A Calculational Approach to Optimizing Functional Programs

(9=@.E*<jK!$K$h$k4X?t%W%m%0%i%`$N:GE,2=)

Zhenjiang Hu

(8U ?69>)

Department of Information Engineering
University of Tokyo

A thesis submitted in partial fulfilment of
the requirements for the degree of Doctor of Engineering
in Information Engineering at the University of Tokyo

September 1996

Committee

Masato Takeichi	Professor
Hidehiko Tanaka	Professor
Hirochika Inoue	Professor
Kokiichi Sugihara	Professor
Masami Hagiya	Professor
Satoshi Matsuoka	Associate Professor
Minoru Terada	Associate Professor

Acknowledgements

First of all, I would like to thank my supervisor, Masato Takeichi, for his constant supporting, encouragement, and guidance, which have been invaluable to the research in this thesis and beyond. I feel very lucky that I have been able to work closely with him for the past three years.

I also feel deeply indebted to Hideya Iwasaki for his great encouragement, understanding, and advise. He has usually kept patient to hear what I have been working on, and has given me immense help in making my ideas more precise and my writing more succinct.

I benefited a great deal from stimulating discussions with Akihiko Takano, Oege de Moor, Fer Jan de Vries, Satoshi Matsuoka, Tetsurou Tanaka, Liangwei Xu. They help me understanding important issues related to this research. In particular, Takano's comments have always helped push this work forward.

Thanks also goto Yohji Akama, Wei-Ngan Chin, Masami Hagiya, Simon Peyton Jones, Eric Meijer, Mizuhito Ogawa, Peter Orbaek, Rose Paterson, Tim Sheard, for discussing about related research through conferences and seminars.

Many friends at Takeichi Laboratory in the University of Tokyo and old friends from the Department of Computer Science of Engineering in Shanghai Jiao Tong University have helped made my years enjoyable. I would not be myself without their friendship.

I acknowledge the financial support I received from Japanese Government Scholarship, Japanese Research Promotion Society, the International Communication Foundation of KDD, and the Telecommunications Advancement Foundation of NTT, which allowed me to carry out the research described in this thesis.

I couldn't forget to thank Yongqiang Sun, who first introduced me to functional programming in Shanghai Jiao Tong University, and supervised me both as an undergraduate and Master student.

Finally, my gratitude and love are to my wife Wurong for her support, patience, and love.

Zhenjiang Hu
September 1996

Contents

Chapter 1

Introduction

There is a well-known Chinese proverb: `5{OB7′>8IT2DF1;~7sF@` (one cannot hope to have both fish and bear palm at the same time), saying that it is hardly to obtain two treasures simultaneously. Same thing happens in our programming: clarity is and is not next to goodness. Clearly written programs have the desirable properties of being easier to understand, show correct, and modify, but it can also be hellishly inefficient.

This is no accident. A major design technique for achieving clarity is *modularity*: breaking a problem into independent components. But modularity can often lead to inefficiency, because of the overhead of communication between components, and because it preludes potential optimizations across components boundaries.

We aim to have both `5{` (fish) and `7′>8` (bear palm), but at different time: using compositional style of *functional programming* to write clean and correct (but probably inefficient) programs, and using *program calculation* technique to convert them to more efficient equivalents.

Functional language is simply the doctrine "side effects should be avoided" carried to its extreme: once bounded, the value of a variable is never changed. Examples includes pure Lisp, FP, ML, Miranda, Gofer, Haskell etc. The special characteristics and advantages of functional programming are two folds. First, it is good for writing clear and modular programs because it supports a powerful and elegant programming style. As Hughes [Hug85] pointed out, functional programming offers important advantages for software development. Second, it is good for performing transformation because of its nice mathematical properties.

Despite widespread interest in functional languages, there are still some objections to their overall performance. This is because functional languages have a higher runtime execution cost than imperative languages, requiring more processing power and memory.

Consider the following problem: "Find the sum of the squares of the numbers from 1 to n." A functional program, written in Gofer [Jon91], that does this is:

$$\mathit{sum}\,(\mathit{map}\ \mathit{square}\,(\mathit{upto}\,(\mathit{1}\,,n))).$$

This generates a list of numbers from 1 to n, forms the second list by squaring each number in the first list, and returns the sum of the numbers in the second list. The structure of this program corresponds well to the structure of the problem as expressed in English: "the sum of (the square of (the numbers from 1 to n))." In contrast, consider the following imperative program for the same problem:

```
s := 0;
for i := 1 to n do s := s + i × i;
return(s)
```

This bears little resemblance to the structure of the problem in English, and hence less clear than the functional version. It "mashes together" the three separate components. But it is much more efficient.

The clarity of the functional program comes from dividing into three components: generate the numbers, square the numbers, and find the sum. This in turn depends on the use of intermediate lists to communicate between these components: *upto* $(1, n)$ passes the list $[1, 2, \cdots, n]$ to *map square*, which passes the list $[1, 4, \cdots, n^2]$ to *sum*.

Generally, functional programming constructs a complex program by gluing components which are relatively simple, easier to write, and potentially more reusable. However, some data structures, which are constructed in one component and consumed in another but never appear in the result of the whole program, give rise to the problem of efficiency. Other inefficiencies due to, for example, multiple traversals of the same data structure or redundant recursive calls, will be discussed later. In this thesis, we aim to show how these inefficiencies could be effectively removed and how functional programs can be improved/optimized by means of *calculation* with programs (i.e., program calculation).

1.1 Program Calculation

One methodology that offers some scope for making the construction of *efficient* programs more mathematical is that of *transformational programming* [BD77, Fea87, Dar81]. Using the transformational approach to programming does not attempt to produce directly a program that is correct, understandable and efficient, rather one initially concentrates on producing a program which is as clear and understandable as possible ignoring any question of efficiency. Having satisfied himself that he has a correct program he successively transforms it to more and more sufficient versions using methods guaranteed to preserve the meaning of the program. In essence, the value of the approach is in its separation of the concerns of clarity and correctness, and of efficiency and practical implementation.

Obviously, functional programming is well suitable for the program transformation methodology, as it is powerful to describe clear program and is good for program transformation with its mice mathematical properties.

Program calculation, i.e., calculation with programs, is a kind of program transformation based on the theory of *Constructive Algorithmics* [Bir87, Bir89, Bac89, Mal90, MFP91, Fok92a, dM92, Jeu93]. It proceeds by means of manipulating of programs based on a rich collection of identities and transformation laws. It resembles the manipulating of formulas as in high school algebra: a formula F is broken up into its semantic relevant constituents and the pieces are assembled together into a different but semantically equivalent formula F', thus yielding the equality $F = F'$. The following example shows a calculation of the solution of x for the equation $x^2 - c^2 = 0$.

$$\begin{array}{ll}
 & x^2 - c^2 = 0 \\
\equiv & \quad \{ \text{ by identity: } a^2 - b^2 = (a-b)(a+b) \ \} \\
 & (x-c)(x+c) = 0 \\
\equiv & \quad \{ \text{ by law: } ab = 0 \Rightarrow a = 0 \text{ or } b = 0 \ \} \\
 & x - c = 0 \text{ or } x + c = 0 \\
\equiv & \quad \{ \text{ by law: } a = b \Rightarrow a \pm d = b \pm d \ \} \\
 & x = c \text{ or } x = -c
\end{array}$$

Here we calculate x rather than guess or "just invent," based on some identities and laws (rules). Particularly, we make use of the transformation law that a higher order equation should be factored into several first order ones whose solution can be easily obtained.

Constructive Algorithmics, also known as *Bird-Meertens Formalism*, is a transformation programming *calculus*, giving the theory for constructing *an algebra of programs* in a systematic way and putting emphasis on its means to express programs concisely. The word *algebra* is used in this context in two meanings. In its technical meanings, an algebra is a collection of operations together with a set on which the operation act. In a non-technical sense, algebra is the art of manipulation of formulas as in high school algebra as shown above.

In Constructive Algorithmics, the calculation are based on calculation laws (i.e., rules) that describe properties of programs. Theorems may be used to tell bigger steps in a calculation in which there is ample opportunity for machine assistance. Basically, these laws and theorems are built upon the algebra of programs, a collection of identities. These identities can be provided by exploiting the algebraic structure of the data concerned. In particular, there is a close correspondence between data structures (terms in an algebra) and control structures (*catamorphisms* over that algebra). This correspondence is the same as that between the manipulation of types and of functions by categorical functors.

This thesis is mainly concerned with how to apply the theory and techniques of Constructive Algorithmics to the optimization of functional programs.

1.2 Contributions of the Thesis

The main contribution of this thesis is that a new method, based on the theory of Constructive Algorithmics, has been proposed for the systematic and practical study on optimization of functional programs. We shall explain it in a more detailed way as below.

Optimization of General Functional Programs

In functional programming, a program *prog* is usually expressed as compositions of transformations over data structures while each transformation may be defined by a recursion $\mathcal{R}_i$ used to traverse over its input data structure for computing the result, namely

$$prog = \mathcal{R}_1 \circ \cdots \circ \mathcal{R}_n.$$

This compositional style of programming allows clearer and more modular programs, but comes at a price of possibly high runtime overhead resulting mainly from the following two categories:

- Unnecessary intermediate data structures passed between the composition of two recursions;
- Inefficiency in a single recursion, such as redundant recursive calls, multiple traversals of the same data structures, and unnecessary traversals of intermediate data structures (see Section 5.1).

We aim to eliminate these efficiencies in a calculational way. We start by proposing a *novel* algorithm ([HIT96d], Chapter 3) which automatically turns almost all practical recursive definitions into hylomorphisms and paves the way for the use of the theory of Constructive Algorithmics. Hylomorphism, one of the most important concept in the theory of Constructive Algorithms, actually plays a significant role in our calculation for improving functional programs.

We deal with the first kind of inefficiency due to unnecessary intermediate data structures by fusion transformation, an optimizing process whereby these small programs are fused into a single one and intermediate data structures are removed. We put forward an algorithm ([HIT96d], Chapter 4) for structuring hylomorphisms so that the Hylo Fusion and Acid Rain Theorems (Section 2.2) can be effectively applied. Particularly, we give a new theorem for deriving the polymorphic functions (Section 4.2.2), namely τ and σ, for the efficient use of the Acid Rain Theorem.

For the second kind of inefficiency occurred in recursions, a new idea called *cheap tupling* transformation ([HIT96b], Chapter 5) is proposed. We identify the importance of the relationship between tupling transformation and *structural* mutual recursions (not a simple mutual recursion) (Section 5.2), based on which three simple and concise calculational rules (Theorem 12, 16 and 18) are derived. These rules can be applied to improve a wide class of recursions (though not all) through the elimination of redundant recursive calls, multiple traversals of the same data structures, and unnecessary traversals of data structures which has not been noticed before. Furthermore, our cheap tupling is suitable to coexist with our fusion transformation. The combination of the two transformation rules is direct and natural (Section 5.3) in the calculational setting. In contrast, the previous study on this combination [Chi95] is much more difficult because of complicated control of infinite unfoldings in the case where fusion and tupling are applied simultaneously.

Another aspect for improving recursions is the well-known technique called *accumulation* making use of intermediate result with an accumulation parameter. In contrast to the previous informal study, we formulate accumulations as higher order catamorphisms facilitating program calculation, and provide several general rules for calculating accumulations, i.e., synthesizing and manipulating accumulations ([HIT96a], Chapter 6). We show how to apply the accumulation strategy to a composition of functions. In particular, it helps if the composed accumulating parameters are functions; this leads to a continuation-passing style of programming.

Optimization of Functional Programs with External Effects

Recently, a lot of studies have been devoted to using *monads*, a particular abstract data type, as a tool for structural functional programming with external effects. Monads provide a uniform framework for describing a wide range of programming language features (external effects) such as states, input/output, non-determinism and so on. To facilitate monadic program transformation, Fokkinga derived an assumption under which there is a kind of so-called monadic catamorphisms which satisfy several general laws useful for the transformation of monadic programs. However, his theory is too restrictive to be applied practically. To remedy this situation, we build up a new theory ([HI95], Chapter 7) for monadic catamorphism by moving Fokkinga's assumption on the monad to the condition on a map between monadic algebras so that our theory is valid for arbitrary monads including, for example, the state monad that is not allowed in Fokkinga's theory. Our theory covers Fokkinga's as a special case but can be applied to a larger class of monadic programs. Many examples will be used to illustrate our idea.

Optimization of Parallel Functional Programs

Functional programming is claimed to be more suitable for parallel implementation than imperative programming. We show that by constructing efficient catamorphisms which have high inherited parallelism one can succeed in deriving efficient parallel functional programs. Rather than giving a general study, we focus ourselves on the construction of a special catamorphism, called *list homomorphism*. We propose a systematic way ([HIT96c, HIT96e], Chapter 8) to embed an algorithm into list homomorphisms so that parallel programs are derived. We show, with an example, how a simple, and "obviously" correct, but possibly inefficient solution to the problem can be successfully turned into a semantically equivalent "almost" homomorphism by means of two transformations: tupling and fusion.

1.3 Organization of the Thesis

The organization of the thesis is as follows. After explaining the motivation in Chapter 1, we begin by reviewing in Chapter 2 the basic concepts of functional programming and the previous work on the Constructive Algorithmics, which provides the basis of this thesis.

Then, in Chapter 3, we give our algorithm which can automatically turn almost all practical recursive definitions into hylomorphisms, a general recursive form covering all those in [MFP91, SF93, TM95] such as catamorphisms and anamorphisms (see Theorem 5). This paves the way for our later optimization of functional programs by calculation.

Next, we explain our ideas of how to optimize functional programs, being either general or special, via program calculation. In Chapter 4 and Chapter 5, we focus on the fusion and the tupling transformation. And in Chapter 6, we formulate accumulations with higher order catamorphisms so that we can manipulate them for the improvement of functional programs. Calculation of monadic and parallel functional programs are discussed in Chapter 7 and Chapter 8 respectively.

Finally, we make the concluding remarks and discuss the future work in Chapter 9.

Chapter 2

Preliminaries

In this Chapter, we shall review the basic concepts of functional programming and the previous work on the Constructive Algorithmics, which will be used in the rest of this thesis.

2.1 Functional Programming

Programming in a functional language, such as FP [Bac78], ML [MTH90], Miranda[1] [Tur85], Gofer [Jon91], Haskell [HW90], consists of building definitions and using the computer to evaluate expressions. The primary role of the programmer is to construct a function to solve a given problem. This function, which may involve a number of subsidiary functions, is expressed in a notation that obeys normal mathematical principles.

A characteristic feature of functional programming is that if an expression possesses a well-defined value, then the order in which a computer may carry out the evaluation does not affect the outcome. In other words, the meaning of an expression is its value and the task of the computer is simply to obtain it. It follows that the expressions in a functional language can be constructed, manipulated, and reasoned about, like any other kind of mathematical expression, using more or less familiar algebraic laws.

In the following, we introduce the basic concepts of functional programming. For more detailed discussion, refer to the excellent textbook written by Bird and Wadler [BW88]. Even though we don't restrict ourselves to a specific functional language, almost all programs in this thesis can be run well under the Gofer system [Jon91], a popular functional language system.

[1]Miranda is a trademark of Research Software Limited.

2.1.1 Expressions and Values

The concept of expressions is central in functional programming. Its most important feature is that an expression is used solely to describe (to denote) a *value*. In other words, the meaning of an expression is value and there are no other effects, hidden or otherwise, in any procedure for actually obtaining it. Furthermore, the value of an expression depends only on the values of its constituent expressions (if any) and their subexpressions may be replaced freely by others possessing the same value.

The term *reduction* is used to describe this process in which an expression is evaluated by reducing it to its "simplest equivalent form."

2.1.2 Types

The universe of values is partitioned into organized collection, called *type*. It is an important principle that every well-formed expression can be assigned a type that can be deduced from the constituents of the expression alone. This principle is called *strong-typing*. The major consequence of the discipline imposed by strong-typing is that any expression which cannot be assigned a type is regarded as not being well-formed and is rejected for being evaluated. Such expressions are simply regarded as illegal.

2.1.3 Functions and Definitions

The most important kind of value in functional programming is a function value. Mathematically speaking, a function f is a value of correspondence which associates with each element of the *source* type S a unique member of the *target* type T, as usually expressed by writing:

$$f : S \to T.$$

The function f is said to take *arguments* (or *parameters*) in S and return *results* in T. If x denotes an element of S, then we write $f(x)$, or just $f\,x$, to denote the result *applying* the function f to x.

Efficiency of definition

It is important, especially in this thesis, to keep in mind the distinction between a function value and a particular definition of it. There are many possible definitions for one and the same function. For instance, we can define the function which doubles its argument in the following two ways:

$$\begin{array}{lcl} \mathit{double}\ x & = & x + x \\ \mathit{double}'\ x & = & 2 \times x. \end{array}$$

The two definitions describe different procedures for obtaining the correspondence, but *double* and *double′* denote the same function and we can assert that

$$double = double'$$

as a mathematical truth. Regarded as procedures for evaluation, one definition may be more or less *efficient* than the other, in the sense that the evaluator may be able to reduce expressions of the form *double x* more or less quickly than expressions of the form *double′*. However, the notion of efficiency is not one which can be attached to function values themselves. Indeed, it depends on the given form of the definition and the precise characteristics of the evaluation mechanism. In this thesis, we adopt *lazy evaluation* mechanism [Jon88, JL92] widely used as in lazy functional languages such as Haskell and Gofer.

Forms of definition

In many situations, we may want to define the value of a function by case analysis. Consider the function *min*:

$$\begin{array}{rcll} min\ x\ y & = & x, & \text{if } x \leq y \\ & = & y, & \text{if } x > y \end{array}$$

This definition consists of two expressions, each of which is distinguished by boolean-valued expressions, called *guard*. The first alternative of the definition says that the value of $min\ x\ y$ is defined to be x, provided that the expression $x \leq y$. The second alternative says that the value of $min\ x\ y$ is defined to be y, provided that the expression $x > y$.

There are many other ways to define *min*. We may write with *otherwise* as

$$\begin{array}{rcll} min\ x\ y & = & x, & \text{if } x \leq y \\ & = & y, & \text{otherwise} \end{array}$$

or write with explicit *case* structure as

$$\begin{array}{rcl} min\ x\ y & = & \text{case } x \leq y \text{ of} \\ & & \quad True \rightarrow x \\ & & \quad False \rightarrow y \end{array}$$

or even write with *if-then-else* structure as

$$min\ x\ y = if\ x \leq y\ then\ x\ else\ y.$$

Another useful form is *local definition* with which one can describe an expression qualified by some local functions, e.g.,

$$\begin{array}{rcl} f\ x & = & double\ x + y \\ & & \text{where } double\ x = x + x \\ & & \qquad\qquad y = 2 + x \end{array}$$

Here, the special word *where* is used to introduce two local definitions whose context (or scope) is the expression on the right hand side of the definition of f. Notice that the whole of the where-clause is intended to show it is part of this expression.

