we use are as follows. The symbol \ and `->` is used instead of λ and . 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)`. Parentheses are used to be *sectioning* an operation, making a function from an operation, i.e., `(+) 2 3 = 2 + 3 = (2+) 3 = (+3) 2`. We assume that evaluation is based on lazy evaluation and associativity of function applications are stronger than that of operators, and `x + f y` means `x + (f y)`. We use pattern-matching implicitly, for example, the meaning of `head (a:x) = a`, where `(a:x)` expresses a pattern-matching, is as follows: The function `head` takes a non-empty list. A binding is constructed from the pattern-matching where the variable `a` corresponds to the first element of the list and `x` corresponds to the rest of it. The function `head` returns `a`, namely the first element of the list.

Many standard Haskell functions are used in this paper, whose informal definitions are given in Figure 2.1. All of them are in the `Prelude` library of Haskell 98 [Has02] and their formal definitions are found in its manual.

We construct lists as usual from `[]` namely an empty list, and `(:)` namely adding an element to the head of a list. Using Haskell notation, it is expressed as follows.

```
data [a] = [] | a : [a]
```

Similarly, we construct leaf-valued binary trees from *`Leaf`* namely a leaf of tree having a value, and *`Node`* namely an internal node of tree having two subtrees, as follows.

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

A function is said to be *curried* if it takes its arguments one by one, such as `(\a b c->a+b+c)`. A function is said to be *uncurried* if it takes its whole arguments as a tuple, such as `(\(a,b,c)->a+b+c)`.

```
id x = x
fst (a,b) = a
snd (a,b) = b
uncurry f (a,b) = f a b
div n m = ⌊n/m⌋
min n m = if n < m then n else m
head [x0,x1,...,xn] = x0
tail [x0,x1,...,xn] = [x1,x2,...,xn]
[x0,x1,...,xm,...,xn] !! m = xm
[x0,x1,...,xn] ++ [y0,y1,...,yn] = [x0,x1,...,xn,y0,y1,...,yn]
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
sum [x0,x1,...,xn] = x0 + x1 + ... + xn
and [x0,x1,...,xn] = x0 && x1 && ... && xn
reverse [x0,x1,...,xn] = [xn,xn−1,...,x0]
cycle [x0,x1,...,xn] = [x0,x1,...,xn,x0,x1,...,xn,x0,x1,...]
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
```

Figure 2.1. Informal definitions of standard functions

We basically use curried notation, but we hardly distinguish these two notations and we shift one to another in some time.

We say a value is higher-order if it is a function. In some time we also call higher-order value as *closure*. We say a value is first-order if it is not higher-order; it is a primitive value or a data structure such as an integer, a boolean, a list, and so on.

2.2 Attribute Grammars

2.2.1 Attribute Grammars

Attribute grammars are first proposed by Knuth [Knu68] for denoting a semantics of context free grammars.

Definition 2.2.1 (Context free grammars)**.**
A context free grammar (or in short, CFG) is a triple $G = (V, P, S)$. V *is a finite non-empty set of* grammar symbols. $\Sigma \subset V$ *is a set of* terminal symbols. $N = V \setminus \Sigma$ *is a set of* non-terminal symbols. $P \subseteq N \times V^*$ *is a finite set of* production rules. $S \in N$ *is the* start symbol.

A CFG constructs a *syntax tree* from a sequence of terminal symbols. For example, think about the following CFG, whose terminal symbols are { `+`, `-`, *val* } where *val* corresponds to integers, a set of non-terminal symbols is {*Exp*}, the start symbol is *Exp*, and production rules are as follows.

$$\begin{array}{rcl} \mathit{Exp} & \to & \mathit{Exp}\ \texttt{+}\ \mathit{Exp} \\ & | & \mathit{Exp}\ \texttt{-}\ \mathit{Exp} \\ & | & \mathit{val} \end{array}$$

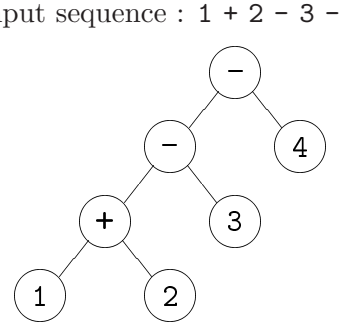

Figure 2.2. An example of syntax tree (Assume that all arithmetic operators are left associative)

From now on, we express a CFG only by a set of production rules if others are clear from context. The CFG above defines a syntax of arithmetic expressions where we have two arithmetic operator + and -. How it construct a syntax tree from a sequence of terminal symbols are shown in Figure 2.2. It clearly defines a structure of syntax, but it does not define its semantics. There is no explicit definition of it. Attribute grammars compose a framework to give a semantics for CFGs.

Definition 2.2.2 (Attribute grammars)**.**
An attribute grammar (or in short, AG) is a triple $AG = (G, A, D)$. $G = (V, P, S)$ *is a CFG.* A *is a finite set of* attributes*, partitioned into two sets* $A_{syn}(X)$ *and* $A_{inh}(X)$ *for each* $X \in V$. *Elements in* $A_{syn}(X)$ *are called as* synthesized attributes *and elements in* $A_{inh}(X)$ *is called as* inherited attributes. $D = (T, E)$ *is the* semantic domain *of AG where* T *is a finite set of types and* $E = \bigcup_{p \in P} E_p$ *is a finite set of semantic equations.*

Note that we usually assume that each element in E_p for $p \ : \ X_0 \to X_1 \cdots X_n$ is an equation of the form:

$$\alpha \ = \ f \ \beta_1 \ \beta_2 \ \cdots \ \beta_k$$

where f is a function called *semantic function*, α is an element of $(A_{syn}(X_0) \ \cup \ \bigcup_{i \in \{i \cdots n\}} A_{inh}(X_i))$, and each β_i satisfies $\beta_i \in (primitive\_values \ \cup \ \{X_0, X_1, \ldots, X_n\} \ \cup \ A_{inh}(X_0) \ \cup \ \bigcup_{i \in \{i \cdots n\}} A_{syn}(X_i))$.

An AG gives a semantics to a CFG by adding attributes to symbols and semantic equations to production rules. Now that we can bring values over the syntax tree constructed by CFG. The assumption about semantic equations gives some characteristics of synthesized attributes and inherited attributes: Synthesized attributes bring values from bottom to top of the syntax tree. Inherited attributes bring values from top to bottom of the syntax tree.

To give a semantics of a CFG by an AG, we should give a semantics to AGs. A semantics of AGs depends on the semantics of its semantic functions. In this thesis, we give a semantics of semantic functions by functional programing. It seems not so good, because we can write anything in semantic function and the framework of AGs is unnecessary, but we write in the framework of AGs as far as we can.

Now we give a semantics of the previous CFG for arithmetic expression as an example. We define $A_{inh}(X) = \emptyset$ and $A_{syn}(X) = \{r\}$ for all $X \in V$, $T = \{Integer\}$, and define E as follows.

$$\begin{array}{lcll} Exp & \to & Exp \ \texttt{+} \ Exp & \langle Exp_0.r = Exp_1.r + Exp_2.r \rangle \\ Exp & \to & Exp \ \texttt{-} \ Exp & \langle Exp_0.r = Exp_1.r - Exp_2.r \rangle \\ Exp & \to & val & \langle Exp.r = val \rangle \end{array}$$

This AG defines the semantics of the CFG. The value of the arithmetic expression X is surely expressed by its attribute denoted by r. Here $\langle\ \rangle$ denotes a set of semantic equations corresponding to the production rule standing its left. $X_i.r$ denotes an attribute r associating with a grammar symbol X of occurrence i. We count occurrence from left to right. Occurrences are omitted if it is clear from context. From now on, we express an AG as a set of production rules associating with semantic equations whenever others are clear from context.

2.2.2 Attribute Grammars as a Functional Programming Paradigm

AGs were first introduced to give a semantics to CFGs [Knu68]. They have been also used for expressing computations over structures. The AG of the example in the Section 2.2.1 defines a semantics of the CFG, but we can recognize that it describes a computation over a structure, namely a syntax tree expressed by the CFG. This recognition gives a relationship between AGs and functional programs, especially structural recursions. For example, think about the following function `reverse`.

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

We will express it in terms of an AG. First for all, an AG requires an underlying CFG. We chose a CFG for lists as the underlying CFG, because the auxiliary function `rev` is a structural recursion on lists and we can express the computation of `reverse` in terms of a computation over structures on lists. Now we show the following AG:

$$
\begin{array}{lll}
\mathit{reverse} \ \rightarrow \ \mathit{List} & & \langle \mathit{reverse.result} = \mathit{List}.r \\
 & & \quad \mathit{List}.h = \texttt{[]} \rangle \\
\mathit{List} \ \rightarrow \ \texttt{a} : \mathit{List} & & \langle \mathit{List}_0.r = \mathit{List}_1.r \\
 & & \quad \mathit{List}_1.h = \texttt{a}\!:\!\mathit{List}_0.h \rangle \\
\mathit{List} \ \rightarrow \ \texttt{[]} & & \langle \mathit{List}.r = \mathit{List}.h \rangle
\end{array}
$$

where the inherited attribute is h and the synthesized attributes are r and *results*. We show its the computation structure in Figure 2.3. We can recognize the correspondence between an AG and a structural recursion. The arguments of the recursion correspond to inherited attributes, except for the argument that corresponds to underlying data structure. The results of the recursion correspond to synthesized attributes. The underlying algebraic structure of the structural recursion corresponds to the underlying CFG of the AG, except for the *top case*. The top case is the production rule with associating semantic equations, which describes a production of the start symbol. As the example above, we use top case because it corresponds to call of the auxiliary function `rev`. Based this correspondence, many researches have done for clarifying and utilizing the relationship between functional programming and AGs [KS86][Joh87][FJMM91][DPRJ96][Sar99][Bac02]. There are also many researches about their relationship among program transformations of functional programming and AGs. We are going to introduce and discuss them in Chapter 3.

Although most of the researches use the correspondence that we have shown above, there is another way to make a correspondence between them. It is introduced by Deransart and Małuszyński [DM93] for giving a correspondence between AGs and logic programming. Using their method, an AG for

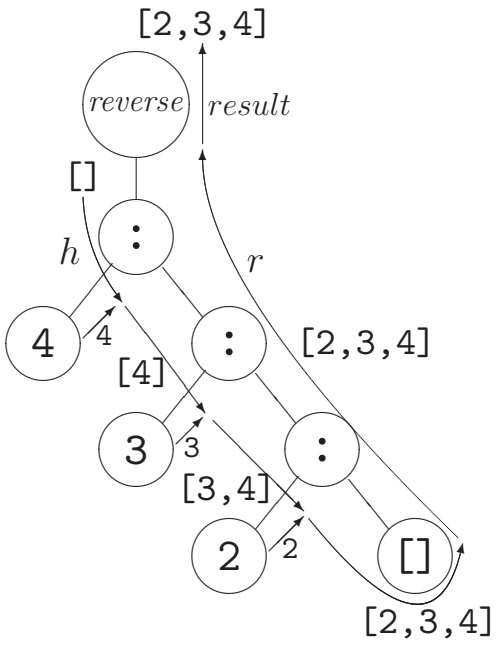

Figure 2.3. Computation structure of the AG representation of `reverse`

`reverse` is expressed as follows:

$$
\begin{array}{lll}
\mathit{reverse} \ \rightarrow \ \mathit{rev} & \langle \mathit{reverse.result} = \mathit{rev.r} \\
 & \quad \mathit{rev.x} = \mathit{reverse.x} \\
 & \quad \mathit{rev.h} = \texttt{[]} \rangle \\
\mathit{rev} \ \rightarrow \ \mathit{rev} & \langle \mathit{rev}_0\mathit{.r} = \mathit{rev}_1\mathit{.r} \\
 & \quad \texttt{a:x} = \mathit{rev}_0\mathit{.x} \\
 & \quad \mathit{rev}_1\mathit{.x} = \texttt{x} \\
 & \quad \mathit{rev}_1\mathit{.h} = \texttt{a:}\mathit{rev}_0\mathit{.h} \rangle \\
\mathit{rev} \ \rightarrow \ \epsilon & \langle \texttt{[]} = \mathit{rev.x} \\
 & \quad \mathit{rev.r} = \mathit{rev.h} \rangle
\end{array}
$$

This approach makes the correspondence between programs and AGs much clearer. Arguments certainly correspond to inherited attributes. The underlying CFG is the underlying call flow of the recursive function. Moreover, the expressiveness of AGs are not restricted to structural recursions. But in this approach, the production of the CFG depends on the semantic equations: Non-terminal symbol *rev* becomes ϵ if and only if the attribute *rev.x* is `[]`, and *rev* should produce *rev* for the other cases. Now we should treat semantic equations as constraints or logical relations, not as evaluation rule. The framework where semantic equations express logical relations is called as *relational AGs*. The relational AG based view seems a bit troublesome for manipulating functional programming, though it fits to logic programming very much.

We use the first correspondence between functional programming and AGs for discussing their computations and manipulations. We implicitly assume that there is much more general correspondence given by relational AGs. Later we are going to see that the first one gives a program transformation methods such as fusion and tupling in Chapter 3, and the second one gives a novel program transformation *IO swapping* in Chapter 4.

2.3 Accumulative Functions and Circular Functions

In this section, we are explaining about accumulative functions and circular functions. We will give their definition and discuss their characteristics, from viewpoints of both functional programming and AGs. The explanations about their manipulations will be given in Chapter 3.

2.3.1 Accumulative Functions

An accumulative function is, roughly speaking, a function that accumulates some values in its arguments. One of the typical examples is the following `reverse` function.

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

The function `reverse` is accumulative because the auxiliary function `rev` accumulates a list in its second argument. Strictly speaking, the function `rev` is accumulative but `reverse` is not, but we call `reverse` as accumulative because most of its computation is managed by an accumulative function `rev`. We introduce a notion about arguments.

Definition 2.3.1 (Recursion arguments and accumulative arguments)**.**
A set of arguments of a recursive function are recursion arguments *if the structure of the recursive call is determined according to their values. A set of arguments of a recursive function is* accumulative arguments *if the structure of the recursive call does not depend on their values.*

For example, the function `rev` above has one recursion argument namely the first argument, and one accumulative argument namely the second argument.

Now we can define a notion of recursive functions.

Definition 2.3.2 (Accumulative functions)**.**
A recursive function is said to be accumulative *if it has accumulative arguments.*

Tail-recursive (or *tail-call*) is a famous and practically important class of accumulative functions. Tail-recursive functions compute all its computations in *call-time*, namely at accumulative arguments. Consider the function `sum` for an example. On one hand, `sum` is usually implemented in terms of a structural recursion as follows:

```
sum [] = 0
sum (a:x) = a + sum x
```

where `sum` does all of its computations in *return-time*, namely at the result. On the other hand, `sum` is often implemented in terms of tail-recursive functions as follows:

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

where `sumTC` does all of its computations in call time. Tail-recursive functions are important because we can optimize it by realizing it by a loop.

All structural recursions are not accumulative. It is because a structural recursion takes exactly one recursion argument that expresses the structure of the recursion. But structural recursions that return a higher-order results are essentially equivalent to accumulative functions, because the higher-order results indicate a necessity of extra arguments to produce first-order results. For example, think about the following function `rev'`.

```
rev' [] = \h->h
rev' (a:x) = \h-> rev' x (a:h)
```

This function is a structural recursion over lists and produces a higher-order result. The result of `rev'` need an extra list to produce a list. Actually η-expansion enables us to transform it to a first-order accumulative function, which is the same as the `rev` above.

From the viewpoint of AGs, the characteristics of accumulative functions is clear. Every AGs having inherited attributes corresponds to accumulative functions. This is because the recursion structure of an AG is determined by the underlying CFG only and every inherited attributes corresponds to an accumulative argument. As we have seen in Section 2.2.2, AG representation of `reverse` function has an inherited attribute.

2.3.2 Circular Functions

A circular function, introduced by Bird [Bir84b], is a function that uses results of a certain recursive call as arguments of itself. The `repmin` problem, which is also introduced by Bird, is a typical application. The problem is to replace its values of leaves by the minimum value in the tree. We can implement it without using any circularity as follows.

```
transform t = replace t (tmin t)
    where replace (Node l r) m = Node (replace l m) (replace r m)
          replace (Leaf n) m = Leaf m
          tmin (Node l r) = min (tmin l) (tmin r)
          tmin (Leaf n) = n
```

Bird showed an alternative solution using a circular function 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)
```

In this program, the first call of `aux` contains circularities expressed in terms of variable `m`: The variable `m` is computed by a function call `aux t m` where `m` is also used as an input of `aux`. Before discussing detail, we will define a concept of circular functions.

Definition 2.3.3 (Circularity and circular functions)**.**
A circularity *is a situation where a result of a certain function call is also used as an argument of that function call, as follows:*

$$(r_0, r_1, \ldots, r_n) \;=\; f\ x_1\ x_2\ \cdots\ x_k\ r_i\ x_{k+1}\ \cdots\ x_m \qquad (0 \le i \le n)$$

A circularity is local *if it occurs inside of recursive calls. A circularity is* global *if it occurs outside of recursive calls.* Circular functions *are the functions that have circularities.*

The `repmin` function has a global circularity, and it has no local circularities because function `aux` is not a circular function.

Circular programs are evaluable under lazy evaluation. Under strict evaluation, we should know the value of `m` before evaluating `aux t m` but we need to evaluate `aux t m` for knowing the value of `m`. This conflict produces infinite recursion and gives no result. But lazy evaluation enables us to get the results. We will confirm it.

First, we express `repmin` as a structural recursion over trees to use church encoding to express computation over trees. The structural recursions over trees are expressed by the following function `foldTree`.

```
foldTree g1 g2 (Leaf n) = g2 n
foldTree g1 g2 (Node l r) = g1 (foldTree g1 g2 l) (foldTree g1 g2 r)
```

The function `foldTree` expresses homomorphisms over trees, because *Leaf* is replaced by `g2` and *Node* is replaced by `g1`. Now we express `repmin` by `foldTree` as follows.