Notational conventions of functions

Functional application is denoted by a space and the argument which may be written without brackets. Thus $f\,a$ means $f\,(a)$. Functions are curried and application associates to the left. Thus $f\,a\,b$ means $(f\,a)\,b$. Functional application is regarded as more binding than any other operator, so $f\,a \oplus b$ means $(f\,a) \oplus b$ and not $f\,(a \oplus b)$.

Functional composition is denoted by a centralized circle $\circ$. By definition, $(f \circ g)\,a = f\,(g\,a)$. Functional composition is an associative operator, i.e.,

$$f \circ (g \circ h) = (f \circ g) \circ h. \tag{2.1}$$

Taking advantages of associativity, chains of compositions are usually written without brackets. The unit (or identity element) of the composition operator is the identity function denoted by id, i.e.,

$$id \circ f = f \circ id = f \tag{2.2}$$

for any function f.

Binary operators will often be denoted by $\oplus, \otimes, \odot$. Binary operators can be *sectioned*; an infix binary operators like $\oplus$ can be turned into unary functions by:

$$(a\oplus)\,b = a \oplus b = (\oplus b)\,a = (\oplus)\ a\ b = \oplus(a, b). \tag{2.3}$$

A prefix binary operator f can be turned into an infix binary operator with back-quotations by

$$f\ x\ y = x\ `f`\ y. \tag{2.4}$$

Two functions $f : S \to T$ and $g : S \to T$ are equal if they coincide on all arguments, i.e.,

$$\frac{\forall x \in S.\ f\,x = g\,x}{f = g} \tag{2.5}$$

This rule is called the *extensionality axiom*. The converse of this axiom is called *Leibniz' rule*:

$$\frac{f = g}{\forall x \in S.\ f\,x = g\,x} \tag{2.6}$$

2.2 Constructive Algorithmics

In this section, we review previous work [Hag87, Mal90, MFP91, Fok92b, TM95] on constructive algorithmics and explain some basic facts which provide theoretical basis of our method.

2.2.1 Category Theory

Category theory is one of the most abstract and general branches of mathematics. It is used, among others, to provide a general framework for theories of programming. Such theories form the basis of the design and definition of programming languages and their associated software engineering methods. In this thesis, category are used to describe a theory of data types following Hagino [Hag87], where

- The notion of functor is of particular importance in the definition of data type.
- The concept of unique extension property, also called a universal mapping property (e.g., map from the initial object), is used for the definition of a function.

We begin with some elementary category theory definitions.

Definition 1 (Category) A category is a structure with objects A, B,$\cdots$, morphisms (or called arrows) f, g, $\cdots$, and associated binary operator $\circ$ (compose) on arrows such that

$$\frac{f : A \to B, \quad g : B \to C}{g \circ f : A \to C} \tag{2.7}$$

that is, whenever there are morphisms $f : A \to B$ and $g : B \to C$ then there is an morphism, called $g \circ f$ from A to C, and

$$id_A : A \to A$$

such that for every morphism $f : A \to B$,

$$\begin{array}{rcl} f \circ id_A & = & f \\ id_B \circ f & = & f \end{array}$$

□

In other words, a category can be thought of as an edge-labeled directed graph whose edge structure is "sufficiently rich" that it includes transitive closures and single vertex cycles. We use $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$, ... for categories, and we often omit the subscript A in id_A when it is clear from context. *In this thesis, we often omit subscripts or superscripts when no ambiguity happens in order to simplify the presentation.*

Throughout this paper, we will assume that we are working within the category of $\mathcal{CPO}$, the category of complete partial orders, whose objects are types and whose arrows are continuous functions between types. We often use $\mathcal{C}$ to denote our default category, $\mathcal{CPO}$. The property we need of this this category are:

- It has an initial object $\mathbf{0}$. There is a unique morphism from this object to every other object in the category.

- It has a final (terminal) object **1**. There is a unique morphism from every object in the category to this object.
- It has products. For all pairs of objects A and B, there is an object $A \times B$, and projections from it to A and B.
- It has coproducts (separated sums). For all pairs of objects A and B, there is an object $A + B$, and injections to it from A and B.

A functor is a structure preserving mapping between categories.

Definition 2 (Functor) Let $\mathcal{C}$ and $\mathcal{D}$ be categories. a functor from $\mathcal{C}$ to $\mathcal{D}$ is a mapping F that maps objects of a category $\mathcal{C}$ to objects of a category $\mathcal{D}$, and maps morphisms in $\mathcal{C}$ to morphisms in $\mathcal{D}$ in such a way that $Ff : FA \to FB$ whenever $f : A \to B$. Furthermore, f preserves identities and compositions, i.e., for all objects A in $\mathcal{C}$,

$$Fid_A = id_{FA}, \tag{2.8}$$

and if the composition $f \circ g$ is defined in $\mathcal{C}$, then $F(g \circ f)$ is a morphism in $\mathcal{D}$ satisfying

$$F(g \circ f) = Fg \circ Ff \tag{2.9}$$

□

The following are two basic functors from $\mathcal{CPO}$ to $\mathcal{CPO}$.

Definition 3 (Identity Functor)
The identity functor I on type X and its operation on functions are defined as follows.

$$\begin{array}{lcl} I\ X & = & X \\ I\ f & = & f \end{array}$$

□

Definition 4 (Constant Functor)
The constant functor $!A$ on type X and its operation on functions are defined as follows.

$$\begin{array}{lcl} !A\ X & = & A \\ !A\ f & = & id_A \end{array}$$

□

The following are two basic (binary) functors from $\mathcal{CPO} \times \mathcal{CPO}$ to $\mathcal{CPO}$.

Definition 5 (Product Functor)
The product $X \times Y$ of two types X and Y and its operation to functions are defined as follows.

$$\begin{array}{lcl} X \times Y & = & \{(x, y) \mid x \in X,\ y \in Y\} \\ (f \times g)\ (x, y) & = & (f\,x,\ g\,y) \end{array}$$

Some related operators are:

$$\begin{array}{lcl} \pi_1\ (a, b) & = & a \\ \pi_2\ (a, b) & = & b \\ (f \vartriangle g)\ a & = & (f\,a,\ g\,a). \end{array}$$

□

The operators $\times$, $\vartriangle$, and π_i satisfy the following laws. We assume that function composition binds stronger than other binary operators such as $\times$, $\vartriangle$.

$$f \times g = (f \circ \pi_1) \vartriangle (g \circ \pi_2) \quad (2.10)$$

$$(f \times g) \circ (h \vartriangle j) = f \circ h \vartriangle g \circ j \quad (2.11)$$

$$f \circ h \vartriangle g \circ h = (f \vartriangle g) \circ h \quad (2.12)$$

$$\pi_1 \circ (f \times g) = f \circ \pi_1 \quad (2.13)$$

$$\pi_2 \circ (f \times g) = f \circ \pi_2 \quad (2.14)$$

$$\pi_1 \circ (f \vartriangle g) = f \quad (2.15)$$

$$\pi_2 \circ (f \vartriangle g) = g \quad (2.16)$$

$$\pi_1 \vartriangle \pi_2 = id_{A \times B} \quad (2.17)$$

$$f \vartriangle g = h \vartriangle j \equiv (f = h) \wedge (g = j) \quad (2.18)$$

Definition 6 (Separated Sum Functor)
The separated sum $X + Y$ of two types X and Y and its operation to functions are defined as follows.

$$\begin{array}{lcl} X + Y & = & \{1\} \times X \cup \{2\} \times Y \\ (f + g)\ (1, x) & = & (1,\ f\,x) \\ (f + g)\ (2, y) & = & (2,\ g\,y) \end{array}$$

Some related operators are:

$$\begin{array}{lcl} \iota_1\ a & = & (1, a) \\ \iota_2\ b & = & (2, b) \\ (f \triangledown g)\ (1, x) & = & f\,x \\ (f \triangledown g)\ (2, y) & = & g\ y. \end{array}$$

□

Dual to the laws for the product functor, the operators $+$, $\triangledown$, and ι_i satisfy the following laws.

$$(f + g) \circ (h + j) = f \circ h + g \circ j \quad (2.19)$$

$$
\begin{aligned}
f + g &= \iota_1 \circ f \triangledown \iota_2 \circ g && (2.20)\\
(f \triangledown g) \circ (h + j) &= (f \circ h) \triangledown (g \circ j) && (2.21)\\
h \circ f \triangledown h \circ g &= h \circ (f \triangledown g) && (2.22)\\
(f + g) \circ \iota_1 &= iota_1 \circ f && (2.23)\\
(f + g) \circ \iota_2 &= iota_2 \circ g && (2.24)\\
(f \triangledown g) \circ \iota_1 &= f && (2.25)\\
(f \triangledown g) \circ \iota_2 &= g && (2.26)\\
\iota_1 \triangledown \iota_2 &= id_{A+B} && (2.27)\\
(f \triangledown g) = (h \triangledown j) &\equiv (f = h) \wedge (g = j) && (2.28)
\end{aligned}
$$

Although the product and the separated sum are defined over 2 parameters, they can be naturally extended for n parameters. For example, the separated sum over n parameters can be defined by

$$
\begin{aligned}
+_{i=1}^{n} X_i &= \cup_{i=1}^{n}(\{\mathsf{i}\} \times X_i)\\
(+_{i=1}^{n} f_i)\,(\mathsf{j}, x) &= (\mathsf{j},\ f_j\, x).
\end{aligned}
$$

All examples of functors we have given until now belong to the class of *polynomial functors*.

Definition 7 (Polynomial Functor) Polynomial functors are generated by

$$F ::= \ I \mid\ !A \mid F_1 \times F_2 \mid F_1 + F_2 \mid F_1\,F_2$$

that is, a polynomial functor is and only is built up by the four basic functors, namely the identity functor, the constant functor, the product binary functor, and the separated sum binary functor. □

A *natural transformation* is a structure-preserving map between functors.

Definition 8 (Natural Transformation) A natural transformation from a functor F to a functor G is a polymorphic function η such that for all functions $f : A \rightarrow B$, η satisfies

$$\eta_B \circ F f = G f \circ \eta_A$$

We write this as $\eta : F \dot{\rightarrow} G$. □

A natural transformation is in fact a family of morphisms; for each type we have one morphism. For example, we have

$$id : I \dot{\rightarrow} I$$

since for all $f : A \rightarrow B$ we have

$$id_B \circ f = f \circ id_A$$

Wadler [Wad89b] and de Bruin [dB89] proved that any "reasonable" polymorphic function is a natural transformation.

2.2.2 Data Type Theory

A data type is a collection of operations (data constructors) denoting how each element of the data type can be constructed in a finite way, and via these data constructors functions on the type may be defined. So a data type is a particular algebra whose one distinguished property is categorically known as the initiality of the algebra. Dualization leads to the notion of final co-algebra.

Endofunctors on category $\mathcal{C}$ (functors from $\mathcal{C}$ to $\mathcal{C}$) are used to capture both data structure and control structure in a type definition. In this thesis, we assume that all the data types are defined by *polynomial functors*.

Definition 9 (F-algebra)
An *F-algebra* is a pair (X, ϕ), where X is an object in $\mathcal{C}$, called the *carrier* of the algebra, and ϕ is a morphism from object $F\,X$ to object X denoted by $\phi : F\,X \to X$, called the *operation* of the algebra. □

Definition 10 (F-homomorphism)
Given two F-algebras (X, ϕ) and (Y, ψ), the F *homomorphism* from (X, ϕ) to (Y, ψ) is a morphism h from object X to object Y in category $\mathcal{C}$ satisfying $h \circ \phi = \psi \circ F\,h$. □

Definition 11 (Category of F-algebras)
The *category of F-algebras* has as its objects the F-algebras and has as its morphisms all F-homomorphisms between F-algebras. Composition in the category of F-algebra is taken from $\mathcal{C}$, and so are the identities. □

It is known that the initial object in the category of F-algebras exists when F is a polynomial endofunctors[Mal90], and that this initial object can be used to define a recursive data type. The representative for the initial algebra is denoted by μF. Let $(T, in_F) = \mu F$, μF defines a data type T with the *data constructor*

$$in_F : F\,T \to T$$

and the *data destructor*

$$out_F : T \to F\,T.$$

The relationship between in_F and out_F is that they are inversions each other, i.e.,

$$in_F \circ out_F \;=\; id \qquad (2.29)$$

$$out_F \circ in_F \;=\; id \qquad (2.30)$$

Example 1 (Cons Lists) To be concrete, consider the data type of *cons lists* given by the following definition with elements of type A:

$$L\,A = Nil \mid Cons(A,\, L\,A).$$

It is categorically defined as the initial object of

$$(L\,A,\ \mathit{Nil} \triangledown \mathit{Cons})$$

in the category of F_{L_A}-algebras[2], where F_{L_A} is the endofunctor defined by

$$F_{L_A} = !\mathbf{1} + !A \times I$$

Here, the data constructor and the data destructor are as follows.

$$\begin{array}{ll} in_{F_{L_A}} & = \mathit{Nil} \triangledown \mathit{Cons} \\ out_{F_{L_A}} & = \lambda xs.\ \mathrm{case}\ xs\ \mathrm{of} \\ & \qquad \mathit{Nil} \ \to (1, ()); \\ & \qquad \mathit{Cons}\ (a, as) \to (2, (a, as)) \end{array}$$

Cons lists will be frequently used in this thesis. For notational convenience, we sometimes write $[\,]$ for the empty list Nil, $[x_1, \cdots, x_n]$ for the list $\mathit{Cons}(x_1, \cdots, \mathit{Cons}(x_n, \mathit{Nil}))$, and $xs +\!\!+\, ys$ for the concatenation of xs and ys. Concatenation is associative, and $[\,]$ is its unit. For example, the term $[1] +\!\!+ [2] +\!\!+ [3]$ denotes a list with three elements, often abbreviated to $[1, 2, 3]$. □

Example 2 (Binary Trees) The type of binary trees with elements of type A, usually declared by

$$\mathit{Tree}\,A \ = \ \mathit{Leaf}\ A \mid \mathit{Node}\ (A, \mathit{Tree}\,A, \mathit{Tree}\,A),$$

is defined by $(\mathit{Tree}\,A,\ \mathit{Leaf} \triangledown \mathit{Node})$, an initial object of the category of F_{T_A}-algebras, where F_{T_A} is the endofunctor defined by

$$F_{T_A} \ = !A + !A \times I \times I.$$

And the data constructor and the data destructor are given as follows.

$$\begin{array}{ll} in_{F_{T_A}} & = \mathit{Leaf} \triangledown \mathit{Node} \\ out_{F_{T_A}} & = \lambda t.\ \mathrm{case}\ t\ \mathrm{of} \\ & \qquad \mathit{Leaf}\ a \ \to (1, (a)); \\ & \qquad \mathit{Node}\ (a, t_l, t_r) \to (2, (a, t_l, t_r)) \end{array}$$

□

Example 3 (Naturals) The data type *natural numbers*:

$$N \ = \ Z \mid S\,N$$

is defined by $(N, Z \triangledown S)$, an initial F_N-algebra, where F_N is the endofunctor defined by

$$F_N \ = !\mathbf{1} \ + \ I.$$

Here, the data constructor and data destructor are as follows.

[2]Strictly speaking, Nil should be written as $\lambda().\mathit{Nil}$. In this paper, the function with the form of $\lambda().t$ will be simply denoted as t.

$$\begin{array}{ll} in_{F_N} & = Z \triangledown S \\ out_{F_N} & = \lambda x.\ \text{case } x \text{ of} \\ & \qquad Z \ \to (1, ()) \\ & \qquad S\, n \to (2, (n)) \end{array}$$

□

Example 4 (Boolean) The data type *boolean*, an enumerated type,

$$Bool \;=\; True \mid Flase,$$

is an initial object of the category of F_B-algebras, where F_B is the endofunctor defined by

$$F_B \;= !\mathbf{1} + !\mathbf{1}.$$

The related data constructor and data destructor are as follows.

$$\begin{array}{ll} in_{F_B} & = True \triangledown Flase \\ out_{F_B} & = \lambda x.\ \text{case } x \text{ of} \\ & \qquad True \ \to (1, ()) \\ & \qquad Flase \to (2, ()) \end{array}$$

□

One of the most attractive characteristics of category theory is that the dual theorems can be obtained directly. For example, from the result that data type can be defined as an initial functor-algebra, we can soon conclude that the data type can also be defined as a final functor-co-algebra. We shall explain it with the following example.

Example 5 (Streams) The data type *streams* over element with type A has as carrier the set $St\,A$ that consists of infinite lists only. There are two functions to destruct a stream into a head in A and a tail that is a stream over A again,. i.e.,

$$\begin{array}{lll} hd & : & St\,A \to A \\ tl & : & St\,A \to St\,A \end{array}$$

It would be more natural to define streams by a final functor-co-algebra than by an initial functor-algebra. That is, we use $(St\,A, hd \vartriangle tl)$, a final object of the category of F_{St}-co-algebras[3], to define *streams*. Here, F_{St} is defined by

$$F_{St} \;= !A \;\times\; I.$$

Obviously we have $hd \vartriangle tl : St\,A \to F_{St}(St\,A)$. And the corresponding data constructor and the data destructor are defined by

[3]Note that (X, ϕ) is a F-co-algebra if ϕ is a morphism from X to $F\,X$, i.e., $\phi : X \to F\,X$, (the dual definition of F-algebra).

$$\begin{array}{ll} in_{F_{St}} &= sCons \\ out_{F_{St}} &= hd \vartriangle tl \end{array}$$

in which hd, tl and $sCons$ satisfy

$$\begin{array}{lcl} hd\ (sCons(x, xs)) &=& x \\ tl\ (sCons(x, xs)) &=& xs \\ sCons(hd\ xs, tl\ xs) &=& xs. \end{array}$$

□

2.2.3 Functions over Data Types: Cata, Ana, Hylo

Functions are maps from elements in the source type to elements in the target type. Since a type can be formulated as an initial functor algebra (or a final functor co-algebra), we can formulate a function as a homomorphism between two functor algebras (or between functor co-algebras). By doing so, an orderly recursive structure information of data structures can be embedded in functions which is useful for program transformation. In this section, we shall introduce three schemes of recursive functions, namely, catamorphisms, anamorphisms and hylomorphisms, and their related calculational rules and properties.

Catamorphisms

Catamorphism (*cata* for short) is a homomorphism from an initial algebra to another algebra. Because a data type can be defined as an initial algebra, catamorphisms form an important class of functions consuming the data of the given type. Let (T, in_F) be the initial F-algebra for the definition of type T. We have, for every F-algebra (X, ϕ) where $\phi : FX \to X$, there exists the unique morphism $f : T \to X$ satisfying

$$f \circ in_F = \phi \circ F\, f.$$

We denote the unique solution of f in the above equation by $(\![\phi]\!)_F$, and call it F-catamorphism (or catamorphism if F is clear). Initiality of (T, in_F) is fully captured by the following *uniqueness* law:

$$f = (\![\phi]\!)_F \quad \equiv \quad f \circ in_F = \phi \circ F\, f. \tag{2.31}$$

For example, a catamorphism over type $L\,A$ can be generally represented as a function $(\![e \triangledown \otimes]\!)_{F_{L_A}} :: L\,A \to X$, where X is another type, e is a function with type $\mathbf{1} \to X$, and $\otimes$ is a function with type $(A, X) \to X$. According to the definition of catamorphisms, we know that

$$(\![e \triangledown \otimes]\!)_{F_{L_A}} \circ (Nil \triangledown Cons) = (e \triangledown \otimes) \circ F_{L_A} (\![e \triangledown \otimes]\!)_{F_{L_A}},$$

which can be in-lined to our familiar recursive equation:

$$\begin{array}{lcl} (\!|e \triangledown \otimes|\!)_{F_{L_A}}\ \mathit{Nil} & = & e \\ (\!|e \triangledown \otimes|\!)_{F_{L_A}}\ (\mathit{Cons}(x, xs)) & = & x \ \otimes\ (\!|e \triangledown \otimes|\!)_{F_{L_A}}\ xs \end{array}$$

In essence, $(\!|e \triangledown \otimes|\!)_{F_{L_A}}$ is *relabeling*: it replaces every occurrence of "*Nil*" with e and every occurrence of "*Cons*" with $\otimes$ in the cons list. For example,

$$\mathit{sum} = (\!|\mathit{0} \triangledown +|\!)_{F_{L_{Int}}} \tag{2.32}$$

is a function computing the sum of a cons list, and

$$\mathit{double} = (\!|\mathit{Nil} \triangledown \mathit{Cons} \circ ((\mathit{2}\times) \times \mathit{id})|\!)_{F_{L_{Int}}} \tag{2.33}$$

is a function doubling every element of a list.

Catamorphisms play an important role in program transformation (program calculation) in that they satisfy a number of nice calculational properties of which the *Fusion* Theorem is of greatest importance:

Theorem 1 (Cata Fusion)

$$\frac{f \circ \phi = \psi \circ F\, f}{f \circ (\!|\phi|\!)_F = (\!|\psi|\!)_F}$$

□

The Fusion Theorem gives the condition that has to be satisfied in order to "fuse" a function into a catamorphism to obtain a new catamorphism. Using fusion, we can derive a catamorphism for *sum* ∘ *double* by the following calculation.

$$\begin{array}{cl} & \mathit{sum} \circ (\mathit{Nil} \triangledown \mathit{Cons} \circ ((\mathit{2}\times) \times \mathit{id})) \\ = & \quad \{\ 2.21\ \} \\ & \mathit{sum} \circ (\mathit{Nil} \triangledown \mathit{Cons}) \circ (\mathit{id} + ((\mathit{2}\times) \times \mathit{id}) \\ = & \quad \{\ \text{Def. of } \mathit{sum}\ (2.32)\ \} \\ & (0 \triangledown +) \circ F_{L_{Int}} \mathit{sum} \circ (\mathit{id} + ((\mathit{2}\times) \times \mathit{id}) \\ = & \quad \{\ \text{Notice } F_{L_{Int}} f = \mathit{id} + \mathit{id} \times f\text{, (2.19), simplification }\ \} \\ & (0 \triangledown +)(\mathit{id} + ((2\times) \times \mathit{sum})) \\ = & \quad \{\ \text{Def. of } F_{L_{Int}}\text{, laws from 2.19 to 2.28 }\ \} \\ & (0 \triangledown (+ \circ ((2\times) \times \mathit{id}))) \circ F_{L_{Int}} \mathit{sum} \end{array}$$

So,

$$\begin{array}{cl} & \mathit{sum} \circ \mathit{double} \\ = & \quad \{\ 2.33\ \} \\ & \mathit{sum} \circ (\!|\mathit{Nil} \triangledown \mathit{Cons} \circ ((\mathit{2}\times) \times \mathit{id})|\!)_{F_{L_{Int}}} \\ = & \quad \{\ \text{Cata Fusion Theorem}\ \} \\ & (\!|0 \triangledown (+ \circ ((2\times) \times \mathit{id}))|\!)_{F_{L_{Int}}} \end{array}$$

Catamorphisms enjoy many useful properties. We summarize them as follows.

Corollary 2 (Cata Properties)

1. If f has a left inverse g, i.e., $g \circ f = id$, then $f = (\![f \circ in_F \circ F g]\!)_F$.
2. $(\![in_F]\!)_F = id$
3. $(\![\phi]\!)_F \vartriangle (\![\psi]\!)_F = (\![\phi \circ F\pi_1 \vartriangle \psi \circ F\pi_2]\!)_F$

$\square$

Anamorphisms

Anamorphism (*ana* for short), the dual concept of catamorphism, is a homomorphism from a functor co-algebra to a final functor co-algebra. As a data type can be defined as a final algebra, anamorphisms forms another important class of function producing the data of the given type. Let (T, out_F) be the final F-co-algebra defining the type T. We have, for every F-co-algebra (X, ψ) where $\psi : X \to FX$, there exists the unique morphism $f : X \to T$ satisfying

$$out_F \circ f = F\,f \circ \psi.$$

We denote the unique solution of f in the above equation by $[\!(\psi)\!]_F$, and call it F-anamorphism (or anamorphism if F is clear). Finality of (T, out_F) is fully captured by the following *uniqueness* law:

$$f = [\!(\psi)\!]_F \quad \equiv \quad out_F \circ f = F\,f \circ \psi. \tag{2.34}$$

For example,

$$\begin{array}{l} upto = [\!(\psi)\!]_{F_{L_{Int}}} \\ \qquad \text{where } \psi = \lambda(b,e).\ \text{case } b \le e \text{ of} \\ \qquad\qquad\qquad\qquad True \to (1,()); \\ \qquad\qquad\qquad\qquad False \to (2,(b,(b+1,e))) \end{array}$$

defines the function ($upto : Int \to List\ Int$) constructing a list of integer from b to e for a given pair (b, e).

Similar to catamorphisms, anamorphisms have the following fusion theorem with the following equation.

$$[\!(out_F)\!]_F = id \tag{2.35}$$

Theorem 3 (Ana Fusion)

$$\frac{\phi \circ g = F\,g \circ \psi}{[\!(\phi)\!]_F \circ g = [\!(\psi)\!]_F}$$

$\square$

Hylomorphisms

A composition of a catamorphism and an anamorphism gives a *hylomorphism*. Hylomorphisms in triplet form[TM95] are defined as follows.

Definition 12 (Hylomorphism in triplet form)
Given two morphisms $\phi : G\,A \to A$, $\psi : B \to F\,B$ and natural transformation $\eta : F \stackrel{\cdot}{\to} G$, the hylomorphism $[\![\phi, \eta, \psi]\!]_{G,F}$ is defined as the least morphism $f : B \to A$ satisfying the following equation.

$$f = \phi \circ (\eta \circ F\,f) \circ \psi$$

□

Theorem 4 (Hylo Split)

$$[\![\phi, \eta, \psi]\!]_{G,F} = (\![\phi \circ \eta]\!)_F \circ [\!(\psi)\!]_F = (\![\phi]\!)_G \circ [\!(\eta \circ \psi)\!]_G$$

□

Hylomorphisms are powerful in description in that practically every recursion of interest (e.g., primitive recursions) can be specified by them [BdM94]. Hylomorphisms are quite general in that many useful forms, such as cata and ana, are their special cases as defined below. Note that we sometimes omit the subscripts G and F when it is clear from the context.

Theorem 5 (Cata, Ana and Hylo) Let $(T_F, in_F) = \mu F$

$$\begin{array}{lcl}
(\![\_]\!)_F & : & \forall A.\ (F\,A \to A) \to T_F \to A \\
(\![\phi]\!)_F & = & [\![\phi, id, out_F]\!]_{F,F} \\
 & & \\
[\!(\_)\!]_F & : & \forall A.\ (A \to F\,A) \to A \to T_F \\
[\!(\psi)\!]_F & = & [\![in_F, id, \psi]\!]_{F,F}
\end{array}$$

□

Catamorphisms $(\![\_]\!)$ are generalized foldr operators (or reduces) that substitute the constructor of a data type with other operation of the same signature. Dually, anamorphisms $[\!(\_)\!]$ are generalized unfold operators (or generations).

Theorem 6 (Hylo Shift)

$$[\![\phi, \eta, \psi]\!]_{G,F} = [\![\phi \circ \eta, id, \psi]\!]_{F,F} = [\![\phi, id, \eta \circ \psi]\!]_{G,G}$$

□

The Hylo Shift Theorem shows that some computations, represented by natural transformation η, can be shifted within a hylomorphism. For program fusion, hylomorphisms possess the general laws called the *Hylo Fusion Theorem*.

Theorem 7 (Hylo Fusion)

Left Fusion Law:

$$\frac{f \circ \phi = \phi' \circ G\, f}{f \circ [\![\phi, \eta, \psi]\!]_{G,F} = [\![\phi', \eta, \psi]\!]_{G,F}}$$

Right Fusion Law:

$$\frac{\psi \circ g = F\, g \circ \psi'}{[\![\phi, \eta, \psi]\!]_{G,F} \circ g = [\![\phi, \eta, \psi']\!]_{G,F}}$$

□

These laws are quite general in the sense that the functions to be fused with, e.g., f and g in Theorem 7, can be any functions. If f and g are restricted to specific hylomorphisms, we could have the following simple but practical *Acid Rain Theorem*[TM95].

Theorem 8 (Acid Rain)

Cata–Hylo Fusion Law:

$$\frac{\tau : \forall A.\ (F\, A \to A) \to F'\, A \to A}{[\![\phi, \eta_1, out_F]\!]_{G,F} \circ [\![\tau\, in_F, \eta_2, \psi]\!]_{F',L} = [\![\tau(\phi \circ \eta_1), \eta_2, \psi]\!]_{F',L}}$$

Hylo–Ana Fusion Law:

$$\frac{\sigma : \forall A.\ (A \to F\, A) \to A \to F'\, A}{[\![\phi, \eta_1, \sigma\, out_F]\!]_{G,F'} \circ [\![in_F, \eta_2, \psi]\!]_{F,L} = [\![\phi, \eta_1, \sigma(\eta_2 \circ \psi)]\!]_{G,F'}}$$

□

Chapter 3

Towards Calculation: Deriving Hylomorphisms

Recursive definitions and higher order functions, as found in most functional programming languages, provide a powerful mechanism for specifying programs, supporting forms of modularity that are difficult to support in other languages [Hug89, Wad89a]. Recent works [Hag87, Mal90, MFP91, Fok92a, Fok92b, CF92, SF93, KL94, GLJ93, TM95], basically based on the theory of *Constructive Algorithmics* (see Section 2.2), have suggested an even higher level of modularity and abstraction can be obtained by the use of generic control structures, such as *catamorphisms* (or called foldr), *anamorphisms* (or called generator), and *hylomorphisms*, which capture patterns of recursion for a large class types in a uniform way. By programming with a small, fixed set of recursive patterns derivable from *algebraic type* definitions, an orderly structure can be imposed upon functional programs. Such structures are exploited to facilitate the proof of program properties, and even to calculate program transformation [MFP91, SF93].

However, this approach imposes recursive structures of specific forms on programs, which is unrealistic in real functional programming. It is impractical to force programmers to write programs with catamorphisms, anamorphisms, or hylomorphisms which are quite abstract and contain much knowledge of category theory.

The purpose of this chapter is to demonstrate how practical recursive definitions can be automatically turned into hylomorphism, general forms covering all those in [MFP91, SF93, TM95] such as catamorphisms and anamorphisms (see Theorem 5), which makes room for our later optimization of functional program by program calculation. The main contribution of this work, the first part in [HIT96d], is as follows.

- We propose an algorithm which can automatically turn almost *all* recursive definitions of interest to structural hylomorphisms. With the fusion systems[SF93, KL94, TM95] successfully developed for hylomorphisms, we can improve a larger class of programs freely defined by programmers.

$Decl$	$::=$	$v = b$	(recursive) function definition
b	$::=$	$\lambda v_s.\ \text{case}\ t_0\ of\ r$	definition body
v_s	$::=$	$v \mid (v_1, \cdots, v_n)$	argument
r	$::=$	$p_1 \to t_1; \cdots; p_n \to t_n$	alternatives
t	$::=$	v	variable
	$\mid$	$(t_1, \cdots, t_n)$	term tuple
	$\mid$	$v\ t$	function application
	$\mid$	$C\ t$	constructor application
p	$::=$	$C\ p$	pattern
	$\mid$	$(p_1, \cdots, p_n)$	pattern tuple
	$\mid$	v	variable

Figure 3.1: The language for Recursive Definitions

- Our algorithm is guaranteed to be correct, and to terminate with a successful hylomorphism. To the contrary, Launchbury and Sheard's derivation algorithm of build-catas [LS95] from recursive definitions may fail to give build-catas. Their algorithm has to be on the alert against failure of the "two-stage fusion" and gives up the derivation in case failure occurs.

This chapter is organized as follows. Section 3.1 defines a simple language for the description of recursive definitions. Section 3.2 defines our algorithm for deriving hylomorphisms from recursive definitions. And Section 3.3 gives the related work and discussions.

3.1 A Simple Language

To demonstrate our techniques, we use the language given in Figure 3.1 for the description of *recursive* definitions. It is nothing special except that functions are defined in an uncurried way. In order to simplify our presentation, we restrict ourselves to single-recursive data types and functions without mutual recursions, since the standard tupling technique can transform mutual recursive definitions to non-mutual ones. We also assume that recursive function calls are not nested and only occur in the terms of the alternatives in the definition body. It should be noted that these restrictions can in fact be removed by program analysis. It acts as our core language. In practical functional programming, we can use some syntactic sugar for readability.

Several simple examples of recursive definitions are given below. The function *sum* sums up all the elements in a list, the function *upto* generates a list of natural numbers between two given numbers, and the function *zip* turns a pair of lists into a list of pairs.

$$sum\ :\ List\ A \to Int$$

$$\begin{array}{l} \mathit{sum} = \lambda xs.\ \text{case}\ xs\ \text{of}\ \mathit{Nil} \to 0; \\ \qquad\qquad\qquad\qquad \mathit{Cons}\,(a, as) \to \mathit{plus}\,(a, \mathit{sum}\ as) \end{array}$$

$$\begin{array}{l} \mathit{upto}\ : \mathit{Int} \times \mathit{Int} \to \mathit{List\ Int} \\ \mathit{upto} = \lambda(b, e).\ \text{case}\ b < e\ \text{of} \\ \qquad\qquad\qquad \mathit{True} \to \mathit{Nil}; \\ \qquad\qquad\qquad \mathit{False} \to \mathit{Cons}\,(b, \mathit{upto}\,(\mathit{plus}\,(b, 1), e)) \end{array}$$

$$\begin{array}{l} \mathit{zip}\ :\ \mathit{List}\ A \times \mathit{List}\ B \to \mathit{List}\ (A \times B) \\ \mathit{zip} =\ \lambda(xs, ys).\ \text{case}\ (xs, ys)\ \text{of} \\ \qquad\qquad\qquad (\mathit{Nil}, \_) \to \mathit{Nil}; \\ \qquad\qquad\qquad (\mathit{Cons}\,(a, as), \mathit{Nil}) \to \mathit{Nil}; \\ \qquad\qquad\qquad (\mathit{Cons}\,(a, as), \mathit{Cons}\,(b, bs)) \to \mathit{Cons}\,((a, b), \mathit{zip}\,(as, bs)) \end{array}$$

Here *plus* is the function adding two integers.

Next, let's consider definitions of higher order functions, such as

$$\begin{array}{l} \mathit{map} : (A \to B) \to \mathit{List}\ A \to \mathit{List}\ B \\ \mathit{map}\ g = \lambda xs.\ \text{case}\ xs\ \text{of} \\ \qquad\qquad\qquad \mathit{Nil} \to \mathit{Nil}; \\ \qquad\qquad\qquad \mathit{Cons}\,(a, as) \to \mathit{Cons}\,(g\,a, \mathit{map}\,g\,as) \end{array}$$

which is not a valid definition in our language since the defined function *map g* is not a variable. The simplest way to solve this problem is to consider *map g* as a "packed" variable and g as a global function. That is, the above definition is considered something like

$$\begin{array}{l} \mathit{map\_g} = \lambda xs.\ \text{case}\ xs\ \text{of} \\ \qquad\qquad\qquad \mathit{Nil} \to \mathit{Nil}; \\ \qquad\qquad\qquad \mathit{Cons}\,(a, as) \to \mathit{Cons}\,(g\,a, \mathit{map\_g}\,as). \end{array}$$

Viewed in this way, *map g* can be regarded as a valid definition. Below is another similar example.

$$\begin{array}{l} \mathit{foldr}1 : (B \times B \to B) \to \mathit{List}\ B \to B \\ \mathit{foldr}1 \oplus = \lambda xs.\ \text{case}\ xs\ \text{of} \\ \qquad\qquad\qquad \mathit{Nil} \to \mathit{error}; \\ \qquad\qquad\qquad \mathit{Cons}\,(a, \mathit{Nil}) \to a; \\ \qquad\qquad\qquad \mathit{Cons}\,(a, as) \to a \oplus (\mathit{foldr}1\ \oplus\ as) \end{array}$$

3.2 Expressing Recursions as Hylomorphisms

In this section, we propose an algorithm to turn recursive definitions into hylomorphisms. Consider the following typical recursive definition of function f.

$$f = \lambda v_s.\ \text{case}\ t_0\ \text{of}\ p_1 \rightarrow t_1; \cdots; p_n \rightarrow t_n$$

If we can turn the right hand side into the form of $\phi \circ F\, f \circ \psi$, it soon follows that $f = [\![\phi, id, \psi]\!]_{F,F}$ from the definition of hylomorphisms (Definition 12).