```
repmin t = fst (repmin' t)
  where repmin' t = foldTree rn rl t m
        m = snd (repmin' t)
        rn l r = \m -> (Node (fst (l m)) (fst (r m)), min (snd (l m)) (snd (r m)))
        rl n = \m-> (Leaf m, n)
```

Now we try to evaluate it on a small tree *Node*(*Leaf* 1)(*Leaf* 2). We use church encoding of trees to express the structure of the tree and the structural recursion on it. The structure of trees is expressed by lambda terms as follows.

$$\begin{aligned} \mathit{Node} &= \lambda l\ r.\ (\lambda f\ g.\ f\ (l\ f\ g)\ (r\ f\ g)) \\ \mathit{Leaf} &= \lambda n.\ (\lambda f\ g.\ g\ n) \end{aligned}$$

To see the computation process, we compute `rempin'` instead of `repmin`. The computation is represented by the following reductions over lambda terms.

`repmin' (`*Node*`(`*Leaf* `1)(`*Leaf* `2))`
$\Rightarrow$ `foldTree rn rl (`*Node*`(`*Leaf* `1)(`*Leaf* `2)) (snd (repmin' (`*Node*`(`*Leaf* `1)(`*Leaf* `2))))`
$\equiv$ {- Church encoding -}
$(\lambda l\ r.\ (\lambda f\ g.\ f\ (l\ f\ g)\ (r\ f\ g)))(\lambda f\ g.\ g\ 1)(\lambda f\ g.\ g\ 2)$ `rn rl`
`(snd (repmin' (`*Node*`(`*Leaf* `1)(`*Leaf* `2))))`
$\Rightarrow^*_\beta$ `rn (rl 1) (rl 2) (snd (repmin' (`*Node*`(`*Leaf* `1)(`*Leaf* `2))))`
$\Rightarrow^*_\beta$ $(\lambda m.$ `(`*Node* `(fst (rl 1` m`)) (fst (rl 2` m`)), min (snd (rl 1` m`)) (snd (rl 2` m`))))`
`(snd (repmin' (`*Node*`(`*Leaf* `1)(`*Leaf* `2))))`

The expression (`repmin' (`*Node*`(`*Leaf* `1)(`*Leaf* `2))`) is denoted by r^* for notational convenience. We continue calculation as follows.

$\Rightarrow^*_\beta$ `(`*Node* `(fst (rl 1 (snd` r^*`))) (fst (rl 2 (snd` r^*`))),`
`min (snd (rl 1 (snd` r^*`))) (snd (rl 2 (snd` r^*`))))`
$\Rightarrow^*_\beta$ `(`*Node*`(fst (`*Leaf* `(snd` r^*`),1))(fst (`*Leaf* `(snd` r^*`),2)),`
`min(snd (`*Leaf* `(snd` r^*`),1))(snd (`*Leaf* `(snd` r^*`),2)))`
$\Rightarrow^*_\beta$ `(`*Node* `(`*Leaf* `(snd` r^*`)) (`*Leaf* `(snd` r^*`)), min 1 2)`
$\Rightarrow^*_\beta$ `(`*Node* `(`*Leaf* `(snd` r^*`)) (`*Leaf* `(snd` r^*`)), 1)`

Pay attention that `min (snd (`*Leaf* `(snd` r^*`),1)) (snd (`*Leaf* `(snd` r^*`),2))` do not use the value of r^*, and lazy evaluation enables us to reduce them into primitive values. We should do the same reduction as described above for r^*. It is apparent that we eventually get the second component of this term for r^*, because it comes from the same term. Now we substitute this term into r^*.

$\Rightarrow^*_\beta$ `(`*Node* `(`*Leaf*`(snd(`*Node* `(`*Leaf* `(snd` r^*`)) (`*Leaf* `(snd` r^*`)), 1)))`
`(`*Leaf*`(snd(`*Node* `(`*Leaf* `(snd` r^*`)) (`*Leaf* `(snd` r^*`)), 1))) , 1)`
$\Rightarrow^*_\beta$ `(`*Node* `(`*Leaf* `1) (`*Leaf* `1), 1)`

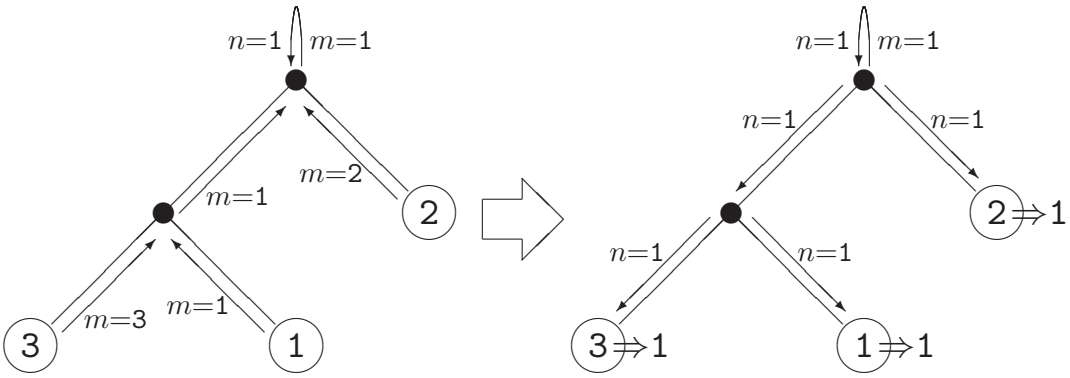

Figure 2.4. Computation structure of AG representation of `repmin`

Certainly we can get the correct result by lazy evaluation.

Circular functions are also accumulative functions, because circularities are always expressed by accumulative arguments. If circularities are expressed by recursion arguments, then we cannot decide which case of recursions should choose, and we fail to continue computations. In other words, circular functions are instances of accumulative functions where they have some irregular structures, namely circularities.

From the viewpoints of AGs, the characteristics of circular programs are apparent. Circularities are dependencies from inherited attributes to synthesized attributes, where both attributes associated to the same non-terminal. To confirm this, we show an AG representation of `repmin` function as follows:

$$
\begin{array}{ll}
\mathit{repmin} \ \rightarrow \ \mathit{Tree} & \langle \mathit{repmin.result} = \mathit{Tree.r} \\
 & \quad \mathit{Tree.n} = \mathit{Tree.m} \rangle \\
\mathit{Tree} \ \rightarrow \ \texttt{Node}\ \mathit{Tree}\ \mathit{Tree} & \langle \mathit{Tree}_0.r = \texttt{Node}\ \mathit{Tree}_1.r\ \mathit{Tree}_2.r \\
 & \quad \mathit{Tree}_0.m = \texttt{min}\ \mathit{Tree}_1.m\ \mathit{Tree}_2.m \\
 & \quad \mathit{Tree}_1.n = \mathit{Tree}_0.n \\
 & \quad \mathit{Tree}_2.n = \mathit{Tree}_0.n \rangle \\
\mathit{Tree} \ \rightarrow \ \texttt{Leaf}\ \texttt{n} & \langle \mathit{Tree.r} = \texttt{Leaf}\ \mathit{Tree.n} \\
 & \quad \mathit{Tree.m} = \texttt{n} \rangle
\end{array}
$$

where n is an inherited attribute and r and m are synthesized attributes. The dependency from $\mathit{Tree.m}$ to $\mathit{Tree.n}$ in the top case corresponds to the circularity. We show its computation structure in Figure 2.4. This AG shows that `repmin` is actually not *circular* in the sense that no attribute dependency makes a cycle. It points out that `repmin` is evaluable by finite times paths of computation over a tree. Lazy evaluation enables to pack a multi-path computation into a closure and produces its result in a one-path computation over a tree.

Chapter 3

Fusion

In this chapter, we are going to explain fusion. Fusion is a program transformation that compresses some functions into one. Starting from explaining why fusion is important, we will do a brief introduction to the existing fusion methods. We will also discuss the manipulation power of the existing fusion methods, that is to say whether or not they can manipulate accumulative functions and circular functions.

3.1 Why Fusion Matters

Correctness, reusability and maintainability of programs have been important, because size of programs has been growing as computational power increases. To get such good characteristics we often adopt a module-based approach. We prepare reusable primitive functions, construct modules by assembling primitives, and organize large programs by combining modules. The module-based approach does not only produce reusable programs, but also contribute to their correctness and maintainability because it enables to divide a program into closed parts and clarify the structure of programs. But it decreases efficiency, for it often introduces overheads. To make things clear, think about the following function that computes variance of the input list.

```
variance x = average (map square (map (\a->a-average x) x))
    where average x = sum x / length x
          square x = x * x
```

This program is apparently correct because it is the direct implementation of definition of variance. Though the module-based approach lead the easiness of proving correctness of it, it is not efficient. It performs multiple traversals over data structures and unnecessary construction and destruction of intermediate data structures. The function `average` traverses over the input data structure two times, one for `sum` and another for `length`, and `average` called by `map` traversals the list many times. Every function calls of `map` makes an intermediate list, which is consumed by the next function. These overheads decline efficiency.

In general, there are much plenty of rooms to optimize a program that is constructed by module-based approach, because some results of functions are often unnecessary for continuing the rest of computation. For example, to find the minimum value of a list, we can implement it as follows:

```
listMin x = head (sort x)
```

where `sort` sorts the elements of a list. This is a modular and correct program, but not efficient. Though out intuition gives that we can solve this problem in $O(n)$ time complexity, where n denotes

the length of the input list, that of `listMin` is $O(n \log n)$ in strict evaluation because of the time complexity of `sort`. We can forget the tail of the list while sorting the list because the function `head` only needs the head of list. But to achieve this optimization we needs to do inter-procedure analysis, i.e., `sort` needs to know that the function waiting for its result is `head`.

Fusion (or *deforestation*, in some time) is a program transformation to fuse some functions into one. Fusion removes overheads of function calls such as multiple traversals and production or consumption of intermediate data structures, and leads possibilities of inter-procedure optimizations. For example, function `variance` above is transformed into the following implementation by existing fusion techniques.

```
variance x = let (rs, lr, av) = aux x 0 0 in rs / lr
    where aux [] s l = (0, l, s/l)
          aux (a:x) s l = let (rs, lr, av) = aux x (a+s) (1+l)
                          in ( square (a-av)+rs, lr, av)
```

Fusion removes a multiple traversals over the input list and intermediate data structures, and improves efficiency.

In summary, fusion is important to construct modular and yet efficient programs. Using fusion, we can enjoy not only reusability, maintainability, and correctness but also efficiency.

3.2 Fusion based on Folding-Unfolding Transformation

Folding-unfolding is a program transformation methodology introduced by Burstall and Darlington [BD77]. It consists of the following six rules:

***Unfolding*:** replace a function call by an appropriate instance of body of its definition.

***Folding*:** replace an appropriate instance of body of a function definition by its call.

***Definition*:** introduce a new function definition.

***Instantiation*:** introduce a substitution instance of an existing equation.

***Abstraction*:** abstract out a value by introducing new variables.

***Laws*:** apply some known laws.

Roughly speaking, folding-unfolding is a program manipulation methodology where we try to get efficient programs by continuously unfolding functions, evaluating subexpressions or applying known program transformation rules, and folding a subexpression into a function. Folding-unfolding is very powerful methodology, however, its effect depends on transformation strategy, namely when and how we apply which rules. To construct effective and terminate strategy is very hard.

Wadler [Wad88] proposed an effective and terminate fusion method based on folding-unfolding methodology. He calls his method as *deforestation*, where he is liken intermediate data structures to trees, so many people call fusion as deforestation. He considered the following subset of functional programs, called treeless terms, as the domain of his transformation.

Definition 3.2.1 (Treeless term)**.**
A term of first-order language is treeless *if all of its subterm is generated by the following grammar and all of its variables appear only once.*

$$
\begin{array}{rcll}
t & ::= & v & \textit{variable} \\
 & | & c\ t_1\ \cdots\ t_n & \textit{constructor} \\
 & | & f\ v_1\ \cdots\ v_n & \textit{function} \\
 & | & \textbf{case}\ v_0\ \textbf{of}\ p_1 : t_1 \mid \cdots \mid p_n : t_n & \textit{pattern-matching} \\
p & ::= & c\ v_1\ \cdots\ v_n & \textit{pattern}
\end{array}
$$

$$
\begin{array}{lll}
(1) & T[\![v]\!] & = v \\
(2) & T[\![c\ t_1\ \cdots\ t_n]\!] & = c\ (T[\![t_1]\!])\ \cdots\ (T[\![t_n]\!]) \\
(3) & T[\![f\ t_1\ \cdots\ t_n]\!] & = T[\![t[t_1/v_1,\ \cdots\ ,t_n/v_n]]\!] \\
 & & \quad \text{where } f \text{ is defined by } f\ v_1\ \cdots\ v_n = t \\
(4) & T[\![\textbf{case}\ v\ \textbf{of}\ p_1 \to t_1 \mid \cdots \mid p_n \to t_n]\!] & \\
 & & = \textbf{case}\ v\ \textbf{of}\ p_1 \to T[\![t_1]\!] \mid \cdots \mid p_n \to T[\![t_n]\!] \\
(5) & T[\![\textbf{case}\ c\ t_1\ \cdots\ t_n\ \textbf{of}\ p_1 \to t_1 \mid \cdots \mid p_n \to t_n]\!] & \\
 & & = T[\![t_i[t_1/v_1,\ \cdots\ ,t_n/v_n]]\!] \\
 & & \quad \text{where } p_i = c\ v_1\ \cdots\ v_n \\
(6) & T[\![\textbf{case}\ f\ t_1\ \cdots\ t_n\ \textbf{of}\ p_1 \to t_1 \mid \cdots \mid p_n \to t_n]\!] & \\
 & & = T[\![\textbf{case}\ t[t_1/v_1,\ \cdots\ ,t_n/v_n]\ \textbf{of}\ p_1 \to t_1 \mid \cdots \mid p_n \to t_n]\!] \\
 & & \quad \text{where } f \text{ is defined by } f\ v_1\ \cdots\ v_n = t \\
(7) & T[\![\textbf{case}\ (\textbf{case}\ v\ \textbf{of}\ p_1 \to t_1 \mid \cdots \mid p_n \to t_n)\ \textbf{of}\ p'_1 \to t'_1 \mid \cdots \mid p'_m \to t'_m]\!] & \\
 & & = T[\![\textbf{case}\ v\ \textbf{of} \\
 & & \qquad p_1 \to (\textbf{case}\ t_1\ \textbf{of}\ p'_1 \to t'_1 \mid \cdots \mid p'_m \to t'_m) \\
 & & \qquad\ \vdots \\
 & & \qquad p_n \to (\textbf{case}\ t_n\ \textbf{of}\ p'_1 \to t'_1 \mid \cdots \mid p'_m \to t'_m)]\!]
\end{array}
$$

where $T[\![t]\!]$ denotes the result of applying the transformation to a term t. We remember the right-hand side term before applying rule (3) or rule (6), and we do folding if we find the same term as remembered one while the transformation.

Figure 3.1. Transformation strategy of Wadler's deforestation

where t, p, v, and c respectively denote a term, a pattern, a constructor, and a function.

He proved that composition of two treeless terms becomes a treeless term by his transformation strategy shown in Figure 3.1. We show an example of transformation in Figure 3.2.

Characteristics of treeless form is that each argument of a function should be a variable. Therefore a treeless term contains no intermediate data or accumulative data. This characteristics leads easiness to fuse functions. Wadler's deforestation strategy is not only deterministic, but also terminate and eliminate all intermediate data structures without fail. But its drawback is expressiveness. We can scarcely write even a small toy example in terms of treeless terms. Of cause we can write neither accumulative programs nor circular programs, because we can express neither accumulation nor circularity in terms of treeless terms.

3.3 Fold Promotion

Fold promotion theorem is a fusion law for structural recursions. It has appeared in the literature using various notations [Bir89][MFP91][SF93]. We show it for structural recursions over lists as follows.

Theorem 3.3.1 (Fold promotion)**.**

$$
\frac{\begin{array}{l} \texttt{f e} = \texttt{e'} \\ \texttt{f (a} \oplus \texttt{y)} = \texttt{a} \otimes \texttt{(f y)} \end{array}}{\texttt{f} \cdot \texttt{(foldr (}\oplus\texttt{) e)} \Rightarrow \texttt{foldr (}\otimes\texttt{) e'}}
$$

□

```
x++y = case x of [] -> y
                 a:w -> a:(w++y)
appapp x y z = (x++y)++z
```

T⟦appapp x y z⟧ ⇒ T⟦(x++y)++z ⟧ (3)(∗)
⇒ T⟦**case** (x++y) **of** []->z
a:w->a:(w++z)⟧ (3)
⇒ T⟦**case** (x **of** []->y
a:w->a:(w++y)) **of** []->z
a:w-> a:(w++z)⟧ (6)
⇒ T⟦**case** x **of** []->(**case** y **of** []->z
a:w->a:(w++z))
a:w->(**case** a:(w++y) **of** []->z
a:w->a:(w++z))⟧ (7)
⇒ T⟦**case** x **of** []->(**case** y **of** []->z
a:w->a:(w++z))
a:w->a:((w++y)++z)⟧ (5)
⇒ T⟦**case** x **of** []->(**case** y **of** []->z
a:w->a:(w++z))
a:w->a:(appapp w y z)⟧ (folding by (∗))

Now that the right-hand side term is treeless, we complete the transformation.

Figure 3.2. Example of Wadler's deforestation

Fold promotion theorem is a simple but effective theorem. We can achieve chain reaction fusions whenever we write a kernel function in terms of `foldr`, and especially it is effective for deriving efficient algorithms in systematic way. To use Theorem 3.3.1 is usually easy because structural recursions are expressive as we have mentioned in Section 2.1.2, but its automation is a bit problematic because finding a proper $(\otimes)$ automatically is difficult and it requires higher-order pattern matching [HL78][dMS01][Yok06]. It is a drawback of fold promotion theorem comparing to shortcut deforestation, which we are going to introduce in Section 3.5.

Now we give an example of transformation based on fold promotion theorem. Think about a program `length (x++y)`, where `length` and `(++)` are defined in terms of `foldr` as follows.

```
length x = foldr (\a r->1+r) 0 x
x++y = foldr (:) y x
```

We can fuse the composition above using Theorem 3.3.1 as follows.

```
length (x++y)  ⇒  length (foldr (:) y x)
               ⇒    {- Fold promotion theorem:
                       length ((:) a r) ⇒ 1+(length r) -}
                  foldr (\a r-> 1+r) (length y) x
```

It certainly fuses the composition and eliminate the intermediate list passed between `length` and `(++)`.

From a viewpoint of folding-unfolding methodology, fold promotion theorem gives a sufficient condition to succeed in an effective folding. If we find a proper $(\otimes)$ satisfying an equation above, then we can do a successful folding by $(\otimes)$. Theorem 3.3.1 can be extended to general structural recursions [MFP91][SF93][HIT96] by giving a sufficient condition for each structural recursions to

achieve a successful folding. Here we show fold promotion theorem for structural recursions on trees for example. First we give an implementation of structural recursions on trees as follows.

```
foldTree g1 g2 (Leaf n) = g2 n
foldTree g1 g2 (Node l r) = g1 (foldTree g1 g2 l) (foldTree g1 g2 r)
```

The function `foldTree` expresses homomorphisms over trees as mentioned in Section 2.3.2. Now we give a fusion law of `foldTree` as a relative of Theorem 3.3.1.

Theorem 3.3.2 (Fold Promotion on trees)**.**

$$\frac{\begin{array}{l} \texttt{f (g2 n)} = \texttt{g2' n} \\ \texttt{f (g1 l r)} = \texttt{g1' (f l) (f r)} \end{array}}{\texttt{f} \cdot \texttt{foldTree g1 g2} \Rightarrow \texttt{foldTree g1' g2'}}$$

□

Applying fold promotion theorem to accumulative functions makes disappointing results. Two functions are fused into one, but intermediate data structures in accumulative arguments remain yet. For example, think about the following accumulative function `reverse`.

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

The auxiliary function of `reverse` is an instance of `foldr` as follows:

```
rev x = foldr (\a r h->r(a:h)) (\h->h) x
```

where `rev` is a higher-order `foldr` in the sense that it returns a function value. Now consider a fusion problem `length (reverse x)` as an example. Theorem 3.3.1 conducts calculations as follows.

```
length (reverse x) ⇒ length (rev x [])
                   ⇒ (length·) (foldr (\a r h->r(a:h)) (\h->h) x) []
                   ⇒   {- Fold promotion theorem:
                         (length·) ((\a r h->r(a:h)) a r) ⇒ (\h->(length·r)(a:h))
                         (length·) (\h->h) ⇒ length -}
                       foldr (\a r h-> r(a:h)) length x []
```

The result is as follows after unfolding `foldr`.

```
lengthreverse x = aux x []
    where aux [] h = length h
          aux (a:x) h = aux x (a:h)
```

This transformation does not improve its efficiency at all. The `length` in the base case of `aux` is still waiting for the whole reversed list. As this result indicates, fold promotion theorem works only for results and it is completely powerless for arguments.

Fold promotion theorem is also problematic to manipulate circular programs. It can do noting about accumulative arguments. It is critical because accumulative arguments are essential for circular programs.

3.4 Higher-Order Promotion

As we have seen in Section 3.3, fold promotion theorem cannot manipulate accumulative argument at all. To solve this problem, Meijer [Mei92] proposes higher-order promotion theorem, which is a fusion method for higher-order structural recursions. It can be polytypic as the same as Theorem 3.3.1, and we write down its list case.

Theorem 3.4.1 (Higher-order promotion)**.**

$$\frac{\begin{array}{l}\texttt{f} \cdot \texttt{e} = \texttt{e'} \cdot \texttt{g} \\ \texttt{f} \cdot (\texttt{a} \oplus \texttt{y}) = (\texttt{a} \otimes \texttt{r}) \cdot \texttt{g} \quad \text{if } \texttt{f} \cdot \texttt{y} = \texttt{r} \cdot \texttt{g}\end{array}}{\texttt{f} \cdot (\texttt{foldr}\ (\oplus)\ \texttt{e}\ \texttt{x}) \Rightarrow (\texttt{foldr}\ (\otimes)\ \texttt{e'}\ \texttt{x}) \cdot \texttt{g}}$$

□

This theorem effectively achieves fusion for accumulative functions whenever we succeed to satisfy this complicate condition. Recall that Theorem 3.3.1 cannot remove the intermediate list of `length (reverse x)`. Using Theorem 3.4.1, we can calculate as follows.

```
length (reverse x)  ⇒  length · (foldr (\a r h->r(a:h)) (\h->h) x) []
                    ⇒    {- Higher-order promotion theorem:
                            (length·y = r·length) →
                              length·((\a y h->y(a:h)) a y) ⇒ (\h->(length·y)(a:h))
                                                            ⇒ (\h->(r·length)(a:h))
                                                            ⇒ (\h->r(1+length h))
                                                            ⇒ (\h->r(1+h))·length)
                              length·(\h->h) ⇒ (\h->h)·length) -}
                         ((foldr (\a r h-> r(1+h)) (\h->h) x)·length) []
                    ⇒  foldr (\a r h-> r(1+h)) (\h->h) x 0
```

But it is still troublesome to manipulate a composition of accumulative functions. For example, to achieve fusion of `reverse (reverse x)`,

```
reverse (reverse x)
    ⇒  reverse · (foldr (\a r h->r(a:h)) (\h->h) x) []
    ⇒    {- Higher-order promotion theorem:
           (reverse·y = r·reverse) →
              reverse·((\a y h->y(a:h)) a y) ⇒ (\h->(reverse·y)(a:h))
                                             ⇒(\h->(r·reverse)(a:h))
                                             ⇒(\h->r(h++[a]))·reverse)
           reverse · (\h->h) ⇒ (\h->h) · reverse) -}
        ((foldr (\a r h-> r(h++[a])) (\h->h) x) · reverse) []
    ⇒  foldr (\a r h-> r(h++[a])) (\h->h) x []
```

we need to use a lemma `reverse (a:h) = reverse h ++ [a]`. Without this lemma we cannot calculate `(\h->(r·reverse)(a:h))` anymore, even though this lemma is not apparent from the program of `reverse`. Theorem 3.4.1 is also troublesome to make a function `g` in the rule above, because there are many choices of `g` and only appropriate choices make fusion successful. Conversely, if we choose a proper `g` and prepare proper lemmas we can achieve fusion successfully. It is like a two-edged sword. It is theoretical very powerful though, we should be careful in practical use of it.

Hu et al. [HIT99] tackled to make its calculation easy. They proposed to divide the process of fusion into two steps. These two steps are expressed in terms of two theorems, named *accumulation*

promotion I and *accumulation promotion II*. The former is essentially the same as Theorem 3.3.1 of higher-order case. We introduce the later one.

Theorem 3.4.2 (Accumulation promotion II)**.**

$$\frac{\begin{array}{rcl} \texttt{e} & = & \texttt{e'} \cdot \texttt{g} \\ \texttt{(a} \oplus \texttt{(r} \cdot \texttt{g))} & = & \texttt{(a} \otimes \texttt{r)} \cdot \texttt{g} \end{array}}{\texttt{(foldr (}\oplus\texttt{) e x)} \;\Rightarrow\; \texttt{(foldr (}\otimes\texttt{) e' x)} \cdot \texttt{g}}$$

□

Using Theorem 3.4.2, we can remove the intermediate data structure of `lengthreverese` that we have derived in Section 3.3 as follows.