3.2.1 Main idea

The trick to do so is to turn each term t_i $(i = 1, \cdots, n)$ into $g_i\, t'_i$, a suitable function being applied to a new term. Put it in more detail, supposing that we have got that $t_i = g_i\, t'_i$, the original definition becomes:

$$f = \lambda v_s.\ \text{case}\ t_0\ \text{of}\ p_1 \rightarrow g_1\, t'_1; \cdots; p_n \rightarrow g_n\, t'_n.$$

Extracting all g_i's out and adding tag i to t'_i can give a compositional description of f:

$$\begin{aligned} f = {} & (g_1 \triangledown \cdots \triangledown g_n) \circ \\ & (\lambda v_s.\ \text{case}\ t_0\ \text{of}\ p_1 \rightarrow (\mathsf{1}, t'_1); \cdots; p_n \rightarrow (\mathsf{n}, t'_n)). \end{aligned}$$

If g_i can be expressed as $\phi_i \circ F_i\, f$ where F_i is some functor, it soon follows that

$$g_1 \triangledown \cdots \triangledown g_n = (\phi_1 \triangledown \cdots \triangledown \phi_n) \circ (F_1 + \cdots + F_n)\, f.$$

Now replacing it in the above compositional description of f gives:

$$\begin{aligned} f = {} & (\phi_1 \triangledown \cdots \triangledown \phi_n) \circ (F_1 + \cdots + F_n)\, f \circ \\ & (\lambda v_s.\ \text{case}\ t_0\ \text{of}\ p_1 \rightarrow (1, t'_1); \cdots; p_n \rightarrow (n, t'_n)). \end{aligned}$$

According to the definition of hylomorphisms, we have

$$f = [\![\phi, id, \psi]\!]_{F,F}$$

where

$$\begin{aligned} F &= F_1 + \cdots + F_n \\ \phi &= \phi_1 \triangledown \cdots \triangledown \phi_n \\ \psi &= \lambda v_s.\ \text{case}\ t_0\ \text{of}\ p_1 \rightarrow (\mathsf{1}, t'_1); \cdots; p_n \rightarrow (\mathsf{n}, t'_n). \end{aligned}$$

The essential point of our algorithm is, therefore, to derive a function ϕ_i, a functor F_i and a new term t'_i from each term t_i satisfying $t_i = (\phi_i \circ F_i\, f)\ t'_i$.

$$
\begin{array}{l}
\mathcal{A}[\![f = \lambda vs.\text{case } t_0 \text{ of } p_1 \to t_1; \cdots; p_n \to t_n]\!] = (f = [\![\phi_1 \triangledown \cdots \triangledown \phi_n, id, \psi]\!]_{F,F}) \\
\quad \text{where } F = F_1 + \cdots + F_n \\
\qquad\quad F_i = !\mathbf{1}, \quad \text{if } k_i = l_i = 0 \\
\qquad\qquad = !\Gamma(v_{i_1}) \times \cdots \times !\Gamma(v_{i_{k_i}}) \times I_1 \times \cdots \times I_{l_i}, \quad (I_1 = \cdots = I_{l_i} = I), \quad \text{otherwise} \\
\qquad\quad \phi_i = \lambda\ (v_{i_1}, \cdots, v_{i_{k_i}}, v'_{i_1}, \cdots, v'_{i_{l_i}}).\ t''_i \\
\qquad\quad \psi = \lambda vs.\text{case } t_0 \text{ of } p_1 \to (\mathsf{1}, t'_1); \cdots; p_n \to (\mathsf{n}, t'_n) \\
\qquad\quad t'_i = (v_{i_1}, \cdots, v_{i_{k_i}}, t_{i_1}, \cdots, t_{i_{l_i}}) \\
\qquad\quad \text{where} \\
\qquad\qquad (\{v_{i_1}, \cdots, v_{i_{k_i}}\}, \{(v'_{i_1}, f\, t_{i_1}), \cdots, (v'_{i_{l_i}}, f\, t_{i_{l_i}})\}, t''_i) = \mathcal{D}[\![t_i]\!], \quad (i = 1, \cdots, n) \\
\qquad\qquad \Gamma(v) = \text{return } v\text{'s type} \\
\\
\qquad\qquad \mathcal{D}[\![v]\!] \qquad\qquad = \text{if } v \text{ is a global variable then } (\{\}, \{\}, v) \text{ else } (\{v\}, \{\}, v) \\
\qquad\qquad \mathcal{D}[\![(t_1, \cdots, t_n)]\!] = (s_1 \cup \cdots \cup s_n, c_1 \cup \cdots \cup c_n, (t'_1, \cdots, t'_n)) \\
\qquad\qquad\qquad\qquad\qquad \text{where } (s_i, c_i, t'_i) = \mathcal{D}[\![t_i]\!], \quad i = 1, \cdots, n \\
\qquad\qquad \mathcal{D}[\![v\, t]\!] \qquad\quad = \text{if } v = f \text{ then } (\{\,\}, \{(u, f\, t)\}, u) \text{ else } (s_v \cup s_t, c_v \cup c_t, t_v\, t_t) \\
\qquad\qquad\qquad\qquad\qquad \text{where } (s_v, c_v, t_v) = \mathcal{D}[\![v]\!], (s_t, c_t, t_t) = \mathcal{D}[\![t]\!] \\
\qquad\qquad\qquad\qquad\qquad\qquad u \text{ is a fresh variable} \\
\qquad\qquad \mathcal{D}[\![C\, t]\!] \qquad\quad = (s, c, C\, t') \text{ where } (s, c, t') = \mathcal{D}[\![t]\!]
\end{array}
$$

Figure 3.2: Algorithm for Deriving Hylomorphisms

Deriving ϕ_i, F_i and t'_i from t_i

Our algorithm to derive ϕ_i, F_i and t'_i from t_i is informally as follows.

1. Identify all the occurrences of recursive calls to f in t_i, say they are $f\, t_{i_1}, \cdots, f\, t_{i_{l_i}}$;

2. Find all the free variables in t_i but not in $t_{i_1}, \cdots t_{i_{l_i}}$, say they are $v_{i_1}, \cdots, v_{i_{k_i}}$;

3. Define t'_i by tupling all the arguments of the recursive calls obtained in Step 1 and the free variables obtained in step 2, i.e. $t'_i = (v_{i_1}, \cdots, v_{i_{k_i}}, t_{i_1}, \cdots, t_{i_{l_i}})$.

4. Define F_i according to the construction of t'_i by $F_i = !\Gamma(v_{i_1}) \times \cdots \times !\Gamma(v_{i_{k_i}}) \times I_1 \times \cdots \times I_{l_i}$, where $I_1 = \cdots = I_{l_i} = I$ and Γ returns the type of the given variable.

5. Define ϕ_i by abstracting all recursive function calls in t_i by

$$\begin{array}{l} \phi_i = \lambda(v_{i_1}, \cdots, v_{i_{k_i}}, v'_{i_1}, \cdots, v'_{i_{l_i}}). \\ \qquad\qquad t_i[f\, t_{i_1} \mapsto v'_{i_1}, \cdots, f\, t_{i_{l_i}} \mapsto v'_{i_{l_i}}] \end{array}$$

   where $v'_{i_1}, \cdots, v'_{i_{l_i}}$ are new variables introduced for replacing those occurrences of $f\, t_{i_1}, \cdots, f\, t_{i_{l_i}}$.

3.2.2 Algorithm for Deriving Hylomorphisms

The derivation algorithm described above is summarized in Figure 3.2. The main algorithm is $\mathcal{A}$ which turns a recursive definition into a hylomorphism. The algorithm $\mathcal{A}$ calls the algorithm $\mathcal{D}$ to process each term t_i returning a triple, that is,

$$\left(\begin{array}{l} \{v_{i_1}, \cdots, v_{i_{k_i}}\}, \\ \{(v'_{i_1}, f\, t_{i_1}), \cdots, (v'_{i_{l_i}}, f\, t_{i_{l_i}})\}, \\ t_i[f\, t_{i_1} \mapsto v'_{i_1}, \cdots, f\, t_{i_{l_i}} \mapsto v'_{i_{l_i}}] \end{array}\right) = \mathcal{D}[\![t_i]\!].$$

The algorithm $\mathcal{D}$ actually implements most of the algorithm in Section 3.2.1. It is worth noting that every time an occurrence of recursive call is found, a fresh variable is allocated for the replacement. The algorithm $\mathcal{A}$ is correct and guaranteed to terminate with a successful hylomorphism as its result.

Theorem 9 (Correctness of Algorithm $\mathcal{A}$)
By the Algorithm $\mathcal{A}$ in Figure 3.2, a recursive definition of

$$f = \lambda v_s.\ \text{case}\ t_0\ \text{of}\ p_1 \to t_1; \cdots; p_n \to t_n$$

can be equivalently described by a hylomorphism as

$$f = [\![\phi_1 \triangledown \cdots \triangledown \phi_n, id, \psi]\!]_{F,F}.$$

Proof: First, we show that $[\![\phi_1 \triangledown \cdots \triangledown \phi_n, id, \psi]\!]_{F,F}$ is in a correct hylomorphic form because

1. F is a polynomial functor;
2. $\phi_1 \triangledown \cdots \triangledown \phi_n$ has the type of $F\,T_o \to T_o$ and ψ has the type of $T_i \to F\,T_i$, as easily verified, where T_i and T_o denote f's input type and output type respectively;
3. id is a natural transformation from F to F.

Next, we argue that $f = [\![\phi_1 \triangledown \cdots \triangledown \phi_n, id, \psi]\!]_{F,F}$ is equivalent to $f = \lambda v_s.$ case t_0 of $p_1 \to t_1; \cdots; p_n \to t_n$ by the following calculation.

$$\begin{array}{ll}
 & f = [\![\phi_1 \triangledown \cdots \triangledown \phi_n, id, \psi]\!]_{F,F} \\
\equiv & \quad \{\ \text{Definition of hylomorphism}\ \} \\
 & f = \phi_1 \triangledown \cdots \triangledown \phi_n \circ id \circ F\, f \circ \psi \\
\equiv & \quad \{\ \text{Definition of } \psi \text{ and } F\ \} \\
 & f = \phi_1 \triangledown \cdots \triangledown \phi_n \circ (F_1 + \cdots + F_n)\, f \circ \\
 & \qquad (\lambda vs.\ \text{case}\ t_0\ \text{of}\ p_1 \to (1, t'_1); \cdots; p_n \to (\mathsf{n}, t'_n)) \\
\equiv & \quad \{\ \text{Promote function into case expression}\ \} \\
 & f = \lambda v_s.\ \text{case}\ t_0\ \text{of} \\
 & \qquad\qquad p_1 \to (\phi_1 \circ F_1\, f)\ t'_1; \\
 & \qquad\qquad \cdots; \\
 & \qquad\qquad p_n \to (\phi_n \circ F_n\, f)\ t'_n \\
\equiv & \quad \{\ \text{To be proved later}\ \} \\
 & f = \lambda v_s.\ \text{case}\ t_0\ \text{of}\ p_1 \to t_1; \cdots; p_n \to t_n
\end{array}$$

To make the above proof complete, we need to show that for any i,

$$(\phi_i \circ F_i\, f)\ t'_i = t_i$$

which can be easily proved as follows.

$$\begin{array}{ll}
 & (\phi_i \circ F_i\, f)\ t'_i \\
\equiv & \quad \{\ \text{Definition of } t'_i\text{'s}\ \} \\
 & (\phi_i \circ F_i\, f)\ (v_{i_1}, \cdots, v_{i_{k_i}}, t_{i_1}, \cdots, t_{i_{l_i}}) \\
\equiv & \quad \{\ \text{Definition of } F_i \text{ and } \phi_i\ \} \\
 & (\lambda(v_{i_1}, \cdots, v_{i_{k_i}}, v'_{i_1}, \cdots, v'_{i_{l_i}}).\, t''_i) \\
 & \qquad (v_{i_1}, \cdots, v_{i_{k_i}}, f\, t_{i_1}, \cdots, f\, t_{i_{l_i}}) \\
\equiv & \quad \{\ \text{Simplication}\ \} \\
 & t''_i[v'_{i_1} \mapsto f\, t_{i_1}, \cdots, v'_{i_{l_i}} \mapsto f\, t_{i_{l_i}}] \\
\equiv & \quad \{\ \text{Property of } \mathcal{D} \text{ algorithm}\ \} \\
 & t_i
\end{array}$$

□

3.2.3 Examples

To see the algorithm $\mathcal{A}$ in action, consider some examples defined in Section 3.1.

We begin by considering a quite simple definition of *sum*. In this case, we know that $t_1 = 0$ and $t_2 = plus\,(a, sum\, as)$. Applying $\mathcal{D}$ to t_1 and t_2 gives

$$\begin{array}{lcl}
\mathcal{D}[\![0]\!] & = & (\{\}, \{\}, 0) \\
\mathcal{D}[\![plus\,(a, sum\, as)]\!] & = & (\{a\}, \{(v'_1, sum\, as)\}, plus\,(a, v'_1)).
\end{array}$$

It follows from $\mathcal{A}$ that

$$\begin{array}{ll}
t'_1 = (), & \phi_1 = \lambda().\, 0 = 0 \\
t'_2 = (a, as), & \phi_2 = \lambda(a, v'_1).\, plus\,(a, v'_1) = plus
\end{array}$$

and

$$\begin{array}{lcl}
\psi & = & \lambda xs.\ \text{case } xs \text{ of} \\
 & & \qquad Nil \to (1, ()); \\
 & & \qquad Cons\,(a, as) \to (2, (a, as)) \\
 & = & out_F \\
 & & \\
F & = & !\mathbf{1} + !Int \times I.
\end{array}$$

Therefore we derive the following hylomorphism for *sum*:

$$sum = [\![0 \triangledown plus, id, out_F]\!]_{F,F}.$$

Notice that the above hylomorphism is also a catamorphism

$$(\!|0 \triangledown \mathit{plus}|\!)_F.$$

Our second example is to deal with the recursive definition of *foldr*1 $\oplus$. This also appeared in [GLJ93] for the illustration of the limitation of their shortcut deforestation algorithm because *foldr*1 $\oplus$ cannot be specified as a catamorphism (since it does not treat all *Cons* cells identically). With the algorithm $\mathcal{A}$, we can get the following hylomorphism.

$$\begin{array}{l}
\mathit{foldr}1 \oplus = [\![\phi, \mathit{id}, \psi]\!]_{F,F} \\
\quad \text{where } F = !\mathbf{1} + !B + !B \times I \\
\qquad \phi = \phi_1 \triangledown \phi_2 \triangledown \phi_3 \\
\qquad\quad \text{where } \phi_1 = \mathit{error}; \\
\qquad\qquad \phi_2 = \lambda a.a = \mathit{id} \\
\qquad\qquad \phi_3 = \lambda(a, v_1').\ (a \oplus v_1') = \oplus \\
\qquad \psi = \lambda xs.\ \text{case } xs \text{ of} \\
\qquad\qquad \mathit{Nil} \to (1, ()); \\
\qquad\qquad \mathit{Cons}\ (a, \mathit{Nil}) \to (2, (a)); \\
\qquad\qquad \mathit{Cons}\ (a, as) \to (3, (a, as))
\end{array}$$

Section 3.3 will show how it helps removing more intermediate data structures than [GLJ93].

Next we like to show that our derivation algorithm does not restrict to the recursive definitions inducting over a single parameter. Consider the recursive definition of *zip* which inducts over multiple parameters. Applying algorithm $\mathcal{A}$ will give the following hylomorphism.

$$\begin{array}{l}
\mathit{zip} = [\![\mathit{Nil} \triangledown \mathit{Nil} \triangledown \lambda(a, b, p).\mathit{Cons}((a, b), p), \mathit{id}, \psi]\!]_{F,F} \\
\quad \text{where } F = !\mathbf{1} + !\mathbf{1} + !A \times !B \times I \\
\qquad \psi = \lambda(xs, ys).\ \text{case } (xs, ys) \text{ of} \\
\qquad\qquad (\mathit{Nil}, \_) \to (1, ()); \\
\qquad\qquad (\mathit{Cons}\,(a, as), \mathit{Nil}) \to (2, ()); \\
\qquad\qquad (\mathit{Cons}\,(a, as), \mathit{Cons}(b, bs)) \\
\qquad\qquad\quad \to (3, (a, b, (as, bs)))
\end{array}$$

The advantage of this transformation is that *zip* now can be fused with other functions by the Hylo Fusion Theorem. Compared with Fegaras' approach[FSZ94] where some new fusion theorems were intentionally developed, our algorithm makes it unnecessary to obtain the same effect. More detailed discussion can be found in Section 4.4.2

Finally, we give some other examples.

$$\mathit{map}\ g = [\![\mathit{Nil} \triangledown \lambda(a, v_1').\mathit{Cons}\,(g\,a, v_1'), \mathit{id}, \mathit{out}_{F_{L_A}}]\!]_{F_{L_A}, F_{L_A}}$$

$$
\begin{aligned}
upto = {} & [\![in_{L_{Int}}, id, \psi]\!]_{L_{Int}, L_{Int}} \\
& \text{where } \psi = \lambda(b,e).\ \text{case } b < e \text{ of} \\
& \qquad\qquad\qquad\qquad \mathit{True} \rightarrow (1, ()); \\
& \qquad\qquad\qquad\qquad \mathit{False} \rightarrow (2, (b, (plus\ (b,1), e)))
\end{aligned}
$$

3.3 Related Work and Discussions

It has been argued that programming with the use of generic control structures which capture patterns of recursions in a uniform way is very significant in program transformation and optimization[GLJ93, MFP91, SF93, TM95]. Our work is much related to these studies. In particular, our work was greatly motivated by Sheard and Fegaras' work[SF93] and Takano and Meijer's[TM95].

Basically, both of them work with a language without general recursion but containing hylomorphisms as basic components, advocating *structural functional programming.* Sheard and Fegaras implemented a fusion algorithm called *normalization algorithm*[1] based on the similar theorem like the Hylo Fusion Theorem. Takano and Meijer generalized Gill, Launchbury and Peyton Jones's one-step fusion algorithm [GLJ93] relying on functions being written in a highly-stylized *build-cata* forms (i.e., catamorphisms with data constructors being parameterized), and implemented another one-step fusion algorithm based on the Acid Rain Theorem. All previous work have made it clear that the fusion process is especially successful if the recursive definitions are expressed in terms of hylomorphisms.

However, as argued by Launchbury and Sheard[LS95], although structural functional programming contributes much to program transformation and optimization, it is unrealistic in real functional programming. It is impractical to force programmers to define their recursive definitions only in terms of the specific forms like hylomorphisms, in which a lot of abstract categorical concepts are embedded.

To remedy this situation, Launchbury and Sheard[LS95] gave an algorithm to turn recursive definitions into build-cata forms. The major problem still left is that many programs cannot be fused because some recursive definitions cannot be specified in build-cata forms. Our algorithm can solve this problem.

This work extends our previous work [HIT95a] in which so-called *mediotypes* are constructed to capture the recursive skeletons so that the fusion laws for these recursions are derived.

[1]Sheard and Fegaras' normalization algorithm first only worked with the language containing folds (i.e., catamorphisms) as basic components. It has been extended to work with languages containing so-called homomorphisms (i.e., hylomorphisms) in [KL94].