```
lengthreverse x  ⇒  foldr (\a r h-> r(a:h)) length x []
                 ⇒    {- Accumulation promotion II:
                        (\a r h->(r·length)(a:h)) a r) ⇒ (\h->r(1+length h))
                                                       ⇒ (\h->r(1+h))·length)
                        length ⇒ (\h->h)·length) -}
                     ((foldr (\a r h-> r(1+h)) (\h->h) x)·length) []
                 ⇒  foldr (\a r h-> r(1+h)) (\h->h) x 0
```

On one hand, calculating 3.4.2 is easier than that of Theorem 3.4.1 because of fewer free parameters. Moreover, combining Theorem 3.4.2 with accumulation fusion I, namely Theorem 3.3.1, we can derive Theorem 3.4.1. On the other hand its manipulation power is inferior to Theorem 3.4.1 in general.

3.5 Shortcut Deforestation

Shortcut deforestation (or shortcut fusion) was proposed by Gill et al. [GLPJ93] for lists, and later extended to be polytypic by Takano and Meijer [TM95]. We introduce it for lists.

Theorem 3.5.1 (Shortcut deforestation)**.**
If a function `g` have the following polymorphic type:
`g` :: $\forall\beta.\ (A \to \beta \to \beta) \to \beta \to \beta$
then,

```
foldr k z (build g) =¿ g k z
```

where `build` is defined by `build g = g (:) []`. □

Recall that `foldr` corresponds to structural recursions on lists, namely homomorphisms on the algebraic structure of lists. Then `foldr k z` replaces all constructors produced by `build g`, i.e., it replaces `(:)` and `[]` with `k` and `z` respectively. Now assume that all constructors are abstracted out by lambda abstraction like a context, or data structures are expressed by Church encoding explicitly, such as for example `(1:(2:(3:[])))` becomes `\c n->(c 1(c 2(c 3 n)))`. Then just a simple substitution, which is equivalent to an application in lambda calculus, gives a computation of `foldr`. It is the main idea of shortcut deforestation. A function `g` is a context where constructors are abstracted out, while `build` expresses a intermediate list by explicitly giving constructors to the context `g`. A function `foldr k z` consumes the intermediate list produced by `build`. Then the computation of `foldr k z(build g)` is equivalent to substituting an operations `k` and `z` to the context `g`. The type annotation guarantees that constructors of `g` are abstracted out appropriately.

One of advantages of shortcut deforestation is simplicity of its rule. If we write programs in terms of `foldr`/`build` and guarantee that they satisfy the type annotation, then we can achieve fusion by cancelling corresponding `foldr` and `build` symbolically. Actually Gill [Gil96] implements Theorem 3.5.1

into GHC (Glasgow Haskell Compiler) [GHC] and confirms its effectiveness. Moreover, as similar with Theorem 3.3.1, shortcut deforestation has its extensions to other data structures [TM95]. But its drawback comes from a difficulty to derive proper `foldr/build` form programs automatically. Several works have done about automatic derivation of `foldr/build` form [LS95][HIT96][Chi99][YHT05], but we do not discuss their detail.

Now we show some transformation examples of shortcut deforestation. First we use the same example of Section 3.3, namely `length (x++y)`. We program `length` and (++) in terms of `foldr/build` form as follows:

```
length x = foldr (\a r->1+r) 0 x
x++y = build (\c n->foldr c (foldr c n y) x)
```

where `(\c n->foldr c (foldr c n y) x)` has the appropriate type. Then Theorem 3.5.1 enables us to achieve fusion as follows.

```
length (x++y)  ⇒  foldr (\a r->1+r) 0 (build (\c n->foldr c (foldr c n y) x))
               ⇒    {- Shortcut deforestation -}
                   (\c n->foldr c (foldr c n y) x))(\a r->1+r) 0
               ⇒  foldr (\a r->1+r) (foldr (\a r->1+r) 0 y) x)
```

Shortcut deforestation also works if the producer function of intermediate structures is either accumulate or circular. We check it with an accumulative function `reverse` and a circular function `cycle` as follows.

```
reverse x = build (\c n->foldr (\a r h->r(c a h)) (\h->h) x n)

length (reverse x)
   ⇒  foldr (\a r->1+r) 0 (build (\c n->foldr (\a r h->r(c a h)) (\h->h) x n))
   ⇒    {- Shortcut deforestation -}
       (\c n -> foldr (\a r h->r(c a h)) (\h->h) x n)(\a r->1+r) 0
   ⇒  foldr (\a r h->r(1+h)) (\h->h) x 0

cycle x = build (\c n->let r = foldr (\a r h->c a (r h)) (\h->h) x r in r)

length (cycle x)
   ⇒  foldr (\a r->1+r) 0
            (build (\c n->let r = foldr (\a r h->c a (r h)) (\h->h) x r in r))
   ⇒    {- Shortcut deforestation -}
       (\c n->let r = foldr (\a r h->c a (r h)) (\h->h) x r in r)(\a r->1+r) 0
   ⇒  let r = foldr (\a r h->1+(r h)) (\h->h) x r in r)
```

But if both the producer and consumer functions of intermediate structures are accumulative, then shortcut deforestation results in a troublesome result. Consider a program `reverse(reverse x)`, which seems to be an identity function after fusion.

```
reverse (reverse x)
   ⇒  foldr (\a r h->r(a:h)) (\h->h)
            (build (\c n->foldr (\a r k->r(c a k)) (\k->k) x n)) []
   ⇒    {- Shortcut deforestation -}
       (\c n->foldr (\a r k->r(c a k)) (\k->k) x n)(\a r h->r(a:h)) (\h->h)[]
   ⇒  foldr (\a r k->r(\h->k(a:h))) (\k->k) x (\h->h) []
```

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

```
revrev x = aux x (\h->h) []
  where aux [] k = k
        aux (a:x) k = aux x (\h->k(a:h))
```

This result differs from an expectation. Its auxiliary function makes a complex closure in spite of an intermediate data structure, and it hiders to improve its efficiency. It is a drawback of shortcut deforestation. Shortcut deforestation cannot completely manipulate accumulative functions. We need another program transformation to remove higher-order accumulation and improve efficiency. We are going to show that a method imported from the world of attribute grammars solves this problem in Section 3.8.

3.6 Tupling

Tupling is a kind of fusion in a wide sense, which eliminate multiple traversals of a data structure. It has appeared in the literature using various notations, for example [Bir84b][Fok89][Chi93][HITT97] are found in the community of functional programming. Here we pick up mutu tupling theorem in [Fok89], and write down for a very simple case.

Theorem 3.6.1 (Simple mutu tupling)**.**

$$\frac{\texttt{f1} = \texttt{foldr k1 z1} \qquad \texttt{f2} = \texttt{foldr k2 z2}}{\texttt{(f1 x,f2 x)} \Rightarrow \texttt{foldr (\a (r1,r2)->(k1 a r1, k2 a r2)) (z1,z2) x}}$$

□

This theorem argues that we can compute results of `f1` and `f2` in the same time because they have the same recursion structure. We can make it polytypic as similar with Theorem 3.3.1 because it only cares about the recursion schemes.

Consider the `average` function in Section 3.1 for example. We programed it as follows:

```
average x = sum x / length x
```

where `sum` and `length` traverses over the same list `x`. Recall that `sum` and `length` are expressed in terms of `foldr` as follows.

```
sum x = foldr (+) 0 x
length x = foldr (\a r->1+r) 0 x
```

We apply Theorem 3.6.1 and eliminate multiple traversals as follows.

```
average x  ⇒ uncurry (/) (foldr (+) 0 x, foldr (\a r->1+r) 0 x)
           ⇒   {- Simple mutu tupling theorem -}
               uncurry (/) (foldr (\a (r1,r2)->(a+r1,1+r2)) (0,0) x)
           ⇒ let (r1,r2) = foldr (\a (r1,r2)->(a+r1,1+r2)) (0,0) x in r1/r2
```

Then we get the following program after unfolding `foldr`.

```
average x = let (r1,r2) = ave x in r1 / r2
  where ave [] = (0,0)
        ave (a:x) = let (r1,r2) = ave x in (a+r1, 1+r2)
```

The multiple traversals over the input list is certainly disappeared.

Tupling is applicable for either accumulative functions or circular programs, because it only pays attention to the structure of recursions, that is to say how to manipulate the recursion argument. Besides, Bird [Bir84b] introduced circular programs as a result of tupling. We will show that tupling certainly derives circular programs. We show a derivation of circular `repmin` program that we have introduced in Section 2.3.2. First recall structural recursions on trees are expressed in terms of the following `foldTree` function, as we have seen in Section 3.3:

```
foldTree g1 g2 (Leaf n) = g2 n
foldTree g1 g2 (Node l r) = g1 (foldTree g1 g2 l) (foldTree g1 g2 r)
```

And recall that we can implement a non-circular version of `repmin`, named `transform`, by two functions `tmin` and `replace`. They are in fact structural recursions on trees. Then `transform` is as follows.

```
transform t = replace t (tmin t)
  where replace t = foldTree (\l r h-> Node (l h) (r h)) (\n h->Leaf h) t
        tmin t = foldTree min id  t
```

We can see that both `replace` and `tmin` have the same recursion structure. We use simple mutu tupling theorem for structural recursions on trees to eliminate multiple traversals as follows.

```
transform x
   ⇒  apply (foldTree (\l r h-> Node (l h) (r h)) (\n h->Leaf h) t, foldTree min id  t)
         where apply (a,b) = a b
   ⇒    {- Simple mutu tupling theorem -}
       apply (foldTree (\(l1,l2) (r1,r2)->(\h->Node (l1 h) (r1 h), min l2 r2))
                       (\n->(\h->Leaf h,n)) t)
         where apply (a,b) = a b
```

We get the following program after unfolding `apply` and `foldTree`.

```
transform t = let (r,m) = aux t in r m
  where aux (Leaf n) = (\h->Leaf h, n)
        aux (Node l r) = let (l1,l2) = aux l
                             (r1,r2) = aux r
                         in (\h->Node (l1 h) (r1 h), min l2 r2)
```

Finally, We remove higher-order results using an extension of η-expansion. Define a new auxiliary function to give an extra argument to the first element of the result of `aux` as follows:

```
aux' x h = let (r, m) = aux x in (r h, m)
```

then we get the following result by replacing `aux` with `aux'`.

```
transform t = let (r,m) = aux' t m in r
  where aux' (Leaf n) h = (Leaf h, n)
        aux' (Node l r) h = let (l1,l2) = aux' l h
                                (r1,r2) = aux' r h
                            in (Node l1 r1, min l2 r2)
```

It is the circular function `repmin` that introduced by Bird.

In the framework of AGs, simple mutu tupling comes out apparently. It is just a merge of semantic equations of two AGs. We confirm it throughout deriving the AG representation of circular `repmin`,

and this derivation was introduced by Johnsson [Joh87]. First we prepare AG representations of `tmin` and `replace` as follows:

$$
\begin{array}{ll}
\mathit{tmin} \ \to \ \mathit{Tree} & \langle \mathit{tmin.result} = \mathit{Tree.m} \rangle \\
\mathit{Tree} \ \to \ \texttt{Node}\ \mathit{Tree}\ \mathit{Tree} & \langle \mathit{Tree}_0.m = \texttt{min}\ \mathit{Tree}_1.m\ \mathit{Tree}_2.m \rangle \\
\mathit{Tree} \ \to \ \texttt{Leaf}\ \texttt{n} & \langle \mathit{Tree.m} = \texttt{n} \rangle \\
\mathit{replace} \ \to \ \mathit{Tree} & \langle \mathit{replace.result} = \mathit{Tree.r} \\
 & \quad \mathit{Tree.n} = \mathit{replace.m} \text{ (the second argument of } \texttt{replace}) \rangle \\
\mathit{Tree} \ \to \ \texttt{Node}\ \mathit{Tree}\ \mathit{Tree} & \langle \mathit{Tree}_0.r = \texttt{Node}\ \mathit{Tree}_1.r\ \mathit{Tree}_2.r \\
 & \quad \mathit{Tree}_1.n = \mathit{Tree}_0.n \\
 & \quad \mathit{Tree}_2.n = \mathit{Tree}_0.n \rangle \\
\mathit{Tree} \ \to \ \texttt{Leaf}\ \texttt{n} & \langle \mathit{Tree.r} = \texttt{Leaf}\ \mathit{Tree}_{\cdot}n \rangle
\end{array}
$$

where n is an inherited attribute and m and r are synthesized attributes. Now that the underlying CFG of `tmin` and `replace` is the same, we can compute both `tmin` and `replace` in one AG, which is constructed by merging these two AGs into one as follows.

$$
\begin{array}{ll}
\mathit{repmin} \ \to \ \mathit{Tree} & \langle \mathit{repmin.tmin} = \mathit{Tree.m} \\
 & \quad \mathit{repmin.replace} = \mathit{Tree.r} \\
 & \quad \mathit{Tree.n} = \mathit{replace.m} \text{ (the second argument of } \texttt{replace}) \rangle \\
\mathit{Tree} \ \to \ \texttt{Node}\ \mathit{Tree}\ \mathit{Tree} & \langle \mathit{Tree}_0.m = \texttt{min}\ \mathit{Tree}_1.m\ \mathit{Tree}_2.m \\
 & \quad \mathit{Tree}_0.r = \texttt{Node}\ \mathit{Tree}_1.r\ \mathit{Tree}_2.r \\
 & \quad \mathit{Tree}_1.n = \mathit{Tree}_0.n \\
 & \quad \mathit{Tree}_2.n = \mathit{Tree}_0.n \rangle \\
\mathit{Tree} \ \to \ \texttt{Leaf}\ \texttt{n} & \langle \mathit{Tree.m} = \texttt{n} \\
 & \quad \mathit{Tree.r} = \texttt{Leaf}\ \mathit{Tree}_{\cdot}n \rangle
\end{array}
$$

This step corresponds to an application of Theorem 3.6.1. Finally we reform so that it actually compute `repmin`. The second argument of `replace`, denoted by *replace.m* should be the same as the result of `tmin`, denoted by *repmin.tmin*, so these attributes should be connected by a semantic equation. And the result of `repmin` is the result of `replace`, namely *repmin.replace*. Then we get the following AG:

$$
\begin{array}{ll}
\mathit{repmin} \ \to \ \mathit{Tree} & \langle \mathit{repmin.result} = \mathit{Tree.r} \\
 & \quad \mathit{Tree.n} = \mathit{Tree.m} \rangle \\
\mathit{Tree} \ \to \ \texttt{Node}\ \mathit{Tree}\ \mathit{Tree} & \langle \mathit{Tree}_0.m = \texttt{min}\ \mathit{Tree}_1.m\ \mathit{Tree}_2.m \\
 & \quad \mathit{Tree}_0.r = \texttt{Node}\ \mathit{Tree}_1.r\ \mathit{Tree}_2.r \\
 & \quad \mathit{Tree}_1.n = \mathit{Tree}_0.n \\
 & \quad \mathit{Tree}_2.n = \mathit{Tree}_0.n \rangle \\
\mathit{Tree} \ \to \ \texttt{Leaf}\ \texttt{n} & \langle \mathit{Tree.m} = \texttt{n} \\
 & \quad \mathit{Tree.r} = \texttt{Leaf}\ \mathit{Tree}_{\cdot}n \rangle
\end{array}
$$

It is certainly the AG representation of the circular function `repmin`.

3.7 Descriptional Composition

Think about compiler construction. To construct a compiler, we usually program a multi-pass manipulation or analysis over an intermediate representation of source programs. It is quite natural to represent such a computation by compositions of AGs, because AGs are suitable to express computations over a structure such as an abstract syntax tree. Such a way of compiler construction is modular

but declines the efficiency of the resulting compiler, for the resulting compiler need to produce and consume many intermediate representations. We need a fusion method for AGs to remove intermediate representations and improve efficiency. Descriptional composition, introduced by Ganzinger and Giegerich [GG84], is a fusion method for AGs. Descriptional composition fuses a composition of two AGs into one and remove the intermediate data structures.

Theorem 3.7.1 (Descriptional composition)**.**
$F_1 = (G_1, A_1, D_1)$ and $F_2 = (G_2, A_2, D_2)$ are two AGs, satisfying the following conditions: (i) Exactly one synthesized attribute a_r, which is the result of F_1, is associated to G_1. The type of a_r is the same as the type of start symbol of G_2. (ii) Each attributes that will be a subtree of a_r appears at most once in the right-hand side of semantic equations of D_1. Then descriptional composition of F_1 with F_2, which corresponds to the computation of F_2 following F_1, is the attribute grammar $F_2 \circ F_1 = (G_1, A, D)$, defined by the following rules.

1. A contains all attributes of A_1 and D contains all types in D_1 and D_2.
2. Attributes in A_1 that will be a subtree of a_r are attributed by A_2, therefore a part of attributes of A_1 are multiplied according to the number of A_2. For example, an attribute $a \in A_1$ being subtree of a_r, for each $b \in A_2$, A contains all attributes denoted by $a.b$.
3. Multiplied attributes that come from synthesized attributes over synthesized attributes are synthesized attributes, synthesized over inherited are inherited, inherited over synthesized are inherited, and inherited over inherited are synthesized.
4. For all $b \in A_2$ and for all semantic equations $X.a_i = Y.a_j$ in D_1 where attributes a_i and a_j will be a subtree of a_r, D contains a semantic equation $X.a_i.b = Y.a_j.b$ if b is associated with a grammar symbol of G_2 that has the same type as $X.a_i$ and $Y.a_j$.
5. For all $b \in A_2$ and for all semantic equations $X_{i_0}.a_{j_0} = Y\ X_{i_1}.a_{j_1}\ X_{i_2}.a_{j_2} \cdots X_{i_k}.a_{j_k}$ in D_1 where Y is a grammar symbol of G_2, D contains a semantic equation for $X_{i_0}.a_{j_0}.b$, if D_2 has a semantic equation for $Z_{i_0}.b$ associated with a production rule $Z_{i_0} \to Y\ Z_{i_1}\ Z_{i_2} \cdots Z_{i_k}$. The semantic equation for $X_{i_0}.a_{j_0}.b$ is obtained from the semantic equations for $Z_{i_0}.b$, by renaming $Z_{i_0}, Z_{i_1}, \ldots, Z_{i_k}$ as $X_{i_0}.a_{j_0}, X_{i_1}.a_{j_1}, \ldots, X_{i_k}.a_{j_k}$.
6. D contains all semantic equations in D_1 if it is not matched with the rule 4 and 5 above.

□

To see how descriptional composition work, we try to fuse `reverse·reverse`. Recall that the AG representation of `reverse` is as follows:

$$
\begin{array}{lll}
\mathit{reverse} \ \to\ \mathit{List} & & \langle \mathit{reverse.result} = \mathit{List.r} \\
 & & \quad \mathit{List.h} = \texttt{[]} \rangle \\
\mathit{List} \ \to\ \texttt{a:}\mathit{List} & & \langle \mathit{List}_0.r = \mathit{List}_1.r \\
 & & \quad \mathit{List}_1.h = \texttt{a:}\mathit{List}_0.h \rangle \\
\mathit{List} \ \to\ \texttt{[]} & & \langle \mathit{List.r} = \mathit{List}\_h \rangle
\end{array}
$$

where we have two attributes h and r, and the type of the result is *List*. Note that the start symbol of `reverse` is *reverse* and different from *List*, although descriptional composition requires that the producer AG produces the intermediate data structure that is the same as the underlying CFG of the consumer AG. This inconsistency is not essential, and it comes from the gap of AGs with functional programming. We assume that there is a start symbol *reverse* in the root of *List*, and try to fuse `reverse·reverse`.

First, we derive the top case of the result of descriptional composition. We regard the semantic equation $\mathit{reverse.result} = \mathit{List.r}$ as $\mathit{reverse.result} = \mathit{reverse}\ \mathit{List.r}$ because of the reason discussed above, and use the rule 5. Then we have two semantic equations as follows.

$reverse.result.result = List.r.r$
$List.r.h = \texttt{[]}$

We also use the rule 5 for $List.h = \texttt{[]}$, and get the following semantic equation.

$List.h.r = List.h.h$

It is worth mentioning that $List.h.h$ is a synthesized attribute of the left-hand side of production rule then it should appear in the right-hand side of equations, and $List.h.r$ is an inherited attribute of the left-hand side of production rule then it should appear in the left-hand side of equations. Now we get the top case of `reverse·reverse`.

$$\begin{array}{ll} reverse \ \rightarrow \ List & \langle reverse.result.result = List.r.r \\ & \ \ List.r.h = \texttt{[]} \\ & \ \ List.h.r = List.h.h\rangle \end{array}$$

Next, we will make the step case. For $List_0.r = List_1.r$, we use the rule 4.

$List_0.r.r = List_1.r.r$
$List_1.r.h = List_0.r.h$

As before, note whether the attribute is synthesized or inherited and swaps the left-hand side and right-hand side of equations if it is necessary. For $List_1.h = \texttt{a:}List_0.h$, we use the rule 5.

$List_1.h.r = List_0.h.r$
$List_0.h.h = \texttt{a:}List_1.h.h$

We similarly do for the base case, and finally we get the following AG.

$$\begin{array}{ll} reverse \ \rightarrow \ List & \langle reverse.result.result = List.r.r \\ & \ \ List.r.h = \texttt{[]} \\ & \ \ List.h.r = List.h.h\rangle \\ List \ \rightarrow \ \texttt{a:}List & \langle List_0.r.r = List_1.r.r \\ & \ \ List_1.r.h = List_0.r.h \\ & \ \ List_1.h.r = List_0.h.r \\ & \ \ List_0.h.h = \texttt{a:}List_1.h.h\rangle \\ List \ \rightarrow \ \texttt{[]} & \langle List.r.r = List.h.r \\ & \ \ List.h.h = List.r.h\rangle \end{array}$$

This AG has many unnecessary carrying of value and apparently different from the computation of the identity function, but it actually describes a computation of `reverse·reverse`. Note that in the top case we have a circularity expressed by a semantic equation $List.h.r = List.h.h$, where an inherited attribute $h.r$ depends on the synthesized attribute $h.h$ of the same grammar symbol. It naturally comes from descriptional composition.

Descriptional composition is effective in the sense that it works even if both of the composed functions are accumulative. As we have explained, other fusion methods do not have this property. Descriptional composition is translated to agree with functional programming by many researchers [Küh98][Küh99][CDPR99][Voi04].

Descriptional composition is also applicable even if the AGs have circularities, namely dependencies from inherited attributes to synthesized attributes of the same grammar symbol. Because descriptional composition treats synthesized attributes and inherited attributes symmetrically, we have no reason to be troubled with the dependency from synthesized attributes to inherited attributes, namely

circularity. It is quite good though, we cannot apply descriptional composition to the following `cycle` function:

$$\begin{array}{lll} cycle \ \rightarrow \ List & & \langle cycle.result = List.r \\ & & \quad List.h = List.r \rangle \\ List \ \rightarrow \ \texttt{a:}List & & \langle List_1.h = List_0.h \\ & & \quad List_0.r = \texttt{a:}List_1.r \rangle \\ List \ \rightarrow \ \texttt{[]} & & \langle List.r = List.h \rangle \end{array}$$

because $List.r$ is used two times in the right-hand side of the semantic equations in the top case and violate the precondition of descriptional composition.

3.8 Shortcut Deforestation based on Descriptional Composition

Descriptional composition is similar with shortcut deforestation. Both methods consist of the following two steps. First we analyze which constructor will be consumed by the next structural recursion, and after that we substitute constructors by the operations according to the consumer structural recursion. But results are different. Shortcut deforestation to a composition of accumulative functions produce involved closures, though descriptional composition gives an AG whose types of attributes are primitive values. This observation implicates that we can remove higher-order closures of the result of shortcut deforestation by using the idea of descriptional composition. This implication is formalized by Nishimura [Nis03] [Nis04].

Here we show the result of shortcut deforestation of `reverse·reverse` again.

```
revrev x = aux x (\h->h) []
  where aux [] k = k
        aux (a:x) k = aux x (\h->k(a:h))
```

In auxiliary function `aux` we have a higher-order accumulative arguments and higher-order results. Recall the AG representation of `reverse·reverse`. The accumulative argument corresponds to the inherited attribute $List.h$, and the result corresponds to the synthesized attribute $List.r$. $List.h$ and $List.r$ are also attributed by attributes r and h, which respectively correspond to the arguments and the results of the closure. The AG representation of `reverse·reverse` indicates that what we should do to achieve higher-order removal is to give new arguments and results to the recursive function `aux`, where extra arguments and results correspond to the multiplied attributes such as $h.r$, $h.h$, etc. First think about the closure in the position of arguments (=inherited attribute). It takes an argument (=inherited attribute), then we need to introduce an extra result (=synthesized attributes), which corresponds to the attribute $h.h$, because inherited attributes over inherited attributes makes synthesised attributes. And we also need to introduce an extra argument corresponding to the attribute $h.r$. As for results it is similar. We need to introduce an extra argument corresponding to inherited attribute $r.h$ and an extra result corresponding to synthesized attributes $r.r$. Now we summarize this procedure as follows.

Procedure 3.8.1 (Nishimura's higher-order removal)**.**
Nishimura's higher-order removal is organized by the following six steps:

1. *For each accumulative argument that is made up of a closure, make a new result that corresponds to its argument.*
2. *For each accumulative argument that is made up of a closure, make a new argument that corresponds to its result.*
3. *Remove the all accumulative made up of closures.*

4. *For each result that is made up of a closure, make a new argument that corresponds to its argument.*
5. *For each result that is made up of a closure, make a new result that corresponds to its result.*
6. *Remove the all results made up of a closure.*

Applying Procedure 3.8.1 to the `revrev` above, we get the following program.

```
revrev2 x = let (rr,hh) = aux x ([],hh) in rr
  where aux [] (rh,hr) = (hr,rh)
        aux (a:x) (rh,hr) = let (rr,hh) = aux x (rh,hr)
                            in (rr,a:hh)
```

This is a first-order circular program for `reverse·reverse` and certainly corresponds to the result of descriptional composition of `reverse·reverse`.

The effectiveness of Nishimura's method is basically the same as that of descriptional composition, because both results are the same. If composed functions satisfy the precondition of descriptional composition it successfully produces first-order programs, even if the functions are either accumulative or circular.

Chapter 4

IO Swapping

IO swapping is a new transformation to change the view of recursive functions through literally swapping their inputs (arguments and call-time computations) and outputs (results and return-time computations). The rule of IO swapping is so general that almost every linear recursive function is in its domain, and it is also extensible toward non-linear recursive functions. Moreover, it has some good properties that is useful for program manipulations.

4.1 IO Swapping

Before going into the general framework, we illustrate the basic idea of the proposed technique using a typical function `foldl`. After that, we give a general rule that can manipulate almost every linear recursive function of our interest.

Corollary 4.1.1 (IO swapping for `foldl`)**.**
The functions `foldl` and `foldl2` defined below are equivalent.

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

foldl2 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)
```

Proof. It is direct consequence of Corollary 4.4.1, which we are going to introduce in Section 4.4. Applying Corollary 4.4.1 to `foldl` and removing unnecessary variables derives `foldl2`. □

It is worth noting how the result is computed using the function argument `f`. While `f` is applied to the accumulative argument in the function `foldl`, it comes to the surface to compute the result of `foldl2`. This is because IO swapping is a rule that swaps the call-time computations and the return-time computations of the original recursive function.

Figure 4.1 illustrates the computation processes of `foldl` and `foldl2`. It shows that turning over the figure of `foldl` looks almost the same figure as `foldl2`. This is the point of IO swapping. To understand what turning over the figure means, recall the role of arguments and results of recursive functions. The role of arguments in a recursive function is to compute and pass values from shallower parts of the recursion to deeper parts; the role of the results is the opposite, namely passing values from the deeper to the shallower. This implies that, if we make this tower of the recursion upside

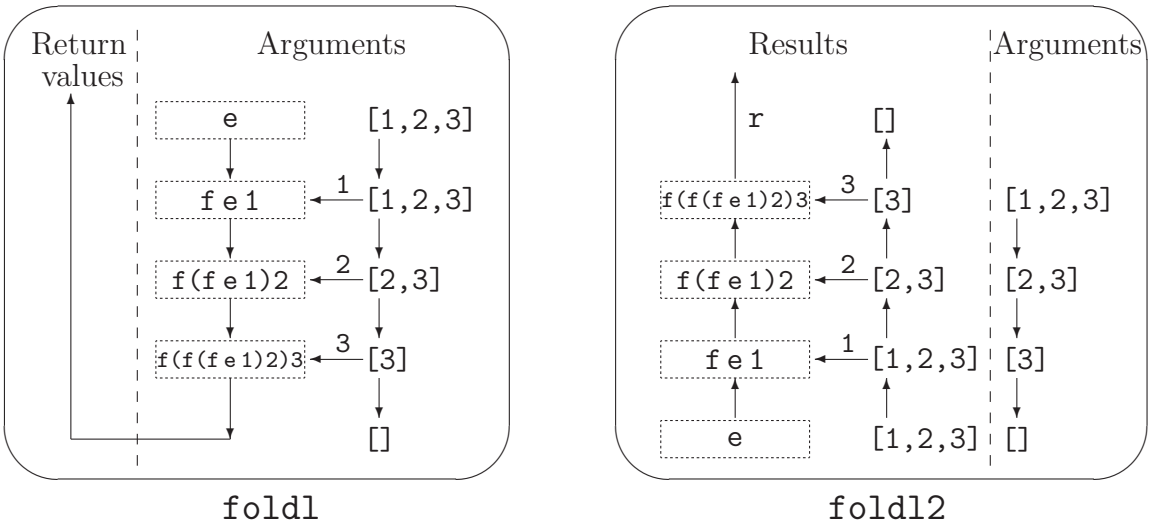

Figure 4.1. The models of computation processes of `foldl` and `foldl2`

down, we need to change the position of computation with arguments and results for keeping the whole results of computation. This is what IO swapping does. In Figure 4.1, the shallower and deeper parts correspond to the upper and lower part, respectively. The computation by `f` proceeds from the upper (shallower) part to the lower (deeper) part in `foldl` and from the lower (deeper) part to the upper (shallower) part in `foldl2`. These facts are reflected to the swapping of call-time computations and return-time computations.

The viewpoint of AGs helps us to understand it. We have a tree from the underlying CFG and synthesized and inherited attributes. Recall that synthesized attributes traversal over the tree from bottom to top, and inherited attributes traversal from top to bottom. It implies that if we want to swap synthesized attributes and inherited attributes, turning over the tree is proper. This observation corresponds completely to the functional-programming-based explanation above. Now the problem is how to turn over the tree. Here usual AGs get a stick. We have no way to manipulate the tree namely the underlying CFG. Now we lift up the view to relational AGs, so that we can manipulate the tree. The relational-AG-based representation of `foldl` is as follows:

$$
\begin{array}{lll}
\mathit{foldl} \ \rightarrow \ \mathit{foldl}' & \langle \mathit{foldl}.\mathit{result} = \mathit{foldl}'.r \\
 & \quad \mathit{foldl}'.x = \texttt{x} \\
 & \quad \mathit{foldl}'.h = \texttt{e} \rangle \\
\mathit{foldl}' \ \rightarrow \ \mathit{foldl}' & \langle {\mathit{foldl}'}_0.r = {\mathit{foldl}'}_1.r \\
 & \quad \texttt{a:x'} = {\mathit{foldl}'}_0.x \\
 & \quad {\mathit{foldl}'}_1.x = \texttt{x'} \\
 & \quad {\mathit{foldl}'}_1.h = \texttt{f}\ {\mathit{foldl}'}_0.h\ \texttt{a} \rangle \\
\mathit{foldl}' \ \rightarrow \ \epsilon & \langle \texttt{[]} = \mathit{foldl}'.x \\
 & \quad \mathit{foldl}'.r = \mathit{foldl}'.h \rangle
\end{array}
$$

where `f`, `e`, and `x` denote the initial arguments of `foldl`. Here production rules have little meaning because their productions completely depend on the semantic equations. The important thing for describing the computation of `foldl` is the sequence of logical relations that are expressed in terms of the semantic equations. In other words, any tree is proper if it can express the sequence of logical relations appropriately. This observation leads that the following relational AG is a proper one to

express the computation of `foldl`:

$$
\begin{array}{lll}
\mathit{foldl}' & \rightarrow & \mathit{foldl} \quad \langle \mathit{foldl.result} = \mathit{foldl}'.r \\
& & \quad\ \mathit{foldl}'.x = \texttt{x} \\
& & \quad\ \mathit{foldl}'.h = \texttt{e} \rangle \\
\mathit{foldl}' & \rightarrow & \mathit{foldl}' \quad \langle \mathit{foldl}'_1.r = \mathit{foldl}'_0.r \\
& & \quad\ \texttt{a:x'} = \mathit{foldl}'_1.x \\
& & \quad\ \mathit{foldl}'_0.x = \texttt{x'} \\
& & \quad\ \mathit{foldl}'_0.h = \texttt{f}\ \mathit{foldl}'_1.h\ \texttt{a} \rangle \\
\epsilon & \rightarrow & \mathit{foldl}' \quad \langle \texttt{[]} = \mathit{foldl}'.x \\
& & \quad\ \mathit{foldl}'.r = \mathit{foldl}'.h \rangle
\end{array}
$$

where we flip left-hand sides and right-hand sides of the production rules, which corresponds to turning over the whole tree. In this AG ϵ does not denote an empty sequence of grammar symbols anymore, but stands for the start symbol. Synthesized attributes and inherited attributes are also flipped.

Next we will translate it to a recursive function of functional programming. Here we should note two things: One is about how to taking out the result of the whole recursion. A recursive function should serve its result of the whole recursion from the top of the recursion, while in the relational AG above *foldl.result* appears at the leaf of the tree. This problem is easily solved by taking out the value of the attribute r from the top case as the result of the whole recursion, because the attribute r brings the same value throughout the tree and that is exactly the result of the whole recursion. Another problem is about the structure of the recursion. The relational AG above is not a structural recursion anymore. Furthermore it has no inherited attribute that expresses the recursion structure, while the synthesized attribute x expresses the structure of the tree. Recursive functions in functional programming must determine their recursion structure by their arguments for deterministic computation. Therefore we need to introduce an extra recursion argument. We use the input list `x` and construct recursion structure in the same manner with `foldl`, because the recursion structure of the relational AG above is the same with `foldl`. Finally we get the definition of `foldl2` in Corollary 4.1.1

The idea of Corollary 4.1.1 can be generalized so that it can be applied to almost every linear recursive function, including circular programs.

Theorem 4.1.2 (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')
```

□

We are going to give its proof in Section 4.2.

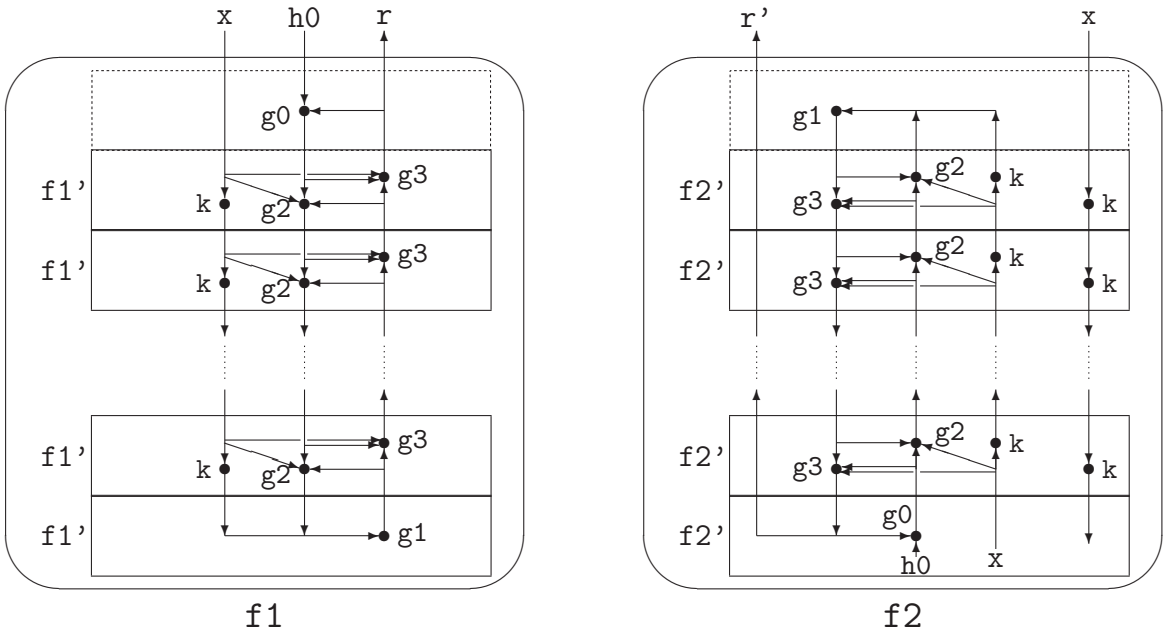

Figure 4.2. The computation processes of `f1` and `f2` in Theorem 4.1.2

Theorem 4.1.2 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.

The idea of Theorem 4.1.2 is the same as Corollary 4.1.1—the function `f2` uses its first argument only for constructing the recursion structure, then performs the same computation of `f1` in the IO-swapped manner, and finally returns the result of the whole recursion from the bottom of the recursion as the second element of the result, denoted by `r'`. Figure 4.2 shows the computation process of `f1` and `f2`. We can easily see that turning over the figure of `f1` gives almost the same figure of `f2`.

4.2 The Proof of IO Swapping

In this section we give a proof of Theorem 4.1.2. To prove it, we assume that all equations have a unique solution and we can compute it with a deterministic and finite computation. This is because we need to shift to the relational framework to prove Theorem 4.1.2. We need to assume that semantic equations in the IO-swapped relational AG are resolved appropriately. In functional programming uniqueness of the solution is no problem because a function has only one solution, but computability is fully dependent on operational semantics. For example, if we treat a strict functional language, we cannot use circular programs and Theorem 4.1.2 is apparently incorrect. That is to say, on one hand the power of IO swapping is restricted by the operational semantics so that its range and domain should contain only the functions that satisfy the assumption above. On the other hand, if the assumption is satisfied, IO swapping is correct in any operational semantics, such as lazy evaluation, many times sweep over data structures, back tracking, and so on.

Before starting the proof, we prepare some definitions and lemmas that we are going to use to prove the theorem.

Definition 4.2.1 (Computation of each recursion)**.**
Inputs of a recursion *are both its arguments and its next recursion's results.* Outputs of a recursion *are both its results and its next recursion's arguments.* Computation of a recursion *is to compute its outputs from its inputs. We call that computation of a recursion A and a recursion B is the same if the inputs and the outputs of A are the same as the inputs and the outputs of B respectively.*

The definition of inputs and outputs of a recursion become clearer by recalling the framework of AGs (not relational AGs but usual AGs). Outputs of a recursion are attributes that are in the left-hand side of semantic equations and computed by them. Inputs of a recursion are attributes that are in the right-hand side of semantic equations and passed from other part of the tree.

Next we give some lemmas.

Lemma 4.2.2.
The second result of `f2'` (denoted by `r'`) is constant during the whole recursions.

Proof. It is obvious from the definition of `f2`. □

Lemma 4.2.3.
The depth of recursions of `f1` and `f2` are the same if the input arguments of these functions are the same.

Proof. It is obvious from the definition of `f1` and `f2`. □

The next lemma is the point of IO swapping. Here "0-th recursive call", which corresponds to the top case in the world of AGs, means the outside of the recursion of the auxiliary function, "first recursive call" means the first call of the auxiliary function, and so on. n ($n \geq 1$) denotes the maximum depth of recursions, where notice that `f1` and `f2` have same maximum depth of the recursion, from Lemma 4.2.3.

Lemma 4.2.4.
For all k such that $0 \leq k \leq n$, the k-th recursion of `f1` does the same computation as the $(n-k)$-th recursion of `f2`, except for the first element of argument and the second element of result of `f2`, denoted by `y` and `r'` respectively.

Proof. We introduce some notations for arguments and results of recursions performed by `f1` and `f1'`. We write the result of their 0-th recursion as $\mathtt{r}_0$, the recursion argument of their 0-th recursion as $\mathtt{x}_0$, the accumulative argument of their 0-th recursion as $\mathtt{h}_0$, the result of their first recursion as $\mathtt{r}_1$, and so on. Note that `f1` and `f1'` have their unique solutions from the assumption. Now we will show that they can be one solution of `f2` and `f2'` with induction.

First think about the case where k is 0. The n-th recursion of `f2` is the bottom of the recursive call of `f2`. Here we assume that the recursion takes $\mathtt{r}_1$ as its second element of the argument. Then it returns `x` and `g0` $\mathtt{r}_1$ `h0`, which are the same as $\mathtt{x}_1$ and $\mathtt{h}_1$ respectively. This computation is the same as the 0-th recursion of `f1`.

Next consider the case of $k = m+1$ such that $0 < m < n-1$, and for all $0 < i \leq m$ the $(n-i)$-th recursion satisfies the hypothesis. Here assume that the $(n-m-1)$-th recursion of `f2` takes $\mathtt{r}_{m+2}$ as its second element of the argument. From the hypothesis it also takes $(\mathtt{x}_{m+1},\mathtt{h}_{m+1})$ as results of the next recursion, namely the $(n-m)$-th recursion. It passes `g3` $\mathtt{x}_{m+1}$ $\mathtt{r}_{m+2}$ $\mathtt{h}_{m+1}$ as the argument of the next recursion, which is the same as $\mathtt{r}_{m+1}$. It returns `(k` $\mathtt{x}_{m+1}$`,g2` $\mathtt{x}_{m+1}$ $\mathtt{r}_{m+2}$ $\mathtt{h}_{m+1}$`)` as its result, which is the same as $(\mathtt{x}_{m+2},\mathtt{h}_{m+2})$. Note that this computation is the same as the $(m+1)$-th recursion of `f1` and the hypothesis holds.

Finally we reach the case where $k = n$. From the hypothesis the n-th recursion of `f2` takes $(\mathtt{x}_n, \mathtt{h}_n)$ as results of the next recursion. It passes `g1` $\mathtt{x}_n$ $\mathtt{h}_n$ as the argument of the next recursion, which is the same as $\mathtt{r}_n$. This computation is the same as the n-th computation of `f1`.

As we have shown, $\{\mathtt{r}_0, \ldots, \mathtt{r}_n\}$, $\{\mathtt{x}_0, \ldots, \mathtt{x}_n\}$, and $\{\mathtt{h}_0, \ldots, \mathtt{h}_n\}$ organize one solution of `f2` and `f2'`, because they satisfy all equation without any conflict. From the assumption, `f2` and `f2'` have only one solution. Then the one and only solution of `f2` and `f2'` is the solution that we have derived, and it satisfies the proposition. □

Now we are ready to prove Theorem 4.1.2.

Proof. Here we prove Theorem 4.1.2. It is direct consequence of Lemma 4.2.2 and Lemma 4.2.4. Assume that maximum depth of the recursion is n. The result of the first recursion of `f1'` is the same as the second element of the argument of the nth recursion of `f2'`, because of the Lemma 4.2.4. That value is passed as the second result of `f2'`, and it is the same as the second result of the first recursion of `f2'`, because of the Lemma 4.2.2. Therefore the result of the whole computation of the `f2` is equivalent to that of `f1`. □

4.3 Characteristics of IO Swapping

IO swapping has some good properties. One is that it is self-inverse in the sense that applying IO swapping twice leads to the original function after removing unnecessary variables.

Theorem 4.3.1 (Self-inverseness of IO swapping)**.**
$\mathcal{T}[\![\mathtt{f}]\!]$ denotes the function `f` after applying IO swapping. Then,

$$\mathcal{T}[\![\mathcal{T}[\![\mathtt{f}]\!]]\!] = \mathtt{f}$$

provided that the function `f` is in the domain of Theorem 4.1.2.

Proof. Without loss of generality, we can assume that `f` has the following form.

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

We obtain $\mathtt{g} = \mathcal{T}[\![\mathcal{T}[\![\mathtt{f}]\!]]\!]$ as follows.

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

During the recursive call, the first argument of `f'` (denoted by the variable `y`) is always the same as the second argument (`x`), and the fourth argument (`r'`) and the third result (`r''`) do not change. Eliminating these unnecessary variables, we get the definition of `g` that is the same as that of `f`. □

Theorem 4.3.1 indicates that, when the effect of IO swapping becomes needless after some program manipulations, we can remove the effect of IO swapping by applying it one more time. It is natural because IO swapping is a rule that swaps call-time computations and return-time computations. In other words, IO swapping does nothing except swapping of them.

Another is about manipulability. If we have a transformation applicable to all functions in the domain of IO swapping, it is also applicable to any results of IO swapping. This comes from that the domain of IO swapping is equivalent to the range of IO swapping.

Theorem 4.3.2 (Self-morphismness of IO swapping)**.**
The domain and range of IO swapping are the same.

Proof. From Theorem 4.3.1, IO swapping should be a bijective morphism. And as also seen in the proof of Theorem 4.3.1, the range of IO swapping is a subset of its domain. Therefore its domain and range should be equal. □

Theorem4.3.2 indicates that IO swapping is manipulablity preserving in the sense that IO swapping keeps the structure of the recursion. It is good for combining IO swapping with other program transformation method because a large number of program transformations over recursive functions transform them by recognizing their recursion structure.

Next we remark on infinite structures. If a function with certain inputs recurs infinitely, the IO-swapped function with the same inputs also does an infinite recursion and does not return any value. This is because we cannot construct a proper recursion structure. Natural though it is, we should be careful about treatments of infinite structures with lazy evaluation. For example, consider the identity function on lists.