Chapter 4

Calculating Fusion Transformation

The compositional style of functional programming has the advantages of clarity and higher level of modularity. It constructs a complex program by gluing components which are relatively simple, easier to write, and potentially more reusable. However, some data structures, which are constructed in one component and consumed in another but never appear in the result of the whole program, give rise to the problem of efficiency.

Consider a toy example of function *all* testing whether all elements of a list satisfy given predicate p. It may be defined as follows.

```
$all\ p = and \circ (map\ p)$
  where $and = \lambda xs.$ case $xs$ of
                 $Nil \rightarrow True;$
                 $Cons\,(a, as) \rightarrow a \wedge (and\ as)$
```

Here p is applied to all elements of the list producing an intermediate list of Booleans which are then "anded" together by the function *and* producing a single Boolean result. To make the function *all* be computed efficiently, it is expected to fuse *and* and *map p* together to have the following new definition where the intermediate list of Booleans is never produced.

```
$all\ p = \lambda xs.$ case $xs$ of
           $Nil \rightarrow True;$
           $Cons\,(a, as) \rightarrow p\,a \wedge (all\ p\ as)$
```

There are two kinds of approaches dealing with such fusion. One, first proposed by Wadler [Wad88, Chi92] as called *deforestation*, aims to fuse *arbitrary* functions by *fold-unfold* transformations, keeping track of function calls and using clever control to avoid infinite unfolding. The other [GLJ93, SF93, TM95], quite differently, makes use of some *specific* forms such as *catamorphisms* (or called *folds*), *anamorphisms* (or called *unfolds*) and *hylomorphisms* and finds how they interact.

The second approach has been proved to be more practical in a real implementation in compilers, although at first sight it seems less general than the former. In Chapter 3, we have demonstrated how practical recursive definitions can be automatically turned into hylomorphism, general forms covering all those in [MFP91, SF93, TM95], which makes program fusion transformation be applied better. In this chapter, we focus ourselves on how to calculate fusion so that more efficient functional programs can be obtained. The main contribution of this work, the second part in [HIT96d], is as follows.

- We propose our algorithm for structuring hylomorphisms so that the Hylo Fusion and Acid Rain Theorems (Section 2.2) can be effectively applied. Particularly, we propose a new theorem for deriving polymorphic functions (Section 4.2.2), namely τ and σ, for the use of Acid Rain Theorem.
- Our algorithm does not limit its use to the fusion of recursions inducting over a single data structure. It is helpful for the fusion of recursions over multiple data structures without introducing new fusion theorems as in [FSZ94] (Section 3.2.3).

4.1 Fusion in Calculational Form

Recall that by the algorithm in Chapter 3 we can derive a hylomorphism, say

$$[\![\phi, \eta, \psi]\!]_{G,F},$$

from a recursive definition. In particular, we have two interesting cases.

- If $\psi = out_F$, then the hylomorphism reduces to the following catamorphism:
$$(\![\phi \circ \eta]\!)_F.$$
- If $\phi = in_G$, the hylomorphism reduces to the following anamorphism:
$$[\!(\eta \circ \psi)\!]_G.$$

Now suppose that the recursion is composed with another function. It turns out that fusion transformation can be performed directly either by Theorem 1, or by Theorem 3, or generally by Theorem 7.

Consider the example given at the beginning of this chapter. Recall that in Section 3.2.3 we obtained that

$$map\ p = [\![Nil \triangledown \lambda(a, v_1').Cons\,(p\,a, v_1'), id, out_{F_{L_A}}]\!]_{F_{L_A}, F_{L_A}}$$

which is simply a catamorphism:

$$map\ p = (\![Nil \triangledown \lambda(a, v_1').Cons\,(p\,a, v_1')]\!)_{F_{L_A}}.$$

We can then fuse the composition, $all\ p = and \circ map\ p$, based on Theorem 1.

First, we calculate ψ from the fusable condition of Theorem 1. Note that in this case $\phi = Nil \triangledown \lambda(a, v_1').Cons\,(p\,a, v_1')$.

$$\begin{array}{cl} & and \circ (Nil \triangledown \lambda(a,v_1').Cons\,(p\,a, v_1')) \\ = & \quad \{\ \triangledown\ \} \\ & (and \circ Nil) \triangledown \lambda(a,v_1').and\,(Cons\,(p\,a,v_1')) \\ = & \quad \{\ \text{Def. of } and\ \} \\ & True \triangledown \lambda(a,v_1').p\ a \wedge (and\ \ v_1') \\ = & \quad \{\ \text{Define } \psi = True \triangledown \lambda(a,v_1').p\ a \wedge v_1'\ \ \} \\ & \psi \circ F_{L_A} and \end{array}$$

Then according to Theorem 1, we soon have

$$all\ p = (\![\psi]\!)_{F_{L_A}}$$

which can be easily in-lined to the efficient program given before.

However, things are not always so simple, especially in the case where recursions can only be expressed as general hylomorphisms rather than simple catamorphisms or anamorphisms. In the following, we would like to show how fusion transformation can be effectively applied to hylomorphisms after some kind of restructuring transformation.

4.2 Structural Hylomorphisms

Once a hylomorphism is obtained, it becomes possible to be fused with other functions. But not all hylomorphisms have good structures for program fusion to be effectively applied. In this section, we show how to structure given hylomorphism $[\![\phi, \eta, \psi]\!]_{G,F}$ by arranging ϕ, η and ψ well inside it, so that it could be fused with other functions effectively by the two fusion theorems, namely the Hylo Fusion and the Acid Rain Theorems.

4.2.1 Suitable Hylomorphisms for Hylo Fusion Theorem

The effective use of the Hylo Fusion Theorem requires that *ϕ (ψ) in a hylomorphism $[\![\phi, \eta, \psi]\!]_{G,F}$ contain as much computation as possible* for the Left (Right) Fusion Law. As an example, consider the following program *foo*, where N represents the type of natural numbers with two constructors Z and S (See Example 3).

$$\begin{array}{lcl} foo & : & List\ N \to List\ N \\ foo & = & h \circ [\![in_{F_{L_N}}, id, (id + S \times id) \circ out_{F_{L_N}}]\!]_{F_{L_N}, F_{L_N}} \end{array}$$

where

$$
\begin{aligned}
h = \lambda xs.\ \text{case } xs \text{ of} & \\
& \mathit{Nil} \to \mathit{Nil}; \\
& \mathit{Cons}\,(Z, \mathit{Nil}) \to \mathit{Nil}; \\
& \mathit{Cons}\,(Z, \mathit{Cons}\,(x', xs')) \to \mathit{Cons}\,(S\ x', h\ xs'); \\
& \mathit{Cons}\,(S\ x', xs') \to \mathit{Cons}\,(S\ x', h\ xs')
\end{aligned}
$$

To fuse *foo* by the Left Fusion Law, we have to derive ϕ' from the equation $h \circ \mathit{in}_{F_{L_N}} = \phi' \circ F_{L_N}\, h$. But such ϕ' cannot be derived because of the lack of information in $\mathit{in}_{F_{L_N}}$. To solve this problem, we shift some computations and get the following program:

$$\mathit{foo} = h \circ [\![\mathit{in}_{F_{L_N}} \circ (\mathit{id} + S \times \mathit{id}), \mathit{id}, \mathit{out}_{F_{L_N}}]\!]_{F_{L_N}, F_{L_N}}.$$

Because of the knowledge that the value of $S\,x$ cannot be Z and thus the second and third branchs of the case expression in h cannot be taken, we can derive that

$$\phi' = \mathit{Nil} \triangledown (\mathit{Cons}\ \circ\ (S \times \mathit{id}))$$

and have

$$\mathit{foo} = [\![\phi', \mathit{id}, \mathit{out}_{F_{L_N}}]\!]_{F_{L_N}, F_{L_N}}.$$

Similar cases might happen to the Right Fusion Law.

4.2.2 Suitable Hylomorphisms for Acid Rain Theorem

The Acid Rain Theorem expects ϕ and ψ in the hylomorphism $[\![\phi, \eta, \psi]\!]_{G,F} : A \to B$ to be described as $\tau\ \mathit{in}_{F_B}$ and $\sigma\ \mathit{out}_{F_A}$ respectively. Here, τ and σ are polymorphic functions and F_A and F_B are functors defining types A and B respectively. Our Laws for deriving such τ and σ are as follows.

Theorem 10 (Deriving Polymorphic Function)
Under the above conditions, τ and σ are defined by the following two laws.

$$\frac{\forall \alpha.\ \ (\![\alpha]\!)_{F_B} \circ \phi = \phi' \circ G\,(\![\alpha]\!)_{F_B}}{\tau = \lambda \alpha.\ \phi'}$$

$$\frac{\forall \beta.\ \ \psi \circ [\!(\beta)\!]_{F_A} = F\,[\!(\beta)\!]_{F_A} \circ \psi'}{\sigma = \lambda \beta.\ \psi'}$$

Proof: We shall only prove the law for defining τ. The law for defining σ can be proved in a dual way.

First, we have to check if τ has the type of $\forall A.\ (F_B\, A \to A) \to G\, A \to A$ as required. This is obvious since τ is defined for all $\alpha : F_B\, A \to A$ with any A;

Next we prove that $\phi = \tau \; in_{F_B}$ by the following calculation.

$$\begin{array}{cl}
 & \phi = \tau \; in_{F_B} \\
\equiv & \quad \{ \text{ Definition of } \tau \text{ } \} \\
 & \phi = \phi'[\alpha \mapsto in_{F_B}] \\
\equiv & \quad \{ \text{ Since } (\![in_{F_B}]\!)_{F_B} = id \text{ and } G\,id = id \text{ } \} \\
 & (\![in_{F_B}]\!)_{F_B} \circ \phi = \phi'[\alpha \mapsto in_{F_B}] \circ G\,(\![in_{F_B}]\!)_{F_B} \\
\Leftarrow & \quad \{ \text{ Assumption } \} \\
 & \forall \alpha. \;\; (\![\alpha]\!)_{F_B} \circ \phi = \phi' \circ G\,(\![\alpha]\!)_{F_B}
\end{array}$$

□

Because in applying the Acid Rain Theorem we have to derive ϕ' (ψ') from ϕ (ψ) for *any* α (β) in order to define τ (σ), it is expected that ϕ *(ψ) is simple (contains few computations).*

To see the practical usage of Theorem 10, we demonstrate how to derive σ from ψ (the third part of hylomorphisms). Observe that ψ is usually in the form of

$$\lambda xs.\; \text{case } xs \textit{ of } p_1 \to (\mathsf{1}, t_1); \cdots ; p_n \to (\mathsf{n}, t_n).$$

Sometimes it can be transformed into

$$\chi \circ \kappa \text{ where } \kappa = F_A^d out_{F_A} \circ \cdots \circ F_A^1 \; out_{F_A} \circ out_{F_A}.$$

Here κ unfolds the data d times to cover all nested pattern p_i's, and χ is a transformation from F_A^{d+1} structure[1] to F structure to match every term t_i with p_i while satisfying the following property.

$$\forall \beta : X \to F_A X.\; \chi \circ F_A^{d+1} [\!(\beta)\!]_{F_A} = F [\!(\beta)\!]_{F_A} \circ \chi[out_{F_A} \mapsto \beta]$$

In this case, according to Theorem 10, σ will be defined as

$$\sigma = \lambda \beta.\, \chi[out_{F_A} \mapsto \beta] \circ F_A^d \beta \circ \cdots \circ F_A^1 \; \beta \circ \beta,$$

because

$$\begin{array}{cl}
 & \sigma = \lambda \beta.\, \chi[out_{F_A} \mapsto \beta] \circ F_A^d \beta \circ \cdots \circ F_A^1 \; \beta \circ \beta \\
\Leftarrow & \quad \{ \text{ Theorem 10 } \} \\
 & \chi \circ F_A^d out_{F_A} \circ \cdots \circ F_A^1 \; out_{F_A} \circ out_{F_A} \circ [\!(\beta)\!]_{F_A} \\
 & \quad = F[\!(\beta)\!]_{F_A} \circ \chi[out_{F_A} \mapsto \beta] \circ F_A^d \beta \circ \cdots \circ F_A^1 \; \beta \circ \beta \\
\equiv & \quad \{ \text{ Anamorphism: } out_{F_A} \circ [\!(\beta)\!]_{F_A} = F_A [\!(\beta)\!]_{F_A} \circ \beta \text{ } \} \\
 & \chi \circ F_A^d out_{F_A} \circ \cdots \circ F_A^1 \; out_{F_A} \circ F_A^1 [\!(\beta)\!]_{F_A} \circ \beta \\
 & \quad = F[\!(\beta)\!]_{F_A} \circ \chi[out_{F_A} \mapsto \beta] \circ F_A^d \beta \circ \cdots \circ F_A^1 \; \beta \circ \beta \\
\equiv & \quad \{ \text{ Functor } F_A \text{ } \} \\
 & \chi \circ F_A^d out_{F_A} \circ \cdots \circ F_A^1 \; (out_{F_A} \circ [\!(\beta)\!]_{F_A}) \circ \beta \\
 & \quad = F[\!(\beta)\!]_{F_A} \circ \chi[out_{F_A} \mapsto \beta] \circ F_A^d \beta \circ \cdots \circ F_A^1 \; \beta \circ \beta \\
\equiv & \quad \{ \text{ Repeat the above transformation } \} \\
 & \chi \circ F_A^{d+1} [\!(\beta)\!]_{F_A} \circ F_A^d \beta \circ \cdots \circ F_A^1 \; \beta \circ \beta \\
 & \quad = F[\!(\beta)\!]_{F_A} \circ \chi[out_{F_A} \mapsto \beta] \circ F_A^d \beta \circ \cdots \circ F_A^1 \; \beta \circ \beta \\
\equiv & \quad \{ \text{ Property of } \chi \text{ } \} \\
 & \textit{True.}
\end{array}$$

[1]Here $F_A^m = F_A^{m-1} \circ F_A$.

$$
\begin{array}{l}
\mathcal{S}[\![[\![\phi_1 \triangledown \cdots \triangledown \phi_n, \eta, \psi]\!]_{G,F}]\!] = [\![\phi'_1 \triangledown \cdots \triangledown \phi'_n, (\eta_{\phi_1} + \cdots + \eta_{\phi_n}) \circ \eta, \psi]\!]_{G',F} \\
\quad \text{where } \phi'_i = \lambda(u_{i_1}, \cdots, u_{i_m}, v'_{i_1}, \cdots, v'_{i_{l_i}}).\ t'_i \\
\qquad\qquad \eta_{\phi_i} = \lambda(v_{i_1}, \cdots, v_{i_{k_i}}, v'_{i_1}, \cdots, v'_{i_{l_i}}).\ (t_{i_1}, \cdots, t_{i_{m_i}}, v'_{i_1}, \cdots, v'_{i_{l_i}}) \\
\qquad\qquad G' = G'_1 + \cdots + G'_n \\
\qquad\qquad G_i = !\mathbf{1}, \quad \text{if } \ m_i = l_i = 0 \\
\qquad\qquad\quad = !\Gamma(u_{i_1}) \times \cdots \times !\Gamma(u_{i_{m_i}}) \times I_1 \times \cdots \times I_{l_i}, \quad (I_1 = \cdots = I_{l_i} = I), \quad \text{otherwise} \\
\qquad\qquad \text{where } \ \lambda(v_{i_1}, \cdots, v_{i_{k_i}}, v'_{i_1}, \cdots, v'_{i_{l_i}}).\ t_i = \phi_i, \ \ (\text{assume } v'_{i_1}, \cdots, v'_{i_{l_i}} \text{ are recursive variables}) \\
\qquad\qquad\qquad (\{(u_{i_1}, t_{i_1}), \cdots, (u_{i_{m_i}}, t_{i_{m_i}})\}, t'_i) = \mathcal{E}[\![t_i]\!]\ \{v'_{i_1}, \cdots, v'_{i_{l_i}}\}
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{E}[\![v]\!]\ s_r & = & \text{if } v \in s_r \text{ then } (\{\}, v) \text{ else } (\{(u,v)\}, u) \\
 & & \quad \text{where } u \text{ is a fresh variable} \\
\mathcal{E}[\![(t_1, \cdots, t_n)]\!]\ s_r & = & \text{if } \forall i,\ \mathit{Var}_{s_r}(t_i) \text{ then } (\{(u, (t_1, \cdots, t_n))\}, u) \text{ else } (w_1 \cup \cdots \cup w_n, (t'_1, \cdots, t'_n)) \\
 & & \quad \text{where } (w_i, t'_i) = \mathcal{E}[\![t_i]\!]\ s_r \ (i = 1, \cdots, n),\ u \text{ is a fresh variable} \\
\mathcal{E}[\![C\,t]\!]\ s_r & = & \text{if } \mathit{Var}_{s_r}(t') \text{ then } (\{(u, C\,t)\}, u) \text{ else } (w, C\,t') \\
 & & \quad \text{where } (w, t') = \mathcal{E}[\![t]\!]\ s_r,\ u \text{ is a fresh variable} \\
\mathcal{E}[\![v\,t]\!]\ s_r & = & \text{if } \mathit{Var}_{s_r}(t'_v) \wedge \mathit{Var}_{s_r}(t'_t) \text{ then } (\{(u, v\,t)\}, u) \text{ else } (w_v \cup w_t, t'_v\ t'_t) \\
 & & \quad \text{where } (w_v, t'_v) = \mathcal{E}[\![v]\!]\ s_r,\ (w_t, t'_t) = \mathcal{E}[\![t]\!]\ s_r,\ u \text{ is a fresh variable} \\
 & & \\
\mathit{Var}_{s_r}(t) & = & t \text{ is a variable } \wedge \ t \notin s_r
\end{array}
$$

Figure 4.1: Algorithm for Structuring Hylomorphisms

As a concrete example, consider the hylomorphism we derived for $\mathit{foldr}1\ \oplus$ in Section 3.2.3. We can derive a polymorphic function

$$\sigma = \lambda\beta.\ (\iota_1 \triangledown (\iota_2 \circ \pi_1 \triangledown \iota_3 \circ (\mathit{id} \times \beta^{-1} \circ \iota_2)) \circ \mathit{dist}) \circ F_{L_A}\,\beta \circ \beta$$

from its ψ such that $\psi = \sigma\,\mathit{out}_{F_{L_B}}$, where β^{-1} can be think of as a "folding" of β and dist is a natural transformation defined as follows.

$$
\begin{array}{lcl}
\mathit{dist} & : & X \times (Y + Z) \to X \times Y + X \times Z \\
\mathit{dist}(x, (1, y)) & = & (1, (x, y)) \\
\mathit{dist}(x, (2, z)) & = & (2, (x, y))
\end{array}
$$

Therefore, we obtain

$$\mathit{foldr}1\ \oplus = [\![\mathit{error} \triangledown \mathit{id} \triangledown \oplus, \mathit{id}, \sigma\,\mathit{out}_{F_{L_B}}]\!]_{F,F}.$$

4.3 Algorithm for Structuring Hylomorphisms

Generally, the behavior of a hylomorphism $[\![\phi, \eta, \psi]\!]_{G,F}$ could be understood as follows: ψ generates a recursive structure, η operates on the elements of the structure, and ϕ

manipulates on the recursive structure. It is possible for ϕ and ψ to have the computation that η can do. They are said to have the *least computation* if they do not have computation that η can do.