```
idL (a:x) = a:idL x
idL [] = []
```

If we give an infinite list as an input of `idL`, `idL` starts infinite computation and tries to compute an infinite list. Lazy evaluation enables `idL` to produce its results, for example we can take the head from it. On the other hand, the IO-swapped `idL` returns nothing with an infinite length of input. As this example illustrates, IO swapping does not conserve the meaning of functions in general, if the original function is non-strict and recurs infinitely. Note that there is no problem to apply IO swapping twice to functions which recur infinitely thanks to Theorem 4.3.1.

Finally, we give a short remark about computational complexities. Since what IO swapping manages is just a flipping, the leading orders of both time and space complexity of an IO-swapped function is equivalent to the original function. To be precise, complexities are different just a portion of two variables, one is the recursion argument of `f2` and another is the second element of the result of `f2` that bring the result of the whole recursion to the top of the recursion.

4.4 IO Swapping on Structural Recursions over Lists

Theorem 4.1.2 is very powerful. But it is sometimes too general to be applied to concrete programs. It is worth making to formalize a rule for structural recursions to discuss combination of other program manipulation methods with IO swapping, because they are suitable for program manipulation as mentioned in Chapter 3.

Corollary 4.4.1 (IO swapping for structural recurisions on lists)**.**
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' [] h = g1 h
        f1' (a:x') h = let r = f1' x' (g2 a r h)
                       in  g3 a r h

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

Proof. It is direct consequence of Theorem 4.1.2. □

Corollary 4.4.1 describes the IO swapping rule for list iterating functions with accumulative arguments. Recall that such functions are essentially structural recursions, because we can hide accumulative arguments by using higher-order results, as mentioned in Section 2.3.1.

The most significant point about Corollary 4.4.1 is that it enjoys the same properties described by Theorems 4.3.1 and 4.3.2. Applying IO swapping to structural recursions over lists results in structural recursions over lists and elimination of the effect of IO swapping is achieved by applying IO swapping once more. Now Corollary 4.4.1 therefore enables us to combine IO swapping with other techniques of program manipulation, once the concerned functions are defined as structural recursions. We will confirm its effectiveness in the following Chapters 5 and 6 with concrete examples.

It is worth mentioning about the relationship between Corollary 4.1.1 and Corollary 4.4.1. Corollary 4.1.1 is a special instance of Corollary 4.4.1 in the sense that functions in the domain and range of Corollary 4.1.1 have no circularity. Circularity is vital for IO swapping, because IO swapping basically makes circular programs from accumulative programs through flipping the direction of dependency between arguments and results. But in the case of Corollary 4.1.1, there appears no circularity because `foldl` is tail-recursive, in which the dependency from arguments to results is unnecessary. This absence of circularity makes Corollary 4.1.1 applicable under strict languages while not the general IO swapping.

4.5 IO Swapping on Trees

We have introduced IO swapping for linear recursions. The idea of IO swapping is turning over the recursion structure and it is not suitable to non-linear recursions. Applying IO swapping to non-linear recursions, we should "linearize" the concerned recursion. Here we introduce two ways.

One is strictly "linearize" the recursion. We make use of the fact (in compiler construction) that the stack frame, which has linear structure, generally captures the order and circumstances of function calls. Making explicit use of the stack frame enables us to linearizing non-linear recursions. For example, we can formalize a rule for structural recursions on trees as follows.

Theorem 4.5.1 (Liniarization of strucutual recursions on trees)**.**
The following two functions are equivalent.

```
f1 gl gr hn hl e t = f1' t e
  where
    f1' (Node l r) h = let lr = f1' l (gl lr rr h)
                           rr = f1' r (gr lr rr h)
                       in hn lr rr h
    f1' (Leaf n) h = hl n h
```

```
f2 gl gr hn hl e t = let [r] = f2' [t] [e] in r
  where
    f2' (Node l r:ts) (h:hs) = let lh = gl lr rr h
                                   rh = gr lr rr h
                                   (lr:rr:rs) = f2' (l:r:ts) (lh:rh:hs)
                               in hn lr rr h:rs
    f2' (Leaf n:ts) (h:hs) = (hl n h):(f2' ts hs)
    f2' [] hs = []
```

Proof. We will prove it by induction with a hypothesis: `head (f2' (t0:ts) (h0:hs)) = f1' t0 h0` and `(f2' (t0:t1:ts) (h0:h1:hs)) !! 1 = f1' t1 h1`. Note that this hypothesis is sufficient to prove this theorem.

If the height of `t0` and `t1` is 1, that is they are just a *Leaf*, then:

```
f2' (Leaf n0 : ts) (h0:hs) ⇒ hl n0 h0 : f2' ts hs
f2' (Leaf n0 : Leaf n1 : ts) (h0:hs) ⇒ hl n0 h0 : hl n1 h1 : f2' ts hs
```

the hypothesis holds.

Assume that the hypothesis holds if the height of `t` is less than k. If the height of `t0` is k, then:

```
f2' (Node l r : ts) (h0:hs)
    ⇒ let lh = gl lr rr h0
          rh = gr lr rr h0
          (lr:rr:rs) = f2' (l:r:ts) (lh:rh:hs)
      in hn lr rr h0:rs
    ⇒  {- From hypothesis: Note that the height of l and r is less than k -}
       hn (f1' l (gl lr rr h0)) (f1' r (gl lr rr h0)) h0: rs

f2' (t0 : Node l r : ts) (h0:h1:hs)
    ⇒  {- From the result above -}
    ⇒ let lh = gl lr rr h1
          rh = gr lr rr h1
          (lr:rr:rs) = f2' (l:r:ts) (lh:rh:hs)
      in f1' t0 h0 : hn lr rr h1:rs
    ⇒  {- From hypothesis: Note that the height of l and r is less than k -}
       hn (f1' l (gl lr rr h1)) (f1' r (gl lr rr h1)) h1: rs
```

the hypothesis holds. □

Starting from the function `f1`, first we determine the order of iteration in left-most depth-first fashion. To express stack frame explicitly, we rewrite the recursion structure such that it would push the arguments to the stack in the same order as the iteration and pop the results from the stack. Then we get the function `f2`. Because the recursion structure of the function `f2` is linear, we can apply Theorem 4.1.2. Using Theorem 4.5.1 with Theorem 4.1.2 we can actually swap call-time computations and return-time computations, because Theorem 4.5.1 does not change whether the computation is managed in call-time or return-time. Theoretically it is good, though, it is quite bad for practical use. It terribly breaks the recursion structure and there is little hope of combining it with other program transformations. We need more manipulable results.

Another way is to apply IO swapping to each spine of its structure of the recursion. For example, consider structural recursions on trees again. We regard it as a linear-recursion by considering that

the recursive call for the right subtree is iterated by another recursive function. This idea is formalized as follows.

Theorem 4.5.2 (IO swapping for structual recursions on trees)**.**
For all `gl`, `gr`, `hn`, `hl`, `e`, and `t`, the following two functions are equivalent.

```
f1 gl gr hn hl e t = f1' t e
    where f1' (Leaf n) h = hl n h
          f1' (Node lt rt) h = let lr = f1' lt (gl lr rr h)
                                   rr = f1' rt (gr lr rr h)
                               in hn lr rr h

f2 gl gr hn hl e t = let (Leaf n, h, r') = f2' t (hl n h) in r'
    where f2' (Leaf n) r = (t, e, r)
          f2' (Node lt' rt') lr = let (Node lt rt, h, r') = f2' lt' (hn lr rr h)
                                      rr = f2 gl gr hn hl (gr lr rr h) rt
                                  in (lt, gl lr rr h, r')
```

Proof. We prove it by induction. If the height of `t` is 1, then

```
f2 gl gr hn hl e (Leaf n)
   ⇒ let (Leaf n', h, r') = f2' (Leaf n) (hl n' h) in r'
   ⇒ let (Leaf n', h, r') = (Leaf n, e, hl n' h) in r'
   ⇒ hl n e
```

the proposition holds.

Assume that for all `t` whose height is less than k the proposition holds. If the height of `t` is k, then

```
f1 gl gr hn hl e t
  ⇒ f1' t e
  ⇒  {- IO swapping (Theorem 4.1.2) -}
    let (Leaf n, h, r') = f2'' t (hl n h) in r'
    where f2'' (Leaf n) r = (t, e, r)
          f2'' (Node lt' rt') lr = let (Node lt rt, h, r') = f2'' lt' (hn lr rr h)
                                       rr = f1 gl gr hn hl (gr lr rr h) rt
                                   in (lt, gl lr rr h, r')
   ⇒  {- From the hypothesis: Note that the height of rt is less than k -}
    let (Leaf n, h, r') = f2'' t (hl n h) in r'
    where f2'' (Leaf n) r = (t, e, r)
          f2'' (Node lt' rt') lr = let (Node lt rt, h, r') = f2'' lt' (hn lr rr h)
                                       rr = f2 gl gr hn hl (gr lr rr h) rt
                                   in (lt, gl lr rr h, r')
```

the proposition holds. □

As similar with the case of Theorem 4.5.1, Theorem 4.5.2 certainly swaps arguments and results but it does not preserve the recursion structure. `f2'` is not a structural recursion on trees anymore. Nevertheless, Theorem 4.5.2 is better than Theorem 4.5.1 for combining other program transformations. We will confirm its effect in Section 6.5.

Chapter 5

Play with TABA Using IO Swapping

In this chapter, we demonstrate derivations and manipulations of *There And Back Again* programs. The goal is to confirm how IO swapping works and how IO swapping is combined with other program transformations through non-trivial examples.

5.1 There And Back Again

There And Back Again (in short TABA), proposed by Danvy and Goldberg [DG02] is a program pattern where a recursive function traverses over its results as if they are recursion arguments. A typical example is a symbolic convolution function `cnv`, which takes two lists $[a_1, a_2, \ldots, a_n]$ and $[b_1, b_2, \ldots, b_n]$, and computes their symbolic convolution, i.e., $[(a_1, b_n), (a_2, b_{n-1}), \ldots, (a_n, b_1)]$. They show the following TABA-pattern `cnv`.

```
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 an unusual auxiliary function `walk`. When the input `x` is empty, `walk` uses the input `y` directly as a result, and this result is traversed together while `x` is traversed. Figure 5.1 shows the computation model of `cnv`.

Our objective is to show how to derive such programs. We can see that `cnv` yields to the following specification.

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

It seems that it is just a fusion problem. But no existing fusion method derives the program above, as we will discuss in Section 5.5. To derive TABA programs we use IO swapping, because IO swapping and TABA programs are very closely related. On one hand, IO swapping derives TABA programs. IO swapping brings an iteration over arguments to that of results because it swaps the call-time computation and the return-time computations. Then usual programs become TABA programs. On the other hand, IO swapping itself is an application of TABA programs. The TABA pattern is essentially necessary for expressing IO swapping rule.

We will carry out our derivation in compositional way: We first prepare a small TABA program by IO swapping, and afterwards we fuse functions into the small TABA function and make it a larger TABA function. We achieve fusions by fold promotion theorem (Theorem 3.3.1) and simple mutu tupling (Theorem 3.6.1), because these are enough for our objective.

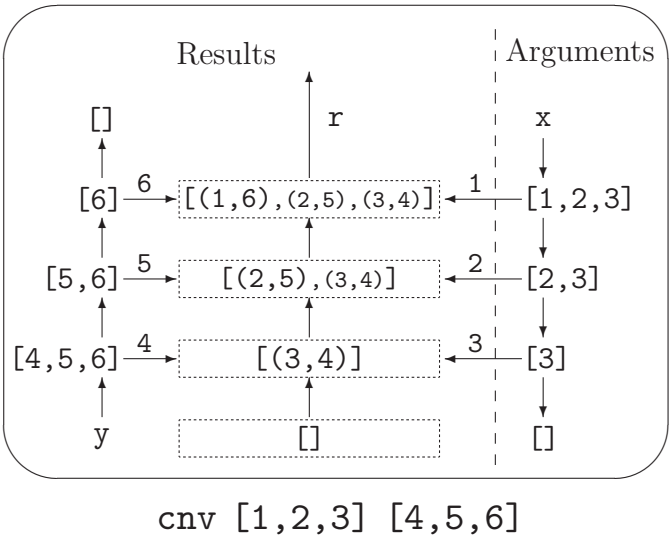

Figure 5.1. The model of computation process of `cnv`

5.2 List Reversal

It is expected that IO swapping derives TABA programs by introducing traversals over results from traversals over arguments. We confirm this observation by demonstrating a derivation of TABA-style `reverse` by IO swapping. It is also the first step to derive the `cnv`, for `cnv` is specified using `reverse`.

The function `reverse` is defined in terms of `foldl` as follows.

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

Applying Corollary 4.1.1 to `reverse`, we instantly get the following function `rev_n`. The function `rev_n` is exactly TABA form of `reverse` as mentioned by Danvy and Goldberg [DG05].

```
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')
```

In fact, both `foldl2` in Corollary 4.1.1 and `f2` in Corollary 4.4.1 are TABA programs. IO swapping is therefore a rule for deriving TABA programs.

5.3 Symbolic Convolution

In Section 4.4 we pointed out that applying IO swapping to structural recursions over lists results in structural recursions over lists. Therefore we can apply fusion law (Theorem 3.3.1) to the result of IO swapping if the original function is a structural recursion on lists. Combining this fact to the result of the previous section, we give a systematic and incremental way of derivation of TABA programs.

We here show a systematic derivation of the TABA form of `cnv` in the Section 5.1. Recall that `cnv` is specified as follows:

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

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

We derive a TABA-style `cnv` in compositional way. We fuse `zip x` to the TABA-style `reverse`, namely `rev_n`, which we have already derived in the previous subsection.

First of all, the function `rev_n` can be described in terms of `foldr`. This form is 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)
```

For promoting `id_zip` into `foldr` in the above, we check the following two conditions of Theorem 3.3.1.

```
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)
```

This program is essentially the same program as the TABA convolution function that we have seen in Section 5.1. We apply some known calculations to derive apparently the same program. First, applying 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 the 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 an unnecessary argument: The first argument of `cnv'` is not used at all for producing results.

```
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 exactly the TABA program of `cnv` introduced by Danvy and Goldberg.

This process indicates that combination of IO swapping with other program transformation methods is effective for deriving TABA programs. In Section 5.4 we develop a much more complicated example of a palindrome detecting program.

5.4 Palindrome Detecting

Danvy and Goldberg [DG02] put a 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 a palindrome in $\lceil n/2 \rceil$ recursive calls and with no auxiliary list." We will show that our derivation strategy works well for involving cases and solve this riddle.

We start by solving the problem in a straightforward way without being concerned with its efficiency. We check whether a list is a palindrome or not by turning up the latter half from 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 the following program.

```
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 warm fusion law [LS95], an extension of shortcut deforestation, but their theorem cannot cope with this problem. It is nice to see later that the existing program transformations 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.

```
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 and tupling law as `foldr`, as mentioned in Section 3.3 for general case.

```
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 id_alleq with foldr -}
  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')) (y,True) x -}
  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 alleqcnv with foldr' -}
  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, which we have discussed in Section 3.6, to avoid twice traversals of the same data structure `x` by two `foldr`'s.

```
palindrome x
  = let
    (y'',([],r)) = foldr' step (\x' x y'->(x',(y',True))) x x x y''
    step a r x' x y' = let (y'',(b:y,r')) = r (tail x') (tail x) y'
                       in (y'',(y, head x==b && r'))
    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 [] x2 x3 y' = (x2,(y',True))
```

Though this program has a circularity denoted by `y''` in the first line. We can eliminated it by removing unnecessary variables. We can eliminate the first element of the result and the forth argument of `pld` because they do not change during the whole computation steps of the `pld`. Now we get the following non-circular program.

```
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 [] x2 x3 = (x2,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 [] x2 = (x2,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. These derivations are an evidence of the manipulability of IO-swapped functions.

5.5 Symbolic Convolution Revisited

As we have mentioned in Section 5.1, no existing fusion method derives the `cnv`, that is the `cnv` introduced by Danvy and Goldberg, from the following specification.

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

For example, using descriptional composition or shortcut deforestation with Nishimura's higher-order removal, we get the following program after removing constant propagation.

```
cnv2 x y = let ([],r) = walk y [] in r
  where
    walk [] r = (x,r)
    walk (b:y) r = let (a:x,r') = walk y ((a,b):r)
                   in (x,r')
```

It is clearly a TABA program for a symbolic convolution function, but differs from the `cnv` because of a bit technical reason. The existing fusion methods produce the function whose recursion structure is the same as that of the function that produces the intermediate data structure. Because the producer function in the specification above is `reverse` with the recursion argument `y`, fusion should produce the function that uses `y` as a recursion argument. What we want to get is the `cnv` function whose recursion argument is `x`. This fact points out that we cannot derive the `cnv` without changing the specification. Actually, we can derive the `cnv` from the following specification with fusion:

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

where producer function use `x` as recursion argument.

What IO swapping achieves is to shift the selection of the recursion argument from `y` to `x`. Then we can derive the desired function without suffering the trouble of specification. In fact applying IO swapping to `cnv2` above derives the `cnv`.

It is worthy to mention about another derivation of the `cnv`, which is shown by Danvy and Goldberg [DG05]. It is based on continuation-passing style (in short, CPS) transformation and defunctionalization [Rey98]. They discovered that starting from some TABA-style programs, applying defunctionalization after CPS transformation removes TABA-style and derives a composition of two tail-recursive functions. Their derivation is the inverse of this step. Starting from a composition of tail-recursive functions, recognizing it as "defunctionalized" form, constructing a function before defunctionalization and CPS transformation, and they derive TABA programs. In short, they derive the `cnv` from this specification:

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

and as we have mentioned before, we can derive the `cnv` by fusion whenever we have this specification for `cnv`. Now we find another issue: What relationship are there between defunctionalization and fusion? Researching about this issue will be fruitful, because CPS transformation and defunctionalization have their applications [DN01] which is a bit different from the existing researches about fusion. But it exceeds the scope of this thesis.

Chapter 6

Reinforce the Power of Transformations by IO Swapping

In Chapter 5, we have shown an application of IO swapping as a program transformation and its effect has been confirmed. This chapter demonstrates an application of IO swapping as a *meta transformation*: IO swapping can take a program transformation and returns a program transformation that is IO-swapped transformation of the old one, and introduces symmetry of program manipulation for call-time computations and return-time computations. We show that the existing transformations such as Nishimura's higher-order removal [Nis04] and Meijer's higher-order promotion [Mei92] are derivable from more primitive cases.

After we discuss a general framework of manipulating recursive functions and the effect of IO swapping on it, we extend our methodology to manipulations of non-linear recursive functions. Throughout some examples, we confirm that IO swapping certainly indicates a relationship between manipulations of results and these of arguments.

6.1 IO Swapping as a Meta Transformation

Recall that fold promotion theorem does not work at all for accumulative arguments as we mentioned in Section 3.3. For example, the result of applying Theorem 3.3.1 to `length·reverse` is the following.

```
lengthreverse x = aux x []
    where aux [] h = length h
          aux (a:x) h = aux x (a:h)
```

On one hand, the result is not sufficient in the sense that no intermediate data structure is removed. On the other hand Theorem 3.3.1 certainly achieve its business, for Theorem 3.3.1 is a method for manipulating *results*. We attain a manipulation for *results* by Theorem 3.3.1, and we need another manipulation for accumulative *arguments*.

Now think about how to solve the problem above, that is to say how to make a new manipulation method for removing intermediate data structures in accumulative arguments. It seems that making a rule from nothing is uneconomical because the new manipulation should be similar with the manipulation for results, namely Theorem 3.3.1. Actually we can make the new rule from Theorem 3.3.1 with IO swapping. We are going to see the solution in Section 6.3, and here we explain the general idea.

Our idea is to change the world by IO swapping for manipulation: If arguments are hard to manipulate in a world, we will transfer the program to another world where arguments are manipulable.

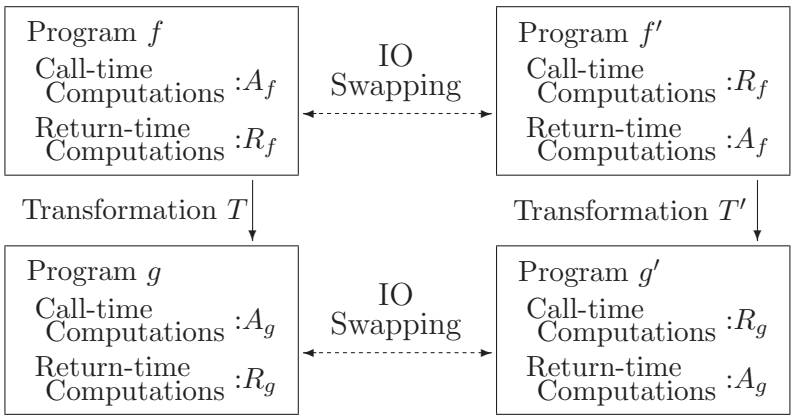

Figure 6.1. The framework provided by IO swapping

Such methodologies are used in quite various field, for example, descriptional composition based fusion method such as [Küh98][Küh99][CDPR99][Voi04] are based on the same methodology where the "another world" is the world of AGs. We chose the "another world" as IO-swapped world that comes from IO swapping. In the IO-swapped world, arguments in the original world are manipulable because it becomes results. For example, applying IO swapping to `lengthreverse` above, then we get the following function:

```
lengthreverse2 x = let ([],h) = aux x in length h
    where aux [] = (x,[])
          aux (b:y) = let (a:x',h) = aux y
                      in (x', a:h)
```

where the problem is the manipulation of results, and Theorem 3.3.1 is sufficient to achieve it.

Our method is summarized in Figure 6.1. 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'. For arbitrarily left-side pair related by T, such as f and g, IO swapping gives corresponding right-side pair, such as f' and g'. In consequence, we can define a program transformation T' by the relation of f' and g'. Note that T' is specified by the sequence of program transformations: Applying IO swapping after applying T after applying IO swapping. Since T' works as an IO-swapped transformation of T, we can make IO-swapped rule of T using IO swapping. In other words, IO swapping gives a relationship between manipulation of arguments and results of recursive functions.

Our method has its advantages than others, which follow from characteristics of IO swapping such as Theorem 4.3.1 and Theorem 4.3.2. First, our method is closed under the world of functional programs. Though view of functions differs, the IO-swapped world is essentially the same as the original world due to Theorem 4.3.1 and we can apply the same manipulation methods on them. Second, we can manipulate the functions of the IO-swapped world easily because of Theorem 4.3.2, and thus our method is easy to combine with other manipulation methods on functional programs. Lastly, we can deal with a very large set of programs, and return from the IO-swapped world easily by applying IO swapping once more. AGs have its drawback to expressiveness, where we cannot express anything but structural recursions. Relational AGs has enough expressiveness, but return to the functional world is problematic as we have seen in Section 4.1. Ours does not suffer from such problems.

6.2 Higher-Order Removal for Accumulative Arguments

In this section, we try to remove higher-order accumulative arguments. It is important even for fusion as mentioned in Section 3.8, though, higher-order removal for accumulative arguments is not so easy as that for results. Though we have the classic, simple and effective transformation rule namely η-expansion to remove higher-order of results, it does not work for arguments. In this section, we show that η-expansion with IO swapping achieves an effective higher-order removal for function arguments. This reveals that IO swapping reinforces power of program transformations so that transformations that copes with results become to be able to manipulate accumulative arguments also.

We use the function `sumCP`, defined as follows, as an example.

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

The function `sumCP` has a higher-order accumulative argument. We derive a first order program from it by using IO swapping and η-expansion.

Recall the discussion in the previous section. We can get the new, IO-swapped transformation by putting the transformation between a pair of IO swapping. So first we apply IO swapping.

Because `sumCP` is an instance of `foldl`,

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

we use Corollary 4.1.1 and get the following program.

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

In this program, higher-order values appear only in the results and applying η-expansion is sufficient for higher-order removal. We define a function `sum_n''` which takes an extra argument and gives it 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)
```

and replace `sum_n'` in the definition of `sumCP'` by `sum_n''`.

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

Higher-order removal is successfully achieved.

Now that the effect of IO swapping becomes needless, we eliminate it by applying IO swapping once more and derive a more familiar definition. Applying Corollary 4.4.1 backward, which is equivalent to applying IO swapping once more and eliminating unnecessary variables, results in the following definition.

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

This is ordinary definition of a function summing up all elements of a list. We can see that our strategy works successfully.

Now we will summarize the transformation above as a formal definition. As we have seen, the point of the 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 result of IO swapping is described as the following function.

```
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')
```

If we can define the rule for η-expansion for this function, then we can get higher-order removal rule for accumulative argument. Though it is not so obvious, we can achieve it by clarifying the intersection of the range of IO swapping and the domain of η-expansion. We should be careful that in the result of `f2'` the first element of first component (denoted by `x'`) is a recursion argument and the second element (`r'`) does no computation. We should manipulate the second element of first component (`h`) only. Intuitively, the rule should be something like the following.

```
f2a x h0 v0 = let ((x',h),r') = f2b' (x, g1 x' (h v0)) in r'
  where
    f2a' (y,r) = if p y then ((x, \v->g0 r h0 v),r)
                 else let ((x',h),r') = f2a' (k y, g3 x' r h)
                      in ((k x', \v->g2 x' r h v),r')
```

⇳

```
f2b x h0 v0 = let ((x',h),r') = f2b' (x, g1 x' h) v0 in r'
  where
    f2b' (y,r) v = if p y then ((x, g0 r h0 v),r)
                   else let ((x',h),r') = f2b' (k y, g3 x' r h)
                        in ((k x', g2 x' r h v),r')
```

This rule seems to be correct, but makes type error because recursive call of `f2b'` should be take one more argument, which corresponds to the argument of `h` because the extra arguments correspond to the argument of recursive call's results. If we can figure out what should be applied to `h`, then we will know what value should be passed as the extra argument of the recursive call. To do this, we specify the definition of `f2'` so that we can know the argument of `h`.

Lemma 6.2.1.
Assume that `g0`, `g1`, `g2`, `g3`, and `g4` are given functions. Then the following two functions `f2a` and `f2b` are equivalent.

```
f2a x h0 v0 = let ((x',h),r') = f2a' (x, g1 x' (h v0)) in r'
  where
    f2a' (y,r)
       = if p y then ((x, \v->g0 r h0 v),r)
         else
         let ((x',h),r') = f2a' (k y, g3 x' r)
             c = \v->g2 x' r v (h (g4 x' r v))
         in ((k x', c),r')
```

```
f2b x h0 v0 = let ((x',h),r') = f2b' (x, g1 x' h) v0 in r'
  where
    f2b' (y,r) v
       = if p y then ((x, g0 r h0 v),r)
         else
         let ((x',h),r') = f2b' (k y, g3 x' r) (g4 x' r v)
         in ((k x', g2 x' r v h),r')
```

Proof. It is proved by η-expansion, as similarly above. Start from `f2a`, we define `f2b'` as

```
f2b' (y,r) v = let ((x',h),r') = f2a' (y,r) in ((x',h v),r')
```

Then we get `f2b`. □

Now we are ready to get the higher-order removal rule for function argument. Applying IO swapping both `f2a` and `f2b`, we get the following theorem.

Theorem 6.2.2 (Higher-order removal for accumulative arguemnts)**.**
Assume that `g0`, `g1`, `g2`, `g3`, and `g4` are given functions. Then the following two functions `f1` and `f2` are equivalent.

```
f1 x h0 = let r = f' (x,\v->g0 r h0 v) in r
  where
    f' (x',h)  = if p x' then g1 x' (h e)
                 else let c = \v->g2 x' r v (h (g4 x' r v))
                          r = f' (k x', c)
                      in g3 x' r

f2 x h0 = let (r,v) = f' (x, g0 r h0 v) in r
  where
    f' (x',h) = if p x' then (g1 x' h, e)
                else let (r,v) = f' (k x', g2 x' r v h)
                     in (g3 x' r, g4 x' r v)
```

Proof. From Lemma 6.2.1, currying the arguments of `f2b'` so that it makes triple, and applying IO swapping backward to both `f2a` and `f2b`, then we get `f1` and `f2` respectively. □

We found that combination of Theorem 6.2.2 and usual η-expansion works as similar to the higher-order removal method proposed by Nishimura [Nis04], which we have mentioned in Section 3.8, in the sense that our rule introduces an extra result that corresponds to the argument of the higher-order argument. He formalized it as a new rule from AG-based method namely descriptional composition, while we have derived it from η-expansion and IO swapping. We extend this results to the non-linear-recursive functions in Section 6.5.3.

6.3 Fusing Accumulative Functions

To confirm the effect of meta transformation of IO swapping, this section exploits fusion for accumulative functions. As we saw in Section 3.3 and Section 6.1, Theorem 3.3.1 has its drawback to treating accumulative functions because it cannot promote a function to accumulative part. We solve this problem by IO swapping.

As the previous section, we want to know the intersection of range of IO swapping and domain of Theorem 3.3.1, and make a transformation rule for it. Because the domain of Theorem 3.3.1 is `foldr` function, it is appropriate to think about IO swapping on structural recursions over lists, that is Corollary 4.4.1. Now we get the following lemma.

Lemma 6.3.1.
Assume that `psi`, `g0`, `g1`, `g2`, and `g3` are given functions. Then the following two functions `f2a` and `f2b` are equivalent provided that `psi(g0 r h0) = g0' r (phi h0)`, `psi(g2 a r h) = g2' a r (psi h)`, and `g3 a r h = g3' a r (psi h)`.

```
f2a x h0 = let ([], h, r') = id_psi_id (f2a' x (g1 h)) in r'
  where
    f2a' [] r = (x, g0 r h0, r)
    f2a' (b:y) r = let (a:x',h,r') = f2a' y (g3 a r h)
                   in (x', g2 a r h, r')
    id_psi_id (a,b,c) = (a, psi b, c)

f2b x h0 = let ([], h, r') = f2b' x (g1 h) in r'
  where
    f2b' [] r = (x, g0' r (phi h0), r)
    f2b' (b:y) r = let (a:x',h ,r') = f2b' y (g3' a r h)
                   in (x', g2' a r h, r')
```

Proof. It is direct consequence of Theorem 3.3.1. With following two calculations, we can fuse `f2a'` and `id_psi_id` into `f2b'`.

```
(id_psi_id·)(\r->(x, g0 r h0, r)) ⇒ \r->(x, psi(g0 r h0), r)
                                  ⇒ \r->(x, g0' r (phi h0), r)

(id_psi_id·)((\a y r->let (a:x',h,r') = y (g3 a r h) in (x', g2 a r h, r')) a y)
    ⇒ \r->let (a:x',h,r') = y (g3 a r h) in (x', psi (g2 a r h), r')
    ⇒ \r->let (a:x',h,r') = y (g3' a r (psi h)) in (x', g2' a r (psi h), r')
    ⇒ \r->let (a:x',h,r') = ((id_psi_id·) y) (g3' a r h) in (x', g2' a r h, r')
```

□

From this lemma, we get the following theorem.

Theorem 6.3.2.
Assume that `psi`, `g0`, `g1`, `g2`, and `g3` are given functions. Then the following two functions `f2a` and `f2b` are equivalent provided that `psi(g0 r h0) = g0' r (phi h0)`, `psi(g2 a r h) = g2' a r (psi h)`, and `g3 a r h = g3' a r (psi h)`.

```
f1a x h0 = let r = f1a' x (g0 r h0) in r
  where
    f1a' [] h = g1 (psi h)
    f1a' (a:x') h = let r = f1a' x' (g2 a r h)
                    in  g3 a r h
```

```
f1b x h0 = let r = f1b' x (g0' r (phi h0)) in r
  where
    f1b' [] h = g1 h
    f1b' (a:x') h = let r = f1b' x' (g2' a r h)
                    in  g3' a r h
```

Proof. From Lemma 6.3.1, applying IO swapping backward to both `f2a` and `f2b`, then we get `f1a` and `f1b` respectively. □

Theorem 6.3.2 is equivalent to Theorem 3.4.2. As we have mentioned in Section 3.4, combining Theorem 3.3.1 with Theorem 6.3.2 gives a new and useful fusion law for accumulative functions, which is almost the same as the Theorem 3.4.1. We define a function `acc` as follows, which is corresponds to non-circular case of `f1a` and `f1b` above.

```
acc h0 g1 g2 g3 x = acc' x h0
  where acc' [] h = g1 h
        acc' (a:x) h = g3 a (acc' x (g2 a h)) h
```

Now we give a fusion law for `acc`

Theorem 6.3.3 (Fusion for accumulative functions)**.**
Provided that

```
phi (g1 h)     = g1' (psi h)
psi (g2 a h)   = g2' a (psi h)
phi (g3 a r h) = g3' a (phi r) (psi h)
psi h0         = h0'
```

Then,

```
phi (acc h0 g1 g2 g3 x) = acc h0' g1' g2' g3' x
```

Proof. Define a function `g3''` such that

```
phi (g3 a r h) = g3'' a (phi r) h
g3'' a r h = g3' a r (psi h)
```

Note that existence and computability of `g3''` are no matter, for it is an intermediate function for proof. Now Theorem 3.3.1 proves `phi (acc h0 g1 g2 g3 x) = acc h0 (g1'·psi) g2 g3'' x`, and Theorem 6.3.2 proves `acc h0 (g1'·phi) g2 g3'' x = acc h0' g1' g2' g3' x`. □

As the same as the previous subsection, we can reinforce a transformational power to a primitive transformation so that it can manipulate accumulative arguments, and derive powerful transformation. These examples convince us of the effectiveness of our method. We are going to extend this result so that it can deal with the structural recursions on trees in Section 6.5.2

6.4 Discussion : Manipulation of Recursive Functions

Here we will summarize what we have done in the first half of this chapter, namely Sections 6.2 and 6.3. Our aim is to give a manipulation method for recursive functions. A recursive function is, from a viewpoint of relational AGs, a sequence of logical relations as we have explained in Section 4.1. If a recursive function `f` has a definition,

```
f x h = let r' = f x' h' in r
```

then `f` organizes a sequence of logical relations from the top to the bottom of the recursion, which is described in terms of the relationship between `x` and `x'`, `h` and `h'`, and `r` and `r'`. Therefore manipulation of recursive functions is to find a new sequence of logical relations that is proper for the aim of the manipulation. In other words a task of a manipulation method is to give a way to find a proper logical relation that agrees with the aim. For example, η-expansion gives the following rewriting scheme is proper to remove function results.

```
f x h = r  ⇒  f2 x h k = r k
```

We do higher-order removal by assuming this `f2` is proper for a new logical relation, namely the specification of the new recursive function, as we have done in Section 6.2. As another example, think about a fusion problem. To fuse `sum` into a recursive function `f`, fold promotion theorem indicates that the following scheme is proper to achieve fusion.

```
f x h = r  ⇒  f2 x h = (sum r)
```

In short, a manipulation for recursive functions is a scheme to give a proper definition of a recursive function.

Now consider IO swapping. IO swapping is also a manipulation of recursive functions, which gives the following scheme to swap its arguments and results:

```
f x h = r  ⇒  f' y r = (x, h, r')
```

where `y` and `r'` are not essential for its computation. To achieve higher-order removal for accumulative arguments, we manipulated the function `f'` as follows:

```
f' y r = (x, h, r')  ⇒  f2' y r k (x, h k, r')
```

and it coincides with the following scheme in the world without IO swapping, as we have seen in Section 6.2.

```
f x h = r  ⇒  f x (h k) = (r,k)
```

It actually corresponds to the IO-swapped manipulation of η-expansion. As for fusion, it is similar. IO swapping gives the following scheme for fusing `sum` to accumulative argument, as we have seen in Section 6.3:

```
f x h = r  ⇒  f2 x (sum h) = r
```

and it is IO-swapped manipulation of fold promotion theorem. In general, if we have a program transformation method T for recursive function, which is expressed in terms of the following scheme:

$$T[\![\texttt{f x h = r}]\!] \;\Rightarrow\; \texttt{f' x}\ T_h[\![\texttt{h}]\!] = T_r[\![\texttt{r}]\!]$$

IO swapping indicates that the following scheme is proper to achieve an IO-swapped manipulation of T:

$$T'[\![\texttt{f x h = r}]\!] \;\Rightarrow\; \texttt{f' x}\ T_r[\![\texttt{h}]\!] = T_h[\![\texttt{r}]\!]$$

This result is a consequence of the fact that IO swapping certainly swaps the arguments and results of recursive functions. Now that we can make their arguments and results symmetrical, it is rather natural that IO-swapped schemes work appropriately. We have used this fact implicitly in Section 6.2 and Section 6.3, and we are going to use it explicitly to manipulate non-linear recursions in Section 6.5.

6.5 Meta Transformations for Non-linear Recursions

In this section, we are going to discuss how IO swapping works for non-linear recursive functions. As we are going to explain in Section 6.5.1, the most difficult point to manipulate non-linear recursive functions is how to find a proper specification of a new recursive function. IO swapping helps this step when we want to manipulate arguments. From the manipulation of results IO swapping indicates a proper specification of new recursive function, and this specification certainly corresponds to the IO-swapped manipulation. We are going to confirm it throughout fusion and higher-order removal.

6.5.1 Problems of Manipulating Non-linear Recursive Functions

Even without IO swapping, manipulations for non-linear recursive functions are truly difficult. To see the difficulties, we introduce the following function `flat` which flattens a tree into a list:

```
flat t = flat' x []
    where flat' (Leaf n) h = n:h
          flat' (Node l r) h = flat' l (flat' r h)
```

Even though `flat` is not so involved, It is difficult to manipulate `flat`. For example, consider the following fusion problem.