A hylomorphism $[\![\phi, \eta, \psi]\!]$ is said to be *structural* if ϕ and ψ contain the least computation. Structural hylomorphisms are fit for the two fusion theorems. For the Acid Rain Theorem, it is suitable since ϕ and ψ are simple. For the Hylo Fusion Theorem, since ϕ (ψ) contain the least computation, it implies that $\eta \circ \psi$ ($\phi \circ \eta$) contains the most computation and so $[\![\phi, id, \eta \circ \psi]\!]$ ($[\![\phi \circ \eta, id, \psi]\!]$) is suitable for the Right (Left) Fusion Law. In other words, once a structural hylomorphism is got, we can "shift" the natural transformation (η) freely inside the hylomorphism according to which Fusion Law is to be applied.

Our algorithm for structuring any given hylomorphism $[\![\phi, \eta, \psi]\!]_{G,F}$ is to shift computations from ϕ and ψ into η by factoring ϕ to $\phi' \circ \eta_\phi$ and ψ to $\eta_\psi \circ \psi'$ so that ϕ' and ψ' contain the least computation, resulting in a structural hylomorphism $[\![\phi', \eta_\phi \circ \eta \circ \eta_\psi, \psi']\!]_{G',F'}$. In the following, we give the algorithm for factoring ϕ to $\phi' \circ \eta_\phi$, while omitting the dual discussion on ψ.

Let $\phi : G\,A \to A$ be given as:

$$\phi = \phi_1 \triangledown \cdots \triangledown \phi_n$$

where $G = G_1 + \cdots + G_n$ and $\phi_i : G_i\,A \to A$. A typical ϕ_i, as in algorithm $\mathcal{A}$, is defined by:

$$\phi_i = \lambda(v_{i_1}, \cdots, v_{i_{k_i}}, v'_{i_1}, \cdots, v'_{i_{l_i}}).\ t_i$$

The variables with type A are explicitly attached with a prime ($'$) and called *recursive variables*.

In order to capture the computations in ϕ_i which can be done by a natural transformation, we define *maximal non-recursive subterms* as follows.

Definition 13 (Maximal Non-Recursive Subterm)
A term t_{i_j} is said to be a *non-recursive subterm* of t_i if (1) t_{i_j} is a subterm of t_i; (2) t_{i_j} does not include recursive variables. A non-recursive subterm is said to be *maximal* if it is not a subterm of other non-recursive subterms. □

The essence of our algorithm is to factor ϕ_i into $\phi'_i \circ \eta_{\phi_i}$ so that all the maximal non-recursive subterms in t_i are shifted into η_{ϕ_i}. Informally, the algorithm is as follows.

1. Find all the maximal non-recursive subterms in t_i, say

$$t_{i_1},\ \ldots,\ t_{i_{m_i}}.$$

2. Let t'_i be the term from t_i with each maximal non-recursive subterm t_{i_j} be replaced by a new variable u_{i_j}, i.e.,

$$t'_i = t_i[t_{i_1} \mapsto u_{i_1}, \cdots, t_{i_{m_i}} \mapsto u_{i_{m_i}}].$$

3. Factor ϕ_i by extracting all the maximal non-recursive subterms out of t_i as follows.

$$\begin{array}{l} \phi_i = \phi'_i \circ \eta_{\phi_i} \\ \quad \text{where } \phi'_i = \lambda(u_{i_1}, \cdots, u_{i_m}, v'_{i_1}, \cdots, v'_{i_{l_i}}).\ t'_i \\ \qquad\quad \eta_{\phi_i} = \lambda(v_{i_1}, \cdots, v_{i_{k_i}}, v'_{i_1}, \cdots, v'_{i_{l_i}}). \\ \qquad\qquad\qquad (t_{i_1}, \cdots, t_{i_{m_i}}, v'_{i_1}, \cdots, v'_{i_{l_i}}) \end{array}$$

4. Factor ϕ by grouping the result of ϕ_i, i.e.,

$$\phi = (\phi'_1 \triangledown \cdots \triangledown \phi'_n) \circ (\eta_{\phi_1} + \cdots + \eta_{\phi_n}).$$

The above algorithm is summarized in Figure 4.1. The main transformation is $\mathcal{S}$ which in turn calls transformation $\mathcal{E}$ to process every term t_i for factoring ϕ_i. Similar to the algorithm in Figure 3.2, a fresh variable is allocated every time a non-recursive subterm is found. It will be discarded when the corresponding non-recursive subterm turns out to be non-maximal by the predicate *Var*. The correctness of the algorithm S is omitted, but it should be noted that the type of $\phi'_1 \triangledown \cdots \triangledown \phi'_n$ is $G'\,A \to A$ and $\eta_{\phi_1} + \cdots + \eta_{\phi_n}$ is a natural transformation from G to G'.

Theorem 11 The $\eta_{\phi_1} + \cdots + \eta_{\phi_n}$, derived in the algorithm $\mathcal{S}$, is a natural transformation from G to G', i.e.,

$$\eta_{\phi_1} + \cdots + \eta_{\phi_n} : G \dot{\to} G'.$$

Proof: We prove it by the following calculation.

$$\begin{array}{ll} & \eta_{\phi_1} + \cdots + \eta_{\phi_n} : G \dot{\to} G' \\ \equiv & \quad \{ \text{ Definition of natural transformation, for any } f \ \} \\ & (\eta_{\phi_1} + \cdots + \eta_{\phi_n}) \circ G\,f = G' f \circ (\eta_{\phi_1} + \cdots + \eta_{\phi_n}) \\ \equiv & \quad \{ \text{ Definitions of } G \text{ and } G' \ \} \\ & \eta_{\phi_1} \circ G_1 f + \cdots + \eta_{\phi_n} \circ G_n f = \\ & \qquad G'_1 f \circ \eta_{\phi_1} + \cdots + G'_n f \circ \eta_{\phi_n} \\ \Leftarrow & \quad \{ \text{ trivial } \} \\ & \eta_{\phi_i} \circ G_i f = G'_i f \circ \eta_{\phi_i}, \quad (i = 1, \cdots, n) \\ \equiv & \quad \{ \text{ Definitions of } \eta_{\phi_i}, G_i \text{ and } G'_i \ \} \\ & \lambda(v_{i_1}, \cdots, v_{i_{k_i}}, v'_{i_1}, \cdots, v'_{i_{l_i}}). \\ & \qquad (t_{i_1}, \cdots, t_{i_{m_i}}, f\,v'_{i_1}, \cdots, f\,v'_{i_{l_i}}) = \\ & \quad \lambda(v_{i_1}, \cdots, v_{i_{k_i}}, v'_{i_1}, \cdots, v'_{i_{l_i}}). \\ & \qquad (t_{i_1}, \cdots, t_{i_{m_i}}, f\,v'_{i_1}, \cdots, f\,v'_{i_{l_i}}), \quad (i = 1, \cdots, n) \\ \equiv & \quad \{ \text{ obvious } \} \\ & \mathit{True} \end{array}$$

□

As an example, let's structure the following hylomorphism obtained in Section 3.2.3 where $g : A \to B$.

$$map\ g = [\![Nil \triangledown \lambda(a, v_1').Cons\,(g\,a, v_1'), id, out_{F_{L_A}}]\!]_{F_{L_A}, F_{L_A}}$$

In this case, $\phi_1 = \lambda().Nil$ and $\phi_2 = \lambda(a, v_1').Cons\,(\underline{g\,a}, v_1')$. Here there is only one maximal non-recursive subterm as underlined. Our algorithm will move $g\,a$ out of ϕ_2 as

$$\begin{aligned} \phi_2 &= \phi_2' \circ \eta_{\phi_2} \\ &\quad \text{where } \phi_2' = \lambda(u_{2_1}, v_1').\ Cons(u_{2_1}, v_1') \\ &\quad\quad\quad\ \ \eta_{\phi_2} = \lambda(a, v_1').\ (g\,a, v_1') \end{aligned}$$

and finally give the following structural hylomorphism:

$$\begin{aligned} map\ g = [\![& Nil \triangledown \lambda(u_{2_1}, v_1').\ Cons(u_{2_1}, v_1'), \\ & id + \lambda(a, v_1').(g\,a, v_1'), out_{F_{L_A}}]\!]_{F_{L_B}, F_{L_A}}. \end{aligned}$$

Moreover, with Theorem 10, we can get $\phi' = \alpha$ and $\tau = id$. So the above hylomorphism becomes

$$[\![in_{F_{L_B}}, id + \lambda(a, v_1').(g\,a, v_1'), out_{F_{L_A}}]\!]_{F_{L_B}, F_{L_A}}$$

a structural hylomorphism for the two fusion theorems.

4.4 Examples

In this section, we use several examples to show how to calculate fusion transformation over recursions so that efficient functional programs can be obtained. There examples would be difficult to be dealt with without our algorithms for the derivation of structural hylomorphisms.

4.4.1 Fusion with the Acid Rain Theorem

In the following, we given an example of fusion transformation over structural hylomorphisms. Consider the program

$$foldr1\ \oplus\ \circ\ map\ g$$

which accepts a list

$$Cons\,(x_1, \cdots, Cons\,(x_{n-1}, Cons(x_n, Nil)) \cdots)$$

and returns

$$g\,x_1 \oplus (\cdots (g\,x_{n-1} \oplus g\,x_n) \cdots).$$

Since $foldr1\ \oplus$ is not a catamorphism (Section 3.2.3), the algorithm in [LS95] will fail and leave the intermediate data structure produced by $map\ g$ remained. Our algorithm can solve this problem. With the resulting hylomorphisms obtained in Section 4.2.2 and the above, we have

$$\begin{aligned}&[\![error \triangledown id \triangledown \oplus, id, \sigma\, out_{F_{L_B}}]\!]_{F,F} \circ \\ &\quad [\![in_{F_{L_B}}, id + \lambda(a, v_1').(g\, a, v_1'), out_{F_{L_A}}]\!]_{F_{L_B}, F_{L_A}}.\end{aligned}$$

We shall demonstrate how the Ana-Hylo Fusion Law is applied to eliminate the intermediate data structure in the program $foldr1 \oplus \circ map\ g$.

$$\begin{aligned}
& foldr1 \oplus \circ map\ g \\
= & \quad \{ \text{ Replace } foldr1 \oplus \text{ and } map\ g \text{ with their hylos } \} \\
& [\![error \triangledown id \triangledown \oplus, id, \sigma\, out_{F_{L_B}}]\!]_{F,F} \circ \\
& \quad [\![in_{F_{L_B}}, id + g \times id, out_{F_{L_A}}]\!]_{F_{L_B}, F_{L_A}} \\
= & \quad \{ \text{ Acid Rain Theorem (Hylo-Ana Fusion Law) } \} \\
& [\![error \triangledown id \triangledown \oplus, id, \sigma((id + g \times id) \circ out_{F_{L_A}})]\!]_{F,F}
\end{aligned}$$

Now we focus on the transformation of the third part in the above hylomorphism.

$$\begin{aligned}
& \sigma((id + g \times id) \circ out_{F_{L_A}}) \\
= & \quad \{ \text{ Definition of } \sigma \text{ (Section 4.2.2), let } h = id + g \times id \ \} \\
& (\iota_1 \triangledown (\iota_2 \circ \pi_1 \triangledown \iota_3 \circ (id \times ((h \circ out_{F_{L_A}})^{-1} \circ \iota_2))) \circ dist) \\
& \quad \circ F_{L_A} (h \circ out_{F_{L_A}}) \circ h \circ out_{F_{L_A}} \\
= & \quad \{ \text{ Definition of } F_{L_A}, \text{ and } h. \ \} \\
& (\iota_1 \triangledown (\iota_2 \circ \pi_1 \triangledown \iota_3 \circ (id \times ((h \circ out_{F_{L_A}})^{-1} \circ \iota_2))) \circ dist) \\
& \quad \circ (id + (g \times (h \circ out_{F_{L_A}}))) \circ out_{F_{L_A}} \\
= & \quad \{ \text{ Definition of } F_{L_A} \ \} \\
& (\iota_1 \triangledown (\iota_2 \circ \pi_1 \triangledown \iota_3 \circ (id \times ((h \circ out_{F_{L_A}})^{-1} \circ \iota_2))) \circ dist) \\
& \quad \circ (id + g \times h) \circ F_{L_A} out_{F_{L_A}} \circ out_{F_{L_A}} \\
= & \quad \{ \text{ Definition of } h \ \} \\
& (\iota_1 \triangledown (\iota_2 \circ \pi_1 \triangledown \iota_3 \circ (id \times ((h \circ out_{F_{L_A}})^{-1} \circ \iota_2))) \circ dist) \\
& \quad \circ (id + g \times (id + g \times id)) \circ F_{L_A} out_{F_{L_A}} \circ out_{F_{L_A}} \\
= & \quad \{ \text{ Move } dist \text{ backwards } \} \\
& (\iota_1 \triangledown (\iota_2 \circ \pi_1 \triangledown \iota_3 \circ (id \times ((h \circ out_{F_{L_A}})^{-1} \circ \iota_2)))) \\
& \quad \circ (id + (dist \circ (g \times (id + g \times id)))) \\
& \qquad \circ F_{L_A} out_{F_{L_A}} \circ out_{F_{L_A}} \\
= & \quad \{ \text{ Transformation property of } dist \ \} \\
& (\iota_1 \triangledown (\iota_2 \circ \pi_1 \triangledown \iota_3 \circ (id \times ((h \circ out_{F_{L_A}})^{-1} \circ \iota_2)))) \\
& \quad \circ (id + (g \times id + g \times (g \times id)) \circ dist) \\
& \qquad \circ F_{L_A} out_{F_{L_A}} \circ out_{F_{L_A}} \\
= & \quad \{ \ (f \triangledown g) \circ (p + q) = f \circ p \triangledown g \circ q,\ out_{F_{L_A}}^{-1} = in_{F_{L_A}} \ \} \\
& (\iota_1 \triangledown (\iota_2 \circ g \circ \pi_1 \\
& \qquad \triangledown \iota_3 \circ (g \times (in_{F_{L_A}} \circ h^{-1} \circ \iota_2 \circ (g \times id)))) \circ dist) \\
& \qquad \circ F_{L_A} out_{F_{L_A}} \circ out_{F_{L_A}} \\
= & \quad \{ \text{ Since } h^{-1} \circ \iota_2 = \iota_2 \circ (g \times id)^{-1} \ \} \\
& (\iota_1 \triangledown (\iota_2 \circ g \circ \pi_1 \triangledown \iota_3 \circ (g \times in_{F_{L_A}} \circ \iota_2)) \circ dist) \\
& \qquad \circ F_{L_A} out_{F_{L_A}} \circ out_{F_{L_A}}
\end{aligned}$$

It then derived the following hylomorphism:

$$\begin{array}{l}[\![\mathit{error} \triangledown \mathit{id} \triangledown \oplus, \mathit{id},\\ \quad (\iota_1 \triangledown (\iota_2 \circ g \circ \pi_1 \triangledown \iota_3 \circ (g \times (\mathit{in}_{F_{L_A}} \circ \iota_2))) \circ \mathit{dist}) \circ \\ \quad F_{L_A}\, \mathit{out}_{F_{L_A}} \circ \mathit{out}_{F_{L_A}}]\!]_{F,F}.\end{array}$$

Inlining the above derived hylomorphism, named *prg*, would give the following familiar program:

$$\begin{array}{ll} \mathit{prg} = \lambda xs.\ \text{case}\ xs\ \text{of} & \\ \qquad \mathit{Nil} \to \mathit{error}\,; & \\ \qquad \mathit{Cons}\,(a, \mathit{Nil}) \to g\ a & \\ \qquad \mathit{Cons}\,(a, as) \to g\ a \oplus \mathit{prg}\ as & \end{array}$$

where the intermediate data structure produced by $\mathit{map}\, g$ no longer exists. Compared with Launchbury and Sheard's algorithm [LS95], this example indicates that ours is more general and powerful.

4.4.2 Fusion with Recursions over Multiple Data Structures

Let's see how to remove the intermediate data structure in the program:

$$\mathit{spec} = \mathit{length} \circ \mathit{zip}$$

where $\mathit{length} = (\![Z \triangledown \lambda(x,p).(S\ p)]\!)_N$ and *zip* is a hylomorphism (Section 3.2.3):

$$\begin{array}{l} \mathit{zip} = [\![\mathit{Nil} \triangledown \mathit{Nil} \triangledown \lambda(a,b,p).\mathit{Cons}((a,b),p), \mathit{id}, \psi]\!]_{F,F} \\ \quad \text{where } F = !\mathbf{1} + !\mathbf{1} + !A \times !B \times I \\ \qquad\quad \psi = \lambda(xs, ys).\ \text{case}\ (xs, ys)\ \text{of} \\ \qquad\qquad\qquad (\mathit{Nil}, \_) \to (1, ()); \\ \qquad\qquad\qquad (\mathit{Cons}\,(a, as), \mathit{Nil}) \to (2, ()); \\ \qquad\qquad\qquad (\mathit{Cons}\,(a, as), \mathit{Cons}(b, bs)) \to (3, (a, b, (as, bs))) \end{array}$$

The same example has also appeared in [FST94], where a special promotion transformation theorem was given for recursions over multiple inductive structures. With our approach, the special theorem turns out to be unnecessary. We would like to use Theorem 7 for this fusion. As is not difficult to see that that ϕ part in the hylomorphism for *zip* contains maximal information for other function fused with it from the left, we do not need to restructure it for this example. Now, we calculate it as follows. Since

$$\begin{array}{ll} & \mathit{length} \circ (\mathit{Nil} \triangledown \mathit{Nil} \triangledown \lambda(a,b,p).\mathit{Cons}((a,b),p)) \\ = & \quad \{\ \triangledown\ \} \\ & \mathit{length} \circ \mathit{Nil} \triangledown \mathit{length} \circ (\lambda(a,b,p).\mathit{Cons}((a,b),p)) \\ = & \quad \{\ \text{Def. of } \mathit{length}\ \} \\ & Z \triangledown \lambda(a,b,p).1 + \mathit{length}\, p \\ = & \quad \{\ \text{Define } \phi' = Z \triangledown \lambda(a,b,p).1 + p\ \} \\ & \phi' \circ F\, \mathit{length} \end{array}$$

Then

$$\begin{array}{ll} & \mathit{spec} \\ = & \quad \{ \text{ Def. of } \mathit{spec} \text{ } \} \\ & \mathit{length} \circ \mathit{zip} \\ = & \quad \{ \text{ Theorem 7 } \} \\ & [\![\phi', \mathit{id}, \psi]\!]_{F,F} \end{array}$$

Now the intermediate data structure passed between *length* and *zip* has been successfully eliminated. Obviously, our approach is more systematic than Fegaras' [FSZ94].

4.5 Conclusion

In this chapter, we propose an algorithm for restructuring hylomorphisms so that the Hylo Fusion and Acid Rain Theorems (Section 2.2) can be effectively applied. Particularly, we propose a new theorem for deriving polymorphic functions (Section 4.2.2), namely τ and σ, for the use of Acid Rain Theorem.

Chapter 5

Calculating Tupling Transformation

As explained in Chapter 1, a functional program *prog* is usually expressed as compositions of transformations over data structures while each transformation is defined by a recursion $\mathcal{R}_i$ traversing over its input data structure, namely

$$prog = \mathcal{R}_1 \circ \cdots \circ \mathcal{R}_n.$$

This compositional style of programming allows clearer and more modular programs, but comes at a price of possibly high runtime overhead resulting mainly from the following two categories: (1) Unnecessary intermediate data structures passed between the composition of two recursions; (2) Inefficiency in a single recursion, such as redundant recursive calls, multiple traversals of data structures, and unnecessary traversals of intermediate data structures (see Section 5.1). Although this chapter is mainly concerned with elimination of the inefficiency in the latter case, these two kinds of inefficiency are much related each other. There are two known tactics, namely *Fusion* (or called *deforestation)* [Wad88, Chi92] and *Tupling* [Chi93] for elimination of these inefficiencies. Fusion, as discussed in Chapter 4, is to merge nested compositions of recursive functions in order to obtain new recursions without unnecessary intermediate data structures, while tupling is to remove redundant recursive calls and multiple traversals of the same data structure from recursions.

The previous approaches to *tupling*, as extensively studied by Chin [Chi92, Chi93], are basically based on the so-called *fold/unfold transformation*[BD77], which have to keep track of all function calls occurred previously and suitably introduce a definition of recursive function on detecting a repetition. This process of keeping track of function calls and the clever control to avoid infinite unfolding introduces substantial cost and complexity, which oppose an obstacle to adopt tupling as part of the regular optimizations in any serious compilers of functional languages.

In this chapter, we aims at an attempt to adapt the calculational approach to the tupling transformation for the improvement of recursive functions. We believe it is worth doing for two reasons. First, tupling and fusion are two most related transformation tactics, so they should be studied in the same framework. In fact, the roles of tupling and fusion are complementary; fusion merges compositions of recursions into one which then should be improved again by tupling in order to obtain a final efficient program. Second, with the same reason for the shortcut deforestation [GLJ93, TM95], tupling should be used practically in a real compiler. So far, we have not seen a real practical compiler which performs tupling transformation.

In this chapter, we propose an idea of *cheap tupling*[1] in a calculational way. Our main contributions are as follows.

- We identify the importance of the relationship between tupling transformation and *structural* mutual recursions (not a simple mutual recursion) in Section 5.2, based on which we propose three simple but effective calculational rules (Theorem 12, 16 and 18) for our cheap tupling transformation. As will be seen, they can be applied to improve a wide class of recursions (though not all) through the elimination of redundant recursive calls, multiple traversals of the same data structures, and unnecessary traversals of data structures which has not been noticed before.
- As discussed above, our cheap tupling follows the approach of transformation in calculational form based on Constructive Algorithmics. This is in sharp contrast to the previous study [Chi93] based on fold/unfold transformation. Therefore, our cheap tupling preserves the advantages of transformation in calculational form, as we have seen in the discussion of shortcut deforestation in [TM95].
  - It can be applied to the improvement of recursions over any data structures as well as recursions over lists.
  - Each of our transformation rules is an automation of the unfold-simplify-fold method without intervention of explicit laws. Therefore, our transformations are guaranteed to terminate and would be more practical to be used in a compiler.
- Our cheap tupling is suitable to coexist with the shortcut deforestation (in calculational form). It will be seen that the combination of the two improvements is direct and natural (Section 5.3). In contrast, the previous study on this combination based on fold/unfold transformation [Chi95] is much more difficult because of complicated control of infinite unfoldings in the case where fusion and tupling are applied simultaneously.

The organization of this chapter is as follows. In Section 5.1, we use several simple examples to show what kind of inefficient recursions we would like to improve. Based on the concept of the transformation in calculational form as in Section 2.2, we propose

[1]We call it *cheap* tupling after the name of *shortcut* deforestation.

three simple but effective rules for our cheap tupling in Section 5.2. In Section 5.3, we give the cheap tupling strategy based our simple rules, and see how cheap tupling can be coexisted with fusion transformation. Related works and conclusions are discussed in Section 5.4 and 5.5 respectively.

5.1 Examples: Inefficient Recursions

In this Section, we shall use several simple examples to show what kind of inefficiency may occur in a recursion, and how an efficient version can be derived. As will be seen later, the efficiency is always achieved at the cost of clarity and conciseness compared with the original one. We would like to write our programs concisely, but have the compiler automatically make them efficient.

5.1.1 Multiple Traversals of Data Structures

Consider the function *deepest*, which finds a list of leaves that are farthest away from the root of a given tree, may be defined as follows.

$$\begin{array}{lcl}
deepest\ (Leaf\ a) & = & [a] \\
deepest\ (Node(l,r)) & = & deepest(l), \quad depth(l) > depth(r) \\
 & = & deepest(l) +\!\!+ deepest(r), \quad depth(l) = depth(r) \\
 & = & deepest(r), \quad \text{otherwise} \\
depth\ (Leaf\ a) & = & 1 \\
depth\ (Node(l,r)) & = & 1 + max(depth(l), depth(r))
\end{array}$$

The infix binary function $+\!\!+$ concatenates two lists and the function *max* gives the maximum of the two arguments. Function *deepest* uses another recursive function *depth*. Being concise, this definition is quite inefficient because *deepest* and *depth* traverse over the same inputs leading to many repeated computations in calculating the depth of subtrees. We would like to eliminate this multiple traversal and have the following efficient program.

$$\begin{array}{lcl}
deepest'\ t & = & u \text{ where } (u,v) = dd\ t \\
dd\ (Leaf\ a) & = & ([a], 0) \\
dd\ (Node\ l\ r) & = & (dpl, 1 + dl), \quad dl > dr \\
 & = & (dpl +\!\!+ dpr, 1 + dr), \quad dl = dr \\
 & = & (dpr, 1 + dr), \quad \text{otherwise} \\
 & & \text{where } (dpl, dl) = dd\ l;\ (dpr, dr) = dd\ r
\end{array}$$

5.1.2 Redundant Recursive Calls

A classical example to illustrate the super-linear speedup achieved when redundant recursive calls are eliminated is the fibonacci function:

$$\begin{array}{lcl} \mathit{fib}\ Z & = & Z \\ \mathit{fib}\ (S(Z)) & = & S(Z) \\ \mathit{fib}\ (S(S(n))) & = & \mathit{fib}(S(n)) + \mathit{fib}(n) \end{array}$$

The *fib* is a recursion over the natural number data type defined in Example 3 in Chapter 2. This definition gives an inefficient exponential algorithm *fib* because of many redundant recursive calls to *fib*. In previous studies, in order to make it efficient, a creative tupling function $f'(S(n)) = (\mathit{fib}(S(n)), \mathit{fib}(n))$ has to be defined, and then the transformation based on fold/unfold is applied to improve f' for sharing common computation and remove redundant recursive calls. We would like to show that the redundant recursive calls can be removed by a simple calculation with our approach (Section 5.2).

5.1.3 Unnecessary Traversals of Intermediate Results

Given a recursion $\mathcal{R}$, we have known that unnecessary intermediate data structures, produced by $\mathcal{R}$, occur when composed with another recursion $\mathcal{R}'$, i.e., $\mathcal{R}' \circ \mathcal{R}$. By fusion, one can merge the two recursions into one without such unnecessary intermediate data structures.

It would be surprising to see that even in a single recursion, there may remain some unnecessary traversals of intermediate data structures produced by $\mathcal{R}$. As an example, consider the recursive function *foo* recursively defined by

$$\begin{array}{lcl} \mathit{foo}\ \mathit{Nil} & = & \mathit{Nil} \\ \mathit{foo}\ (\mathit{Cons}(x, xs)) & = & \mathit{Cons}(x + \mathit{sum}\ p, p) \\ & & \text{where } p = \mathit{foo}\ xs \end{array}$$

where *sum* is used to sum up all the elements in a list. Although *foo* is defined in a single recursion, it is inefficient because the intermediate results produced by *foo* is traversed by *sum*, which is actually unnecessary. We could eliminate this unnecessary traversal and obtain the following efficient one.

$$\begin{array}{lcl} \mathit{foo}\ xs & = & i, \quad \text{where } (i, s) = f'\ xs \\ f'\ \mathit{Nil} & = & (\mathit{Nil}, 0) \\ f'\ (\mathit{Cons}(x, xs)) & = & (\mathit{Cons}(x + s,\ i),\ x + s + s) \\ & & \text{where } (i, s) = f'\ xs \end{array}$$

5.2 Cheap Tupling Rules in Calculational Form

Tupling achieves efficient recursive functions through elimination of redundant recursive calls and multiple traversals of common inputs. In this section, we propose three simple but effective calculational rules for our cheap tupling.

We shall start by examing the relationship between tupling and mutual recursive definitions and propose our *basic* calculational rule (Theorem 12) on how to perform tupling transformation on mutual recursions. Based on it, we give another two calculational rules for removing redundant recursive calls and unnecessary intermediate data structures respectively. We shall also demonstrate how the rules work practically.

Mutual Recursions and Tupling

There has been a folklore that mutual recursive definitions can be turned into a single non-mutual recursive definition if the functions mutually defined are tupled. Take as an example the following mutual recursive definitions:

$$\begin{array}{rcl} f & = & C_1[g,\, f] \\ g & = & C_2[f,\, g] \end{array}$$

where C_i denotes a context. It says that f calls g and f, and g calls f and g too. One can turn this mutual recursion into a single non-mutual one by tupling the functions f and g to another function h:

$$h = f \vartriangle g.$$

It follows that

$$\begin{array}{rcl} f & = & \pi_1 \circ h \\ g & = & \pi_2 \circ h \end{array}$$

and that h becomes a single non-mutual recursion as follows.

$$h = C_1[\pi_2 \circ h,\, \pi_1 \circ h] \vartriangle C_2[\pi_1 \circ h,\, \pi_2 \circ h]$$

This transformation is interesting in theory in the sense that one need not consider the transformation for mutual recursive functions because they can definitely be turned into a single non-mutual one. It is, however, unsatisfactory in practice as it is not clear how to make h efficient. Generally, the new definition h costs more than the direct implementation of mutual recursions if no further simplification is applied. This is why many compilers, such as Glasgow Haskell, implement mutual recursions directly without such transformation.

Things become expected when the recursive structure information of f and g are known. Such structural information can be exploited to make the tupled function efficient, as shown in the following theorem.

Theorem 12 (Tupling)

$$\frac{f \circ in_F = \phi \circ F(f \vartriangle g),\ \ g \circ in_F = \psi \circ F(f \vartriangle g)}{f \vartriangle g = (\!|\phi \vartriangle \psi|\!)_F}$$

Proof: We prove it by the following calculation.

$$\begin{array}{ll} & (f \vartriangle g) \circ in_F \\ = & \quad \{\ \vartriangle\ \} \\ & f \circ in_F \vartriangle g \circ in_F \\ = & \quad \{\ \text{Assumptions for } f \text{ and } g\ \} \\ & \phi \circ F(f \vartriangle g) \vartriangle \psi \circ F(f \vartriangle g) \\ = & \quad \{\ \vartriangle\ \} \\ & (\phi \vartriangle \psi) \circ F(f \vartriangle g) \end{array}$$

It soon follows that

$$f \vartriangle g = (\![\phi \vartriangle \psi]\!)_F$$

according to the definition of catamorphisms (Theorem 5). □

A simple generalization of the Tupling Theorem from one step of unfolding of input to n step is given as follows.

Corollary 13 Given the following mutual recursive definitions for f and g:

$$\begin{array}{lcl} f \circ in_F & = & \phi \circ (F(f \vartriangle g) \vartriangle F^2(f \vartriangle g) \circ out_F \vartriangle \cdots \vartriangle F^n(f \vartriangle g) \circ out_F^{n-1}) \\ g \circ in_F & = & \psi \circ (F(f \vartriangle g) \vartriangle F^2(f \vartriangle g) \circ out_F \vartriangle \cdots \vartriangle F^n(f \vartriangle g) \circ out_F^{n-1}) \end{array}$$

then f and g can be tupled as

$$(f \vartriangle g) \circ in_F = (\phi \vartriangle \psi) \circ (F(f \vartriangle g) \vartriangle F^2(f \vartriangle g) \circ out_F \vartriangle \cdots \vartriangle F^n(f \vartriangle g) \circ out_F^{n-1})$$

where $F^n = F^{n-1} \circ F$ and $out_F^n = F^{n-1} out_F \circ out_F^{n-1}$. □

The Tupling Theorem is quite simple, and some similar studies can be found in [Tak87, Fok92b]. What interests us is its significant use in calculating recursions to efficient ones, which has not yet received its worthy consideration. In the following, we shall show how this simple calculational rule can be effectively used for our cheap tupling to handle the inefficient recursions in Section 5.1.

5.2.1 Eliminating Multiple Data Traversals

The Tupling Theorem reads that, if f and g are recursive functions *traversing over the same data structures* in a certain uniform way, then tupling them will definitely give a catamorphism without multiple traversals over the same data structures by both f and g. Therefore, direct use of the tupling theorem can help to eliminate multiple data traversals in a recursion.

To see how the Tupling Theorem works, let us recall the definition of *deepest* given in Section 5.1. Since *deepest* and *depth* are mutually defined and traverse over the same input tree, we can apply the Tupling Theorem to calculate them into an efficient one. First, we rewrite them to be our required form.

$$\begin{array}{lcl}
\mathit{deepest} \circ in_{F_T} & = & \phi \circ F_T(\mathit{deepest} \vartriangle \mathit{depth}) \\
& & \text{where} \\
& & \quad \phi = \phi_1 \triangledown \phi_2 \\
& & \quad \phi_1\ a = [a] \\
& & \quad \phi_2\ ((tl, hl), (tr, hr)) \;=\; tl, \quad \text{if } hl > hr \\
& & \qquad\qquad\qquad\qquad\quad\;\; =\; tl +\!\!+ tr, \quad \text{if } hl = hr \\
& & \qquad\qquad\qquad\qquad\quad\;\; =\; tr, \quad \text{otherwise} \\
\mathit{depth} \circ in_{F_T} & = & \psi \circ F_T(\mathit{deepest} \vartriangle \mathit{depth}) \\
& & \text{where} \\
& & \quad \psi = \psi_1 \triangledown \psi_2 \\
& & \quad \psi_1\ a = 1 \\
& & \quad \psi_2\ ((tl, hl), (tr, hr)) = 1 + \mathit{max}(hl, hr)
\end{array}$$

where $F_T = \,!Int + I \times I$ (the functor defining the binary tree type) and $in_{F_T} = \mathit{Leaf} \triangledown \mathit{Node}$. Now, according to the Tupling Theorem, we get the following efficient linear recursion:

$$\mathit{deepest} = \pi_1 \circ (\mathit{deepest} \vartriangle \mathit{depth}) = \pi_1 \circ (\!|\phi \vartriangle \psi|\!)_{F_T}$$

which can be in-lined to the one as given in Section 5.1.

It should be noted that the above two processing steps, namely rewriting into the required form and applying the Tupling Theorem, can be done automatically at a low cost; the first step is basically an abstraction of recursive calls for the definition of ϕ (similar to the study in [HIT96d]) while the second step is just a simple calculation. One should compare with the previous expensive approach on the basis of fold/unfold transformation [Chi92], where it is required to keep alert on any sub-expression which could be folded and to define many new functions in order to remember occurred sub-expressions during its transformation process.

In the Tupling Theorem, f and g are defined mutually. One special interesting case is that they are independent catamorphisms. They can be tupled as well, as stated in the following corollary.

Corollary 14

$$(\!|\phi|\!)_F \vartriangle (\!|\psi|\!)_F = (\!|\phi \circ F\pi_1 \vartriangle \psi \circ F\pi_2|\!)_F$$

Proof. Directly from the Tupling Theorem and the following two equations.

$$\begin{array}{lcl}
(\!|\phi|\!)_F \circ in_F & = & \phi \circ F\pi_1 \circ F((\!|\phi|\!)_F \vartriangle (\!|\psi|\!)_F) \\
(\!|\psi|\!)_F \circ in_F & = & \psi \circ F\pi_2 \circ F((\!|\phi|\!)_F \vartriangle (\!|\psi|\!)_F)
\end{array} \qquad \square$$

This corollary can reduce two traversals of the input (by the two catamorphisms respectively) into one.

5.2.2 Eliminating Redundant Recursive Calls

It is impractical to eliminate all redundant recursive calls in recursions as done by the most general approach called *memoization*[Mic68]. Therefore some restrictions on recursions are necessary. For instance, Chin [Chi93] restricted his method on **T0** class of recursions; Hughes [Hug85] argued that it is more practical to eliminate the redundant recursive calls that are applied to exactly the identical arguments – that is, arguments stored in the same place in memory.

The restriction we impose on the recursive definitions, much related to Hughes' restriction, is that the parameters of the recursive calls to the defined function should be the sub-structure of the input data without any processing. This restriction helps to verify the identity of two parameters at the stage of compilation (rather than seeing if the arguments are stored in the same place in memory at execution time as Hughes did). For example, we can treat the recursive definition like

$$foo(Cons(x_1, Cons(x_2, xs))) = x_1 + foo(Cons(x_2, xs)) + foo(xs)$$

but not

$$foo(Cons(x_1, Cons(x_2, xs))) = x_1 + foo\underline{(Cons(2 * x_2, xs))} + foo\underline{(Cons(x_1, xs))}$$

because the two underlined parameters are not sub-structures of $Cons(x_1, Cons(x_2, xs))$.

Another restriction, just for the sake of simple presentation, is that the input is only traversed by a single function. For example, our restriction excludes the following case

$$foo(Cons(x_1, Cons(x_2, xs))) = x_1 + foo(Cons(x_2, xs)) + foo(xs) + g(x2 : xs)$$

because both foo and g traverse over the same input. As a matter of fact, this restriction is unnecessary because we are always be able to tuple foo and g to have a new function meeting this restriction.