```
sumflat t = sum (flat t)
```

First we use simple unfolding-folding calculation as follows:

```
sumflat t = sum (flat t)
          ⇒ sum (flat' t [])
```

Here it seems that we need another function name,

```
sf1 t h = sum (flat' t h)
```

and continue the calculation as follows:

```
sumflat t ⇒ sum (flat' t [])
          ⇒ sf1 t []
sf1 (Leaf n) h ⇒ sum (flat' (Leaf n) h)
               ⇒ sum (n:h)
               ⇒ n + sum h
sf1 (Node l r) h ⇒ sum (flat' (Node l r) h)
                 ⇒ sum (flat' l (flat' r h))
                 ⇒ sf1 l (flat' r h)
```

Then we get the following program.

```
sumflat t = sf1 x []
    where sf1 (Leaf n) h = n + sum h
          sf1 (Node l r) h = sf1 l (flat' r h)
```

It is actually correct but unsatisfactory. Most of its intermediate data structures remain yet.

Next we try to achieve it by fold promotion theorem. Recall that structural recursions on trees are expressed in terms of the following function `foldTree`:

```
foldTree g1 g2 (Leaf n) = g2 n
foldTree g1 g2 (Node l r) = g1 (foldTree g1 g2 l) (foldTree g1 g2 r)
```

and we can write `flat'` by `foldTree` as follows:

```
flat' t = foldTree (\l r h-> l(r h)) (\n h-> n:h) t
```

Using Theorem 3.3.2, we calculate as follows:

```
(sum·)((\n h-> n:h) n) ⇒ (\h-> n+sum h)
(sum·)((\l r h-> l(r h)) l r) ⇒ (\h-> (sum·l)(r h)))
```

Here we reach to a stick. We cannot make `(sum·)` reach to `r` and satisfy the condition of fusion. Theorem 3.3.2 indicates that to achieve a sufficient transformation we should make all recursions have the same recursive definition. We cannot make the calculation succeed because of this condition, nevertheless it is truly proper, for if we omit it we get an insufficient result as we have got from unfolding-folding calculation. To solve this problem, we need to change the specification of the produced recursive function. We have used the following equation as a specification:

```
sf1 t h = sum (flat' t h)
```

but the calculations above have shown this specification is not sufficient. In fact, the following one is a proper:

```
sf2 t (sum h) = sum (flat' t h)
```

because we need to make `(sum·)` reach to the recursion of its right subtree.

It is a typical difficulty that comes from non-linear use of the recursion. If we have non-linear use of the recursion we often need to keep these recursions are the same after a manipulation. It makes a great hardship. To keep them the same we need to find a proper specification of the resulting recursive function. As we have explained in Section 6.4, a manipulation of a recursive function is to find a new invariant that holds throughout its recursion. In a linear-recursion, the invariant is used only by is its parent recursion, then top-down manipulation is enough because it implicitly corresponds to induction. In a non-linear recursion, the invariant is used not only by its parent recursion step but also by its brothers, then explicit bottom-up induction is indispensable. A necessity to find a proper specification reflects a necessity of induction.

We are going to show that IO swapping indicates a proper specification.

6.5.2 Fusion for Accumulative Functions on Tree

As we have seen in Section 6.5.1, fusion for `flat` is difficult. We need to find a proper specification of the recursive function. Naive transformation derives the following specification:

```
sf1 t h = sum (flat' t h)
```

and it is not proper. Now we will show IO swapping helps to derive the proper specification.

Recall that the result of a naive fusion is as follows.

```
sumflat t = sf1 x []
    where sf1 (Leaf n) h = n + sum h
          sf1 (Node l r) h = sf1 l (flat' r h)
```

Here `sum` is applied to the accumulative argument `h` thus it is a problem to fuse a function into accumulative arguments. As we have seen in Section 6.3, IO swapping enables to solve such a problem for linear-recursions. We try manipulating by the same way with that in Section 6.3. First, applying IO swapping for tree iterating functions (Theorem 4.5.2) to `sf1`, we get the following program.

```
sf1_ t e = let (Leaf n, h, r') = sf1_' t (n+sum h) in r'
   where sf1_' (Leaf n) r = (t, e, r)
         sf1_' (Node lt' rt') lr = let (Node lt rt, h, r') = sf1_' lt lr
                                   in (lt, flat' rt h, r' )
```

Extracting `sum` as follows,

```
let (Leaf n, h, r') = sf1_' t (n+sum h) in r'
   ⇒ let (Leaf n, h, r') = id_sum_id (sf1_' t (n+h)) in r'
      where id_sum_id (a,b,c) = (a, sum b, c)
```

and we try fusing as follows.

```
id_sum_id (sf1_' (Leaf n) r) ⇒ (t, sum e, r)
id_sum_id (sf1_' (Node lt' rt') lr)
    ⇒ let (Node lt rt, h, r') = sf1_' lt lr
      in (lt, sum (flat' rt h), r')
    ⇒ let (Node lt rt, h, r') = sf1_' lt lr
      in (lt, sf1 rt h, r')
```

Now we face a stick again. The `sum` (or `id_sum_id`) disappears, and we cannot derive any recursive function. This result indicates that the definition of `sf1` is certainly not proper.

We should find a new and proper hypothesis. To get it, recall the calculation for the base case above:

```
id_sum_id (sf1_' (Leaf n) r) ⇒ (t, sum e, r)
```

where `e` is the initial accumulative argument of `sf1_`. This equation indicates that the accumulative argument is summed before used in the recursion, whenever we success the fusion. In other words, the accumulative argument has no need to be a list but a summed value is enough. That is, the following specification of the new recursive function may be much more proper.

```
sf2 t (sum h) = sf1 t h
```

Note that it is certainly IO-swapped manipulation of Theorem 3.3.2. Theorem 4.5.2 actually indicates that IO-swapped manipulation is proper. Now that we get a new specification, we retry calculation to get the definition of `sf2`

```
sf2 (Leaf n) h ⇒ sf1 (Leaf n) h'    {- where h = sum h' -}
               ⇒ n + sum h'
               ⇒ n + h

sf2 (Node l r) h ⇒ sf1 (Node l r) h'    {- where h = sum h' -}
                 ⇒ sf1 l (flat' r h')
                 ⇒ sf2 l (sum (flat' r h'))
                 ⇒ sf2 l (sf1 r h')
                 ⇒ sf2 l (sf2 r (sum h'))
                 ⇒ sf2 l (sf2 r h)
```

```
sumflat t ⇒ sf1 t []
          ⇒ sf2 t (sum [])
          ⇒ sf2 t 0
```

Calculations are successfully finished, and we get the following program.

```
sumflat t = sf2 t 0
   where sf2 (Leaf n) h = n + h
         sf2 (Node l r) h = sf2 l (sf2 r h)
```

We can achieve the transformation completely.

Now we formalize these calculations above. Its formalization is achieved as similar way with that in Section 6.3. We clarify the intersection of range of IO swapping and domain of the transformation, which is fusion in this case, and confirm the condition to succeed in the transformation with finding a proper specification of the resulting recursive function. We give a fusion law for the following function `accTree`, which is a general form of functions that iterate its computation over trees with accumulative arguments.

```
accTree g1 g2 g3 g4 h0 t = accTree' t h0
  where accTree' (Leaf n) h = g1 n h
        accTree' (Node l r) h = let lr = accTree' l (g2 lr rr h)
                                    rr = accTree' r (g3 lr rr h)
                                in g4 lr rr h
```

Theorem 6.5.1 (Fusion for Accumulative Functions on Tree)**.**
Provided that

```
g1 n h = g1' n (phi h)
phi (g2 lr rr h)  = g2' lr rr (phi h)
phi (g3 lr rr h)  = g3' lr rr (phi h)
phi (g4 lr rr h)  = g4' lr rr (phi h)
psi h0  = h0'
```

Then,

```
accTree g1 g2 g3 g4 h0 t =  accTree g1' g2' g3' g4' h0' t
```

for all `g1`, `g2`, `g3`, `g4`, `h0`, and `t`.

Proof. We will give an inductive proof about the height of the input tree `t`. If the height of `t` is 1, that is `t` is *Leaf* `n`, then

```
accTree g1 g2 g3 g4 h0 (Leaf n)  ⇒ g1 n h0
                                 ⇒ g1' n (phi h0)
                                 ⇒ g1' n h0'
                                 ⇒ accTree g1' g2' g3' g4' h0' (Leaf n)
```

the hypothesis holds.

Assume that the hypothesis holds for all `t` whose height is less than k. If the height of `t`=*Node* `l r` is k, then

```
accTree g1 g2 g3 g4 h0 (Node l r)
     ⇒ let lr = accTree g1 g2 g3 g4 (g2 lr rr h0) l
           rr = accTree g1 g2 g3 g4 (g3 lr rr h0) r
       in g4 lr rr h0
     ⇒   {- From hypothesis: Note that the height of l and r is less than k -}
       let lr = accTree g1' g2' g3' g4' (g2 lr rr h0) l
           rr = accTree g1' g2' g3' g4' (g3 lr rr h0) r
       in g4 lr rr h0
     ⇒ let lr = accTree g1' g2' g3' g4' (g2' lr rr (phi h0)) l
           rr = accTree g1' g2' g3' g4' (g3' lr rr (phi h0)) r
       in g4' lr rr (phi h0)
     ⇒ let lr = accTree g1' g2' g3' g4' (g2' lr rr h0') l
           rr = accTree g1' g2' g3' g4' (g3' lr rr h0') r
       in g4' lr rr h0'
     ⇒ accTree g1' g2' g3' g4' h0' (Node l r)
```

the hypothesis holds. □

This theorem is an extension of Theorem 6.3.2 to the tree iterating functions. As similar with the case of Section 6.3, combining this theorem with usual fold promotion theorem gives higher-order promotion theorem namely Theorem 3.4.1.

The use of our strategy is not limited to fusion. In general, if we have a recursive function `f` on trees, we try applying some transformation T to an auxiliary function of IO-swapped `f`, namely `f_'`, and get the following rewiring step:

$$T[\![\ \texttt{f\_' (Leaf n) r = (t, e, r)}\ ]\!] \ \Rightarrow\ \ \texttt{f\_' (Leaf n)}\ T_r[\![\texttt{r}]\!] = \texttt{(t,}\ T_h[\![\texttt{e}]\!]\texttt{, r)}$$

indicates the existence of a much more proper specification `g`,

$$\texttt{g t}\ T_h[\![\texttt{e}]\!] = T_r[\![\texttt{r}]\!]\ \ \textbf{where}\ \texttt{r = f t e}$$

and it points out that our discussion in Section 6.4 is certainly applicable for manipulations of non-linear recursions. We are going to confirm its effectiveness by using it for a higher-order removal problem.

6.5.3 Removing Higher-Order Accumulative Arguments on Tree

In this subsection, we try to remove higher-order terms in accumulative arguments, as the same as Section 6.2. We can derive another definition of `sumflat` by choosing another specification of recursive function as follows:

```
sf3 t (\n->n+sum h) = sf1 t h
```

and applying Theorem 6.5.1 we get the following program.

```
sumflat t = sf3 t id
   where sf3 (Leaf n) h = h n
         sf3 (Node l r) h = sf3 l (\n-> n + sf3 r h)
```

The function `sf3` has a higher-order accumulative argument. We remove it with η-expansion and IO swapping.

First we apply Theorem 4.5.2 to `sf3`. The result is as follows.

```
sf3_ t e = let (Leaf n, h, r') = sf3_' t (h n) in r'
   where sf3_' (Leaf n) r = (t, e, r)
         sf3_' (Node lt' rt') lr = let (Node lt rt, h, r') = sf3_' lt' lr
                                   in (lt, \n-> n + sf3 rt h, r' )
```

Because the second element in the result is a higher-order term, we try to use an extension of η-expansion in the same manner with Section 6.2.

```
sf4_' tt (r,k) = let (t, h, r') = sf3_' tt r in (t, h k, r')
```

Applying an extension of η-expansion is fail as similar with the fusion case. The existence of `sf3` prevents the transformation. What we need is a new specification of the resulting recursive function. From our strategy we derive a candidacy of a proper specification from the base case.

```
sf3_' (Leaf n) r = (t, e, r)  ⇒  sf4_' (Leaf n) (r,k) = (t, e k, r)
```

And it is actually IO-swapped manipulation of η-expansion. Now we get a specification:

```
sf4 t (e k) = (r,k)     {- where r = sf t e -}
```

We start applying an extension of η-expansion with this hypothesis. For the base case,

```
sf4_' (Leaf n) (r,k) = (t, e k, r)
```

and top of the recursion is easy.

```
sf4_ t e  = let (Leaf n, h, r') = sf4_' t (h, n) in r'
```

Step case is as follows:

```
sf4_' (Node lt' rt') (lr,k)
    ⇒ let (Node lt rt, h, r') = sf3_' lt' lr in (lt, k + sf3 rt h, r' )
    ⇒ let (Node lt rt, h, r') = sf3_' lt' lr
          (rr,rk) = sf4 rt (h rk)
      in (lt, k + rr, r')
    ⇒ let (Node lt rt, h, r') = sf4_' lt' (lr,rk)
          (rr,rk) = sf4 rt h
      in (lt, k + rr, r')
```

Then we get the following definition.

```
sf4_ t e  = let (Leaf n, h, r') = sf4_' t (h, n) in r'
  where
     sf4_' (Leaf n) (r,k) = (t, e k, r)
     sf4_' (Node lt' rt') (lr,k) = let (Node lt rt, h, r') = sf4_' lt (lr,rk)
                                       (rr,rk) = sf4 rt h
                                   in (lt, k + rr, r')
```

Higher-order removal is certainly achieved. Finally we remove the effect of IO swapping, and we get the following program.

```
sumflat t = let (r,k) = sf4 t k in r
    where sf4 (Leaf n) h = (h,n)
          sf4 (Node lt rt) h = let (lr,k) = sf4 lt (k+rr)
                                   (rr,rk) = sf4 rt h
                               in (lr,rk)
```

Here we get a first order program. We succeed in removing higher-order in the accumulative argument.

Now we formalize the calculation above into a rule as Section 6.2. The following theorem is an extension of Theorem 6.2.2 to tree iterating functions.

Theorem 6.5.2 (Higher-order removal for accumulative arguments on trees)**.**
Assume that `g0`, `g1`, `g2`, `g3`, `g4`, `g5`, `g6`, and `g7` are given functions. Then the following two functions `f1` and `f2` are equivalent for all `x` and `h`.

```
f1 t h = let r = f1' t (\v->g0 r (h(g5 r v))) in r
  where f1' (Leaf n) h = g1 n (h(g6 n))
        f1' (Node l r) h = let lr = f1' l (\lv->g2 lr rr lv)
                               rr = f1' r (\rv->g3 lr rr rv (h(g7 lr rr rv)))
                           in g4 lr rr h

f2 t h = let (r,v) = f2' t (g0 r (h(g5 r v))) in r
  where f2' (Leaf n) h = (g1 n h, g6 n)
        f2' (Node l r) h = let (lr,lv) = f2' l (g2 lr rr lv)
                               (rr,rv) = f2' r (g3 lr rr rv h)
                           in (g4 lr rr h, g7 lr rr rv)
```

Proof. We give an inductive proof concerning with the height of the input tree. Start from the function `f1`, we show that the following specification is proper to remove the higher-order accumulative argument.

```
f2' t (h k) = (r,k)  {- where f1' t h = r, for all t and h -}
```

We use this specification as the hypothesis, for it is sufficient to prove the theorem.

If the height of the input tree is 1, that is `t` is a *Leaf* `n`, then

```
f1' (Leaf n) h ⇒ g1 n (h(g6 n))
f2' (Leaf n) h' ⇒ (g1 n h', g6 n)
```

here `h'`$=$ `h(g6 n)` is appropriate and the hypothesis holds.

Assume that the hypothesis holds if the height of `t` is less than k. If the height of `t`$=$*Node* `l r` is k then

```
f1' (Node l r) h
    ⇒ let lr = f1' l (\lv->g2 lr rr lv)
          rr = f1' r (\rv->g3 lr rr rv (h(g7 lr rr rv)))
      in g4 lr rr
    ⇒  {- From hypothesis: Note that the height of l and r is less than k -}
      let (lv,lr) = f2' l (g2 lr rr lv)
          (rr,rv) = f2' r (g3 lr rr rv (h(g7 lr rr rv)))
      in g4 lr rr

f2' (Node l r) h'
    ⇒ let (lv,lr) = f2' l (g2 lr rr lv)
          (rr,rv) = f2' r (g3 lr rr rv h')
      in (g4 lr rr, g7 lr rr rv)
```

here `h'` $=$ `h(g7 lr rr rv)` is appropriate and the hypothesis holds. □

As we have mentioned in Section 6.2, this theorem is quite similar with the higher-order removal method proposed by Nishimura.

Chapter 7

Manipulating Circular Functions

In this chapter, we will try to manipulate circular programs based on IO swapping. IO swapping gives an insight: Circular programs are related with accumulative programs. To say more precisely, circularities are IO-swapped appearance of accumulation. According to this insight we well show that the difficulty to manipulating a circular program is not circularity itself, but a non-linear use of values.

7.1 Relating Circular and Accumulative Functions

First of all, a circularity is a relative of an accumulation. As we have explained in Section 2.3.2, a circularity is computational dependency from a result to an argument. It is an IO-swapped representation of an accumulation, namely computational dependency from an argument to a result. To see this, consider the following `repminl` function:

```
repminl x = let (r,m) = repminl' x m in r
   where repminl' [a] m = ([m], a)
         repminl' (a:x) m = let (r,n) = repminl' x in (m:r, min a n)
```

which corresponds to the list version of the `repmin` problem. It has an accumulation denoted by the accumulative argument `m` and a global circularity in the top of the recursions denoted by the variable `r`. Applying IO swapping we get the following program.

```
repminl_ x = let ([a],m,r') = aux x ([m],a) in r'
     where aux (b:y) (r,n) = let (a:x,m,r')= aux y (m:r, min a n)
                             in (x,m,r')
           aux [b] (r,m) = (x,m,r)
```

Though the global circularity turns into an accumulation of the base case, every recursive step has a local circularity, denoted by `m`. These are originated from the computational dependency of `repminl'`. The relationship between circularities and accumulations are apparent. Circularities are not peculiar.

In fact, IO swapping itself is led from this recognition of circularities. As we have mentioned in Section 4.4, circularities are vital for IO swapping and it frequently produces circular programs. But it makes no serious problems, except for a failure of evaluation in a strict setting, because circularities are nothing but another appearance of accumulations. In addition, and against the intuition, circular programs are essentially manipulatable.

7.2 Fusing Circular Functions

In this section, we will derive a fusion law for circular programs. Our methodology is that we have introduced in Chapter 6. We derive a fusion law for circular programs from that of accumulative programs, as the IO-swapped manipulation of it. However, it is not a direct consequence of IO swapping. Non-linear use of values prevents manipulations, and we need to remove them. We will clarify the difficulty to manipulate circular programs.

First of all, circularity itself does not disturb manipulations. Recall `repminl` function, namely the list version of `repmin`.

```
repminl x = let (r,m) = repminl' x m in r
   where repminl' [a] m = ([m], a)
         repminl' (a:x) m = let (r,n) = repminl' x
                            in (m:r, min a n)
```

To fuse `repminl` with other functions, for example `map (1+)`, is easy.

```
map (1+) (repminl x) ⇒ let (r,m) = repminl' x m in map (1+) r
                     ⇒ let (r,m) = mapS_id (repminl' x m) in r
                            where mapS_id (r,m) = (map (1+) r,m)
```

Now that we promote `map` to the inside of the circularity, normal fusion methods are sufficient to achieve fusion. As we have explained in the previous section, circularities are nothing but a representation of computation flows and they do not disturb manipulations more than accumulative arguments. Actually we have manipulated circularities both implicitly and explicitly through this thesis.

But the same method does not works the following `cycle` function.

```
cycle x = let r = cyc x r in r
    where cyc [] h = h
          cyc (a:x) h = a: cyc x h
```

If we do as the same as the previous, we fail as follows:

```
map (1+) (cycle x) ⇒ let r = cyc x r in map (1+) r
                   ⇏ let r = map (1+) (cyc x r) in r
```

because, the accumulative argument of `cyc`, denoted by `r`, should be the list that is not affected by `map`. Then we cannot promote `map` over the circularity.

Consequently, and we have pointed out it in Section 3.7, the problem of `cycle` does not come from the circularity, but comes from the non-linear use of a value. The function `cyc` uses its result twice. One for the result of the whole recursion, and another for the accumulative argument.

IO swapping gives an evidence of this observation. From IO swapping, we get the following `cycle'` function as an IO-swapped form of `cycle`:

```
cycle' x = let ([], h, r') = cyc' x h in r'
  where  cyc' [] r = ([], r, r)
         cyc' (b:y) r = let (a:x, h, r') = cyc' y (a:r)
                        in (x, h, r')
```

where it has no global circularity concerned with the result of the whole recursion `r'`. We try fusing `map (1+)` as follows.

```
map (1+) (cycle' x) ⇒ let ([], h, r') = cyc' x h in map (1+) r'
                    ⇒ let ([], h, r') = id_id_mapS (cyc' x h) in r'
                      where id_id_mapS (a,b,c) = (a,b, map (1+) c)
```

Using Theorem 3.3.1, we get the following result where `mcyc' x` is equivalent to `map (1+) (cycle' x)`.

```
mcyc' x = let ([], h, r') = aux x h in r'
  where  aux [] r = ([], r, map (1+) r)
         aux (b:y) r = let (a:x, h, r') = aux y (a:r)
                       in (x, h, r')
```

The function `map (1+)` reaches to the bottom of the recursion. To continue the fusion transformation it seems that we should promote `map (1+)` into the accumulative argument `r`. But here we meet with a stick again. We cannot promote `map` into the accumulative argument `r`. The value of `r` is used twice and changing it affects the value of second element of the result. It is completely the same problem with that of circularity of `cycle`. IO swapping cannot remove non-linear use of variables though it removes circularities. In summary, the main difficulty to manipulate `cycle` is a non-linear use of a value and IO swapping does not help us.

Unwillingly, we try another approach. What we should do is to remove the non-linear use of values. We achieve it by duplicating the value as follows.

```
cycle2 x = let (r1,r2) = cyc2 x r2 in r1
    where cyc2 [] r = (r,r)
          cyc2 (a:x) r = let (r1,r2) = cyc2 x r
                         in (a:r1,a:r2)
```

The function `cycle2` computes the same value as `cycle` and it has no non-linear use of value at the top of the recursion. For `cycle2`, we can promote a function into it as similar to `repminl`, using fold promotion theorem.

Lemma 7.2.1.
The following two functions `f1` and `f2` are equivalent for all `phi`, `g1`, `g2`, `g3`, `g4`, `g5`, and `x`.

```
f1 g1 g2 g3 g4 g5 x = let r = f1' x (g4 (phi r) r) in g5 (phi r) r
  where
    f1' [] h = g3 h
    f1' (a:x) h = let r = f1' x (g2 a r h)
                  in g1 a r h

f2 g1 g2 g3 g4 g5 x = let (r1, r2) = f2' x (g4 r1 r2) in g5 r1 r2
  where
    f2' [] h = (phi(g3 h), g3 h)
    f2' (a:x) h = let (r1, r2) = f2' x (g2 a r2 h)
                  in (g1' a r1 h, g1 a r2 h)
```

provided that for all `a`, `r` and `h`, `phi (g1 a r h)` = `g1' a (phi r) h`.

Proof.
Define the following function `f3`.

```
f3 g1 g2 g3 g4 g5 x = let (r1, r2) = f3' x (g4 (phi r1) r2)
                      in g5 (phi r1) r2
  where
    f3' [] h = (g3 h, g3 h)
    f3' (a:x) h = let (r1, r2) = f3' x (g2 a r2 h)
                  in (g1 a r1 h, g1 a r2 h)
```

From their definition `f3 g1 g2 g3 g4 g5 x` is equivalent with `f2 g1 g2 g3 g4 g5 x` for all `g1`, `g2`, `g3`, `g4`, and `x`, because throughout the recursions, the auxiliary function `f3'` of `f3` has the same values in the first element and the second element of the result.

Now we start calculation as follows:

```
f1 g1 g2 g3 g4 g5 x
    ⇒ f3 g1 g2 g3 g4 g5 x
    ⇒ let (r1, r2) = f3' x (g4 (phi r1) r2) in g5 (phi r1) r2
    ⇒ let (r1, r2) = phi_id (f3' x (g4 r1 r2)) in g5 r1 r2
            where phi_id (r1,r2) = (phi r1, r2)
```

We will fuse `phi_id` with `circ2'` using Theorem 3.3.1, for we can express `f3'` in terms of `foldr` as follows.

```
f3' x = foldr (\a r h-> let (r1,r2) = r (g2 a r2 h) in (g1 a r1 h, g1 a r2 h))
              (\h-> (g3 h, g3 h)) x
```

We achieve the fusion by checking the following conditions.

```
phi_id · (\h-> (g3 h, g3 h)) = (\h-> (phi (g3 h), g3 h))
phi_id · ((\a r h-> let (r1,r2) = r (g2 a r2 h) in (g1 a r1 h, g1 a r2 h)) a r)
  ⇒ (\h-> let (r1,r2) = r (g2 a r2 h) in (phi (g1 a r1 h), g1 a r2 h))
  ⇒ (\h-> let (r1,r2) = (phi_id·r) (g2 a r2 h) in (g1' a r1 h, g1 a r2 h))
  ⇒ (\a r h-> let (r1,r2) = r (g2 a r2 h) in (g1' a r1 h, g1 a r2 h)) a (phi_id·r)
```

Then we get the following program.

```
f1 g1 g2 g3 g4 g5 x = let (r1, r2) = aux x (g4 r1 r2) in g5 r1 r2
 where
   aux x = foldr (\a r h-> let (r1,r2) = r (g2 a r2 h) in (g1' a r1 h, g1 a r2 h))
                 (\h-> (phi (g3 h), g3 h)) x
```

The result is actually `f2` in the proposition above, after unfolding `foldr`. □

But Lemma 7.2.1 is not enough. It derives the following program for `mcyc = map (1+)(cycle x)`.

```
mcyc x = let (r1,r2) = aux x r2 in r1
    where aux [] r = (map (1+) r,r)
          aux (a:x) r = let (r1,r2) = aux x r
                        in ((a+1):r1,a:r2)
```

Now the base case of `aux` has a non-linear use of a value, denoted by the variable `r`, that prevents the further manipulation. It is the IO-swapped version of the non-linear use of a value in `cycle`. In `cycle` results are used twice, and in `mcyc` arguments are used twice. IO swapping with Lemma 7.2.1 solves this problem, as similar to Section 6.3.

Lemma 7.2.2.
The following two functions `f4_` and `f5_` are equivalent for all `psi`, `g1`, `g2`, `g3`, `g4`, and `x`.

```
f4_ g1 g2 g3 g4 x = let ([],h,r') = f4_' x (g3 (psi h) h) in r'
    where f4_' [] r = (x, g4 r, r)
          f4_' (b:y) r = let (a:x,h,r') = f4' y (g1 a r h)
                         in (x, g2 a r h, r')

f5_ g1 g2 g3 g4 x = let ([],(h1,h2),r') = f5_' x (g3 h1 h2) in r'
    where f5_' [] r = (x, (psi(g4 r), g4 r), r)
          f5_' (b:y) r = let (a:x,(h1,h2),r') = f4' b (g1 a r h2)
                         in (x, (g2' a r h1, g2 a r h2), r')
```

provided that for all `a`, `r` and `h`, `psi (g2 a r h)` = `g2' a r (psi h)`.

Proof. We start calculation as follows.

```
f4_ g1 g2 g3 g4 x
   ⇒ let ([],h,r') = f4_' x (g3 (psi h) h) in r'
   ⇒ let ([],h,r') = f4_' x (g3 (id_psi_id ([],h,r')) h) in r'
          where id_psi_id (x,h,r) = (x, psi h, r)
```

Then, we get the following program from Lemma 7.2.1:

```
f4_ g1 g2 g3 g4 x = let (([],h1,r1'),([],h2,r2')) = aux x (g3 h1 h2) in r2'
  where
     aux [] r = (id_psi_id (x, g4 r, r), (x, g4 r, r))
     aux (b:y) r = let ((a1:x1,h1,r1'),(a2:x2,h2,r2')) = f4' b (g1 a2 r2 h2)
                   in ((x1, g2' a1 r h1, r1'),(x2, g2 a2 r h2, r2'))
```

Removing unnecessary variables and swapping the order of the elements inside the tuple, we get the program equivalent to `f5`. □

Lemma 7.2.3.
The following two functions `f4` and `f5` are equivalent for all `psi`, `g1`, `g2`, `g3`, `g4`, and `x`.

```
f4 g1 g2 g3 g4 x = let r = f4' x (g4 r) in r
    where f4' [] h = g3 (psi  h) h
          f4' (a:x) h = let r = f4' x (g2 a r h)
                        in g1 a r h

f5 g1 g2 g3 g4 x = let r = f5' x (psi (g4 r), g4 r) in r
    where f5' [] (h1, h2) = g3 h1 h2
          f5' (a:x) (h1, h2) = let r = f5' x (g2' a r h1, g2 a r h2)
                               in g1 a r h2
```

provided that for all `a`, `r` and `h`, `psi (g2 a r h)` = `g2' a r (psi h)`.

Proof. Applying IO swapping backward to `f4_` and `f5_` these are defined in Lemma 7.2.2, then we get `f4` and `f5` respectively. □

Using Lemma 7.2.3, we can get the following program for `mcyc2 = (map (1+))·cycle`.

```
mcyc2 x = let (r1,r2) = aux x (map (1+) r2, r2) in r1
    where aux [] (h1,h2) = (h1,h2)
          aux (a:x) (h1,h2) = let (r1,r2) = aux x (h1,h2)
                              in ((a+1):r1,a:r2)
```

Here we encounter another non-linear use of values and we might need to duplicate `r2` once more to remove it. However, we do not need to do so. Recall that `r1 = map (1+) r2`, then we can remove `map` with unnecessary variables as follows.

```
mcyc3 x = let r1 = aux x r1 in r1
    where aux [] h1 = h1
          aux (a:x) h1 = let r1 = aux x h1
                         in (a+1):r1
```

We have succeeded in fusion. All intermediate data structures are removed.

We summarize the result in the following theorem. Consider the following higher order function `circ`:

```
circ g1 g2 g3 g4 x = let r = circ' x (g4 r) in r
  where
    circ' [] h = g3 h
    circ' (a:x) h = let r = circ' x (g2 a r h)
                    in g1 a r h
```

where the function `circ` is a quite general circular function scanning over a list. We can derive a fusion law for the function `circ` from Lemma 7.2.1 and Lemma 7.2.3. It is a novel result as far as the best of our knowledge.

Theorem 7.2.4 (Fusion for Circular Functions)**.**
`phi (circ g1 g2 g3 g4 x)` = `circ g1' g2' g3' g4' x` provided that for all `a`, `r` and `h`, the following equations hold.

```
phi (g1 a r h) = g1' a (phi r) (psi h)
psi (g2 a r h) = g2' a (phi r) (psi h)
phi (g3 h)     = g3' (psi h)
psi (g4 r)     = g4' (phi r)
```

Proof. We define `f x = phi (circ g1 g2 g3 g4 x)`. From Lemma 7.2.1, we get the following program.

```
f x = let (r1, r2) = aux x (g4 r2) in r1
    where aux [] h = (phi(g3 h), g3 h)
          aux (a:x) h = let (r1, r2) = aux x (g2 a r2 h)
                        in (g1' a r1 (psi h), g1 a r2 h)
```

Remember that throughout the whole recursion, the value of `r1` is equivalent to `phi r2`. From assumption `phi (g3 h)` is equivalent to `g3' (psi h)`. Now we can apply Lemma 7.2.3 and get the following program.

```
f x = let (r1, r2) = aux x (r1, psi (g4 r2)) in r1
  where
    aux [] (h1,h2) = (g3' h1, g3 h2)
    aux (a:x) (h1,h2) = let (r1, r2) = aux x (g2' a (phi r2) h1, g2 a r2 h2)
                        in (g1' a r1 (psi h2), g1 a r2 h2)
```

From assumption `psi (g4 r2)` is equivalent to `g4' (phi r2)`. We will remove all `psi` and `phi`, because `r1` is equivalent to `phi r2` and `h1` is equivalent to `psi h2` throughout the whole recursion.

```
f x = let (r1, r2) = aux x (r1, g4' r1) in r1
  where
    aux [] (h1,h2) = (g3' h1, g3 h2)
    aux (a:x) (h1,h2) = let (r1, r2) = aux x (g2' a r1 h1, g2 a r2 h2)
                        in (g1' a r1 h1, g1 a r2 h2)
```

Finally we remove unnecessary variables. The second element of the result and the second element of the accumulative argument is useless anymore. After removing them we get the program that is actually `circ g1' g2' g3' g4' x`. □

Note that we can easily prove Theorem 7.2.4 based on *free theorem* [Wad89]. It implies that this result is not completely novel.

What IO swapping gives is a guideline for manipulating circular programs. The characteristic of circular programs is dependency from a result to an argument. IO swapping rewrites this characteristic as "dependency from an argument to a result", which usually appears in accumulative programs. This implies that, if we can appropriately manipulate accumulative programs, IO swapping gives a way to manipulate circular programs. And IO swapping also gives manipulations of accumulations from manipulations of results. But the difficulties of circular programs often comes from non-linear use of values. IO swapping can do nothing about it. Therefore we need to work more to give a sufficient method to manipulate circular programs.

Chapter 8

Conclusion

In this thesis, 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 derivations and manipulations of TABA programs. We derived IO-swapped manipulations from primitive cases, such as a fusion method for accumulative programs and a higher-order removal method that copes with functional accumulation, and discussed its extension to manipulations 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* [MFP91][Fok92], where programs are abstracted using the theory of categories in mathematics. The framework of constructive algorithmics is very powerful so that many fusion methods introduced in Chapter 3 are actually described [MFP91][Mei92][TM95][HITT97]. 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.

Acknowledgements

Many people contributed to this thesis. I would like to give my thanks to them.

First of all, I express my great gratitude to my supervisor, Prof. Masato Takeichi. He always gave me prudent advice and lighted me to the right way of research. I would not have been able to achieve my work without him.

I am very grateful to Prof. Zhenjiang Hu, Dr. Kazuhiko Kakehi, and Prof. Olivier Danvy. Professor Hu instructed me in the art of calculational programming with its profound appreciation, and moreover, he supported me by innumerable inspiring discussions. Dr. Kakehi gave a lesson on attribute grammars and its composition method; besides he gave many pieces of beneficial advice to my research and its presentation. Professor Danvy introduced to us the TABA work and its relation to defunctionalization and CPS transformation, and he also cared about the progress of our work and encouraged us. The basis of my research consists of their instructions.

I am thankful to Mr. Kiminori Matsuzaki, Mr. Tetsuo Yokoyama, Dr. Shin-Cheng Mu, and Mr. Keisuke Nakano for their insightful discussions that help to improve my research. Mr. Matsuzaki gave me an insight about the manipulation of non-linear recursions. Mr. Yokoyama taught me the comparison of the existing calculational program transformations and their automation. Dr. Mu showed me another derivation of a TABA program that indicated the characteristics of TABA programs and IO swapping. Mr. Nakano introduced to me a relationship between attribute grammars and functional programs with many existing researches.

I thank all members of Takeichi laboratory. I every day discussed with them, laughed with them, studied with them, played with them, learned from them, chattered with them, researched with them, and enjoyed with them. Such everything aided and supported me.

Finally I would like to express my thanks again to everyone helped me. Thank you very much.

References

[Bac02] Kevin Backhouse. A functional semantics of attribute grammars. In *Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'02*, pages 142–157, London, UK, 2002. Springer-Verlag.

[Bar84] Hendrik Pieter Barendregt. *The lambda calculus: its syntax and semantics*, volume 103 of *Studies in logic and the foundations of mathematics*. North-Holland, 1984.

[BD77] Rod M. Burstall and John Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, 1977.

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

[Bir84a] Richard Bird. The promotion and accumulation strategies in transformational programming. *ACM Transactions on Programming Languages and Systems*, 6(4):487–504, 1984.

[Bir84b] 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.

[Boi92] Eerke A. Boiten. Improving recursive functions by inverting the order of evaluation. *Science of Computer Programming*, 18(2):139–179, 1992.

[CDPR99] Loic Correnson, Etienne Duris, Didier Parigot, and Gilles Roussel. Declarative program transformation: A deforestation case-study. In *Proceedings of the 1st International Conference on Principles and Practice of Declarative Programming, PPDP'99*, pages 360–377, 1999.

[Chi93] Wei-Ngan Chin. Towards an automated tupling strategy. In *Proceedings of the ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation, PEPM'93*, pages 119–132, New York, NY, USA, 1993. ACM Press.

[Chi99] Olaf Chitil. Type inference builds a short cut to deforestation. In *Proceedings of the 4th ACM SIGPLAN International Conference on Functional Programming, ICFP'99, Paris, France*, volume 34(9), pages 249–260. ACM Press, New York, 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.

[DG05] Olivier Danvy and Mayer Goldberg. There and back again. *Fundamenta Informaticae*, 66(4):397–413, 2005.

[DM93] Pierre Deransart and Jan Małuszyński. *A grammatical view of logic programming*. MIT Press, Cambridge, MA, USA, 1993.

[dMS01] Oege de Moor and Ganesh Sittampalam. Higher-order matching for program transformation. *Theoretical Computer Science*, 269(1–2):135–162, 2001.

[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.

[Fok89] Maarten M. Fokkinga. Tupling and mutumorphisms. In *Squiggolist*, volume 1, 1989.

[Fok92] Maarten M. Fokkinga. *Law and Order in Algorithmics*. PhD thesis, University of Twente, Dept INF, Enschede, The Netherlands, 1992.

[GG84] Harald Ganzinger and Robert Giegerich. Attribute coupled grammars. In *Proceedings of the of the 1984 SIGPLAN symposium on Compiler construction*, pages 157–170. ACM Press, 1984.

[GHC] The Glasgow Haskell Compiler. Available from `http://www.haskell.org/ghc/`.

[Gil96] Andrew Gill. *Cheap Deforestation for Non-strict Functional Languages*. PhD thesis, Department of Computing Science, Glasgow University, 1996.

[GLPJ93] Andrew Gill, John Launchbury, and Simon L. Peyton Jones. A short cut to deforestation. In *Proceedings of the 6th ACM SIGPLAN/SIGARCH International Conference on Functional Programming Languages and Computer Architecture, FPCA'93*, pages 223–232. ACM Press, New York, 1993.

[Has02] *The Haskell 98 Report*, 2002. Available from `http://www.haskell.org/definition/`.

[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.

[HIT99] Zhenjiang Hu, Hideya Iwasaki, and Masato Takeichi. Calculating accumulations. *New Generation Computing*, 17(2):153–173, 1999.

[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.

[HL78] Gérard P. Huet and Bernard Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, 11:31–55, 1978.

[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.

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

[KS86] Matthijs F. Kuiper and S. Doaitse Swierstra. Using attribute grammars to derive efficient functional programs. Technical Report RUU–CS–86–16, Institute of Information and Computing Sciences, Utrecht University, 1986.

[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.

[Mei92] Erik Meijer. *Calculating Compilers*. PhD thesis, Nijmegen University, 1992.

[MFP91] Erik Meijer, Maarten Fokkinga, and Ross Paterson. 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, Cambridge, MA, USA*, volume 523, pages 124–144. Springer-Verlag, Berlin, 1991.

[MKHT05a] Akimasa Morihata, Kazuhiko Kakehi, Zhenjiang Hu, and Masato Takeichi. IO swapping leads you there and back again. In *Proceedings of the 7th Generative Programming and Component Engineering Young Researchers Workshop, GPCE-YRW'05*, pages 7–13, 2005. Extended abstract of *Technical Report METR* 2005-11 [MKHT05b].

[MKHT05b] Akimasa Morihata, Kazuhiko Kakehi, Zhenjiang Hu, and Masato Takeichi. Reversing iterations: IO swapping leads you there and back again. *Technical Report METR* 2005-11, Department of Mathematical Informatics, University of Tokyo, 2005.

[Nis03] Susumu Nishimura. Correctness of a higher-order removal transformation through a relational reasoning. In *Proceedings of the First Asian Symposium on Programming Languages and Systems, APLAS'03*, volume 2895 of *LNCS*, 2003.

[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.

[Rey98] John C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, 1998. Reprinted from the proceedings of the 25th ACM National Conference (1972), with a foreword.

[Sar99] João Saraiva. *Purely Functional Implementation of Attribute Grammars*. PhD thesis, Department of Computing Science, Utrecht University, 1999.

[SF93] Tim Sheard and Leonidas Fegaras. A fold for all seasons. In *Proceedings of the 6th ACM SIGPLAN/SIGARCH International Conference on Functional Programming Languages and Computer Architecture, FPCA'93, Copenhagen, Denmark*, pages 233–242. ACM Press, New York, 1993.

[TM95] Akihiko Takano and Erik Meijer. Shortcut deforestation in calculational form. 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 306–313. ACM Press, New York, 1995.

[Voi04] Janis Voigtländer. Using circular programs to deforest in accumulating parameters. *Higher-Order and Symbolic Computation*, 17(1-2):129–163, 2004.

[Wad88] Philip Wadler. Deforestation: Transforming programs to eliminate trees. In *Proceedings of the 2nd European Symposium on Programming, ESOP'88*, volume 300 of *LNCS*, pages 344–358. Berlin: Springer-Verlag, 1988.

[Wad89] Philip Wadler. Theorems for free! In *Proceedings of the 4th International Conference on Functional Programming Languages and Computer Architecture, FPCA'89, London, UK*, pages 347–359. ACM Press, New York, 1989.

[YHT05] Tetsuo Yokoyama, Zhenjiang Hu, and Masato Takeichi. Calculation rules for warming-up in fusion transformation. In *Proceedings of 6th Symposium on Trends in Functional Programming, TFP'05*, 2005. To appear.

[Yok06] Tetsuo Yokoyama. *Deterministic Higher-order Matching for Program Transformation*. PhD thesis, The University of Tokyo, 2006. To appear.