Proposition 15 Let $h :: T \to R$ be a recursively defined function over T, where T is defined by functor F, i.e., $(T, in_F) = \mu F$. If (1) every parameter of all recursive calls to h is the sub-structure of the input; and (2) there is no other function traversing over the part or the whole of h's input, then h can be transformed into the following hylomorphism:

$$h = [\![\phi, id, out_F \vartriangle out_F^2 \vartriangle \cdots \vartriangle out_F^n]\!]_{G,G}$$

where $G = F \times F^2 \times \cdots \times F^n$.

Proof Sketch: According to the restriction of parameters of recursive calls to h, we can see that each recursive call to h can be embedded in the term of $F^i h \circ out_F^i$ for an integer i (Note out_F^i can be considered as unfolding i steps of input data). In addition, since no other functions traversing h's input, it implies that the input is only traversed by h. Thus, there must exist a ϕ so that h can be expressed as

$$h = \phi \circ (Fh \circ out_F \vartriangle F^2 h \circ out_F^2 \vartriangle \cdots \vartriangle F^n h \circ out_F^n)$$

which is exactly as we expected. □

Now, our theorem for removing redundant recursive calls in the above restricted recursive functions is given below.

Theorem 16 Let $h :: T \to R$ be the hylomorphism given in Proposition 15. Then, h can be defined by the following mutual recursive definitions.

$$\begin{array}{lcl} h \circ in_F & = & \phi \circ \eta \circ F(h \vartriangle g_1 \vartriangle \cdots \vartriangle g_{n-1}) \\ g_1 \circ in_F & = & Fh \\ g_2 \circ in_F & = & Fg_1 \\ & \vdots & \\ g_{n-1} \circ in_F & = & Fg_{n-2} \end{array}$$

where η is a natural transformation defined by

$$\begin{array}{lcl} \eta & :: & F(X_1 \times \cdots \times X_n) \to (FX_1 \times \cdots \times FX_n) \\ \eta & = & F\pi_1 \vartriangle \cdots \vartriangle F\pi_n \end{array}$$

Proof: This can be proved by the following calculation.

$$\begin{array}{ll} & h \circ in_F = \phi \circ \eta \circ F(h \vartriangle g_1 \vartriangle \cdots \vartriangle g_{n-1}) \\ \equiv & \quad \{ \text{ unfolding } g_i \text{ one time } \} \\ & h \circ in_F = \phi \circ \eta \circ F(h \vartriangle Fh \circ out_F \vartriangle \cdots \vartriangle Fg_{n-2} \circ out_F) \\ \equiv & \quad \{ \text{ repeat the above unfolding to remove } g_i \} \\ & h \circ in_F = \phi \circ \eta \circ F(h \vartriangle Fh \circ out_F \vartriangle \cdots \vartriangle F^{n-1} h \circ out_F^{n-1}) \\ \equiv & \quad \{ \eta \} \\ & h \circ in_F = \phi \circ (Fh \vartriangle F(Fh \circ out_F) \vartriangle \cdots \vartriangle F(F^{n-1} h \circ out_F^{n-1})) \\ \equiv & \quad \{ \text{ functor } F,\ in_F^{-1} = out_F,\ \vartriangle \text{ and } \circ, \text{ def. of } G \} \\ & h = \phi \circ Gh \circ (out_F \vartriangle Fout_F \circ out_F \vartriangle \cdots \vartriangle Fout_F^{n-1} \circ out_F) \\ \equiv & \quad \{ \ Fout_F^i \circ out_F = out_F^{i+1}, \text{ as proved later. } \} \\ & h = \phi \circ Gh \circ (out_F \vartriangle out_F^2 \vartriangle \cdots \vartriangle out_F^n) \\ \equiv & \quad \{ \text{ def. of hylo } \} \\ & h = [\![\phi, id, out_F \vartriangle out_F^2 \vartriangle \cdots \vartriangle out_F^n]\!]_{G,G} \end{array}$$

To complete the proof, we prove that $Fout_F^i \circ out_F = out_F^{i+1}$ by induction on i. For the base case of $i = 0$, it is obviously true. For the inductive case, we calculate it as follows.

$$\begin{array}{cl} & F\,out_F^i \circ out_F \\ = & \quad \{\ \text{def. of } out_F^i\ \} \\ & F(F^{i-1}out_F \circ out_F^{i-1}) \circ out_F \\ = & \quad \{\ \text{functor } F\ \} \\ & F^i out_F \circ F\,out_F^{i-1} \circ out_F \\ = & \quad \{\ \text{induction}\ \} \\ & F^i out_F \circ out_F^i \\ = & \quad \{\ \text{def. of } out_F^{i+1}\ \} \\ & out_F^{i+1} \end{array}$$

□

Theorem 16 says that, if every parameter of a recursive call to function h is only the sub-structure of the input and no other functions traverse over the same input of h, than h can be successfully transformed to a single mutual recursive definition which could be further improved by the Tupling Theorem, as stated in the following corollary.

Corollary 17 Under the same assumption in Theorem 16, we have

$$h \vartriangle g_1 \vartriangle \cdots \vartriangle g_{n-1} = (\!|\phi \circ \eta \vartriangle F\pi_1 \vartriangle \cdots \vartriangle F\pi_{n-1}|\!)_F$$

□

Before showing how the theorem works practically, let's define some useful natural transformations as follows.

$$\begin{array}{lcl} dist & :: & (X \times (Y + Z)) \to (X \times Y + X \times Z) \\ dist(x, (1, y)) & = & (1, (x, y)) \\ dist(x, (2, z)) & = & (2, (x, z)) \\ \\ zipSum & :: & (X_1 + Y_1) \times (X_2 + Y_2) \\ zipSum((1, x_1), (1, x_2)) & = & (1, (x_1, x_2)) \\ zipSum((2, y_1), (2, y_2)) & = & (2, (y_1, y_2)) \end{array}$$

Now recall the definition of $fib :: N \to N$ in Section 5.1. According to Proposition 15, we can transform it to the following hylomorphism.

$$fib = [\![\phi, id, out_{F_N} \vartriangle out_{F_N}^2]\!]_{G,G}$$

Here

$$\begin{array}{lcl} \phi & = & (Z \triangledown ((S(Z) \triangledown +) \circ dist)) \circ zipSum \\ G & = & F_N \times F_N^2 \end{array}$$

In addition, we remind readers that

$$\begin{array}{lcl} F_N & = & !\mathbf{1}\ +\ I \\ F_N^2 & = & !\mathbf{1}\ +\ (!\mathbf{1}\ +\ I) \\ out_{F_N} & = & \lambda x.\ \text{case } x \text{ of } Z \to (1, ());\ S(n) \to (2, n) \\ out_{F_N}^2 & = & \lambda x.\ \text{case } x \text{ of } Z \to (1, ());\ S(n) \to (2, \text{case } n \text{ of } Z \to (1, ());\ S(n') \to (2, n')) \end{array}$$

By Theorem 16 and Corollary 17, we soon have that

$$\mathit{fib} = \pi_1 \circ (\![\phi \circ (F_N\,\pi_1 \vartriangle F_N\,\pi_2) \vartriangle F_N\,\pi_1]\!)_{F_N}.$$

Simple simplification gives

$$\mathit{fib} = \pi_1 \circ (\![(Z \triangledown ((S(Z) \triangledown +) \circ \mathit{dist})) \vartriangle F_N\,\pi_1]\!)_{F_N}$$

which could be in-lined to the following linear program.

$$\begin{array}{lcl} \mathit{fib}\ n & = & x, \quad \text{where } (x, y) = f'\ n \\ f'\ Z & = & (Z, (1, ())) \\ f'\ (S(n)) & = & (\text{case i of } 1 \to S(Z); 2 \to x + y,\ (2, x)) \\ & & \text{where } (x, (\mathsf{i}, y)) = f'\ n \end{array}$$

5.2.3 Removing Unnecessary Traversals of Intermediate Results

It is commonplace in a recursive definition where some intermediate results are traversed by another recursive function. We can generally formulate it by the equation:

$$f \circ \mathit{in}_F = \phi \circ F(f \vartriangle (g \circ f)). \tag{5.1}$$

where the intermediate results of f is not only used for the final results (the first f in the RHS) but also traversed by another recursive function g (the second f in RHS).

In fact, the traversals of intermediate results by g can be calculated away under a certain condition leading to a more efficient recursion. This is shown in the following theorem.

Theorem 18 Given is the recursive function f defined by Equation (5.1). If there exists ψ satisfying

$$g \circ \phi = \psi \circ F((g \vartriangle \mathit{id}) \times \mathit{id}) \tag{5.2}$$

then

$$f = \pi_1 \circ (\![\phi \vartriangle (\psi \circ F((\pi_2 \vartriangle \pi_1) \vartriangle \pi_2))]\!)_F$$

Proof: It suffices to prove that

$$f \vartriangle (g \circ f) = (\![\phi \vartriangle (\psi \circ F((\pi_2 \vartriangle \pi_1) \vartriangle \pi_2))]\!)_F.$$

According to the Tupling Theorem, it is only need to prove

$$(g \circ f) \circ \mathit{in}_F = (\psi \circ F((\pi_2 \vartriangle \pi_1) \vartriangle \pi_2)) \circ F(f \vartriangle (g \circ f)).$$

To this end, we do calculation as follows.

$$\begin{array}{ll}
 & g \circ f \circ in_F = \psi \circ F((\pi_2 \vartriangle \pi_1) \vartriangle \pi_2) \circ F(f \vartriangle (g \circ f)) \\
\equiv & \quad \{ \text{ Equation (5.1) } \} \\
 & g \circ \phi \circ F(f \vartriangle (g \circ f)) = \psi \circ F((\pi_2 \vartriangle \pi_1) \vartriangle \pi_2) \circ F(f \vartriangle (g \circ f)) \\
\equiv & \quad \{ \text{ Equation (5.2) } \} \\
 & \psi \circ F((g \vartriangle id) \times id) \circ F(f \vartriangle (g \circ f)) = \psi \circ F((\pi_2 \vartriangle \pi_1) \vartriangle \pi_2) \circ F(f \vartriangle (g \circ f)) \\
\equiv & \quad \{ \text{ Functor } F \text{, simplification } \} \\
 & \psi \circ F((g \circ f \vartriangle f) \vartriangle (g \circ f)) = \psi \circ F((g \circ f \vartriangle f) \vartriangle (g \circ f)) \\
\equiv & \quad \{ \text{ obvious } \} \\
 & \text{True}
\end{array}$$

□

The idea behind Theorem 18 is simple. In order to avoid unnecessary traversals of intermediate results by g, it is sufficient to find a condition for $g \circ f$ being a single recursion. This condition is expressed by Equation (5.2) reading that g should not recursively applied to the results produced by itself.

With the above theorem, we can improve *foo* given in Section 5.1. First, we rewrite the recursive definition into the form of Equation (5.1).

$$\begin{array}{lll}
foo \circ in_{F_{L_A}} & = & \phi \circ F_{L_A}(foo \vartriangle (sum \circ foo)) \\
 & & \text{where } \phi = Nil \triangledown (\lambda(x,(p_i,p_s)).\, Cons(x+p_s,p_i))
\end{array}$$

Next, we check whether *sum* can be expressed in the form of Equation 5.2.

$$\begin{array}{ll}
 & sum \circ \phi \\
= & \quad \{ \ \phi \ \} \\
 & sum \circ (Nil \triangledown (\lambda(x,(p_s,p_i)).\, Cons(x+p_s,p_i))) \\
= & \quad \{ \ \circ \text{ and } \triangledown \ \} \\
 & sum \circ Nil \triangledown sum \circ (\lambda(x,(p_s,p_i)).\, Cons(x+p_s,p_i)) \\
= & \quad \{ \ sum \ \} \\
 & 0 \triangledown (\lambda(x,(p_s,p_i)).\, (x+p_s+sum(p_i))) \\
= & \quad \{ \text{ Define } \psi = \lambda(x,((p_i,p),p_s)).\, (x+p_s+p_i) \ \} \\
 & (0 \triangledown \psi) \circ F_{L_A}((sum \vartriangle id) \times id)
\end{array}$$

Finally, according to Theorem 18, it soon follows that

$$foo = \pi_1 \circ (\!|\phi \vartriangle (\psi \circ F_{L_A}((\pi_2 \vartriangle \pi_1) \vartriangle \pi_2))|\!)_{F_{L_A}}$$

which can be easily in-lined to the efficient program as given in Section 5.1.

5.3 Cheap Tupling Strategy

So far, we have already given the three simple calculational rules, and have demonstrated how they are used to improve recursions. in summary, the general strategy in the application of each calculational rule is a two-step calculation, namely,

- Rewriting the given recursion into a suitable form required by the theorem;
- Performing simple calculation according to the rule in the theorem.

Since these two steps can be *automatically* implemented at a rather low cost compared with the previous tupling methods [Chi93], it becomes direct and possible to embed the rule in a real compiler of functional languages.

Now let's turn to see how to give full play of these three rules to handle more complicated recursions. Rather than discussing in a formal way, we show our idea by the following example. Suppose that we want to improve function f defined by[2]

$$\begin{array}{lcl} f\,(x_1 : x_2 : xs) & = & x_1 + f\,(x_2 : xs) + f\,(xs) + g\,(x2 : xs) \\ g\,(x_1 : x_2 : x_3 : xs) & = & x_1 + g\,(x_2 : x_3 : xs) + g\,(x_3 : xs) + g\,(xs) + f\,(x_2 : x_3 : xs) \end{array}$$

where f calls another recursive function g. Here we omit the equations for the base cases for both f and g, e.g., $f\,[\,] = \cdots$, $f\,[x] = \cdots$, $f\,[x_1, x_2] = \cdots$, etc. Obviously, there are has many redundant recursive calls to f and multiple traversals of the input (by f and g), and hence this definition is inefficient.

First, we eliminate multiple traversals in the recursive definition by grouping the functions *traversing over the same data structure*. We usually start with this calculation as it could turn mutual recursive definitions into a single non-mutual one, making room for the later removal of redundant recursive calls by Corollary 17 and unnecessary traversals by Theorem 18. To this end, we define $h = f \vartriangle g$. By Corollary 13, we can have

$$\begin{array}{lcl} f & = & \pi_1 \circ h \\ h\,(x_1 : x_2 : x_3 : xs) & = & x_1 + x_1 + d_1\,(h\,(x_2 : x_3 : xs)) + d_2\,(h\,(x_3 : xs)) + d_3\,(h\,xs) \end{array}$$

where d_1, d_2 and d_3 are functions defined by $d_1(x, y) = x + x + y + y$; $d_2(x, y) = x + y$; $d_3(x, y) = y$. Next, we turn to improve h in order to improve f. We can eliminate redundant recursive calls to h by Corollary 17, and obtain the result something like

$$h = \pi_1 \circ (\![\phi]\!)_{F_{L_A}}$$

where ϕ is another function. Finally, we use Theorem 18 to eliminate possible unnecessary traversals in the catamorphism $(\![\phi]\!)_{F_{L_A}}$.

In summary, our cheap tupling is proceeded in the following way. Given a set of functions defined by recursions. For each function, we calculate its recursive definition to an efficient one in the order of eliminating multiple traversals, removing redundant recursive calls and getting ride of unnecessary data traversals.

It is worth noting that our cheap tupling can be well coexisted with fusion under the transformation in calculational form. As a matter of fact, they assist each other to obtain better optimization:

[2]For notational convenience, we use infix operator : for list Cons operator, i.e., $x : xs$ denotes $Cons(x : xs)$, and use $[\,]$ to denote *Nil*.

- Fusion merges several recursive functions into one, which makes our cheap tupling algorithm easy to find how to tuple them.

- Tupling improves the recursion by constructing an efficient catamorphism (a special instance of hylomorphism) for it. It does not lose any possibility for fusion in case it is composed with another recursion (see Section 2.2)

We shall not give the detailed discussion to convince the reader. A relevant study can be found in [HIT96c] where tupling and fusion are used together to derive list homomorphisms (i.e., catamorphisms over append lists). Recently, Chin [Chi95] discussed this issues on the fold/unfold transformation basis. But it is quite complicated to choose carefully the order of tupling and fusion since they may hinder each other.

5.4 Related Work and Discussions

It has been argued that the use of generic control structures which capture patterns of recursions in a uniform way is of great significance in program transformation and optimization[MFP91, Fok92b, SF93, TM95]. Our work is much related to these studies. In particular, our work was greatly inspired by the success of applying this approach to fusion transformation as studied by Sheard and Fegaras'[SF93], Gill Launchbury and Peyton Jones[GLJ93], and Takano and Meijer's[TM95].

We made the first attempt to apply this calculational approach to the tupling transformation as well. Previous work, as intensively studied by Chin [Chi93], tries to tuple *arbitrary* functions by *fold-unfold* transformations. In spite of its generality, it has to keep track of all function calls and devise clever control to avoid infinite unfolding, resulting in high runtime cost which prevents it from being employed in a real compiler system.

We follows the experience of work of fusion in calculational form [TM95] where a simple calculational rule is used. We identify three patterns of inefficient recursions, and construct three calculational rules to improve them based on the structural knowledge in these patterns. Though being simple and less general than Chin's, our cheap tupling transformation, as demonstrated, can be applied to a wide class of functions. Moreover, it can be efficiently implemented in a practical compiler and can be naturally coexisted with fusion under the basis of transformation in calculational form. Chin [Chi95] discussed the intergration of fusion and tupling under the basis of fold/unfold transformation, whose algorithm is much more complicated than ours because of the complicated control of infinite unfoldings for when fusion and tupling are used in one system.

The idea in Theorem 12 for the tupling of mutual recursive definitions is not new at all. It basically the same as Takeichi's *generalization algorithm* [Tak87] and Fokkinga's *mutumorphisms* [Fok92b]. Takeichi showed how to define a higher order function common to all functions mutually defined so that multiple traversals of the same data structures in the mutual recursive definition can be eliminated. Fokkinga proposed the idea of mutumorphism and develop the laws for the fusion of mutual recursions with other functions. We generalize their idea (Corollary 13), and investigate deeply how it can be used further for the elimination of redundant recursive calls and unnecessary traversals of data (Corollary 17 and Theorem 18).

Our idea (Theorem 16) for avoiding redundant recursive calls is much simpler than the existed techniques such as, memoization [Mic68, Hug85], tabulation [Bir80, Coh83] and tupling [Chi92]. Though being less general, it can be done by a simple calculation rather than by complicated program analysis or applied at run time. In fact, we are much influenced by Hughes' idea of *lazy memoization* [Hug85] in which it is only required to reuse the previously computed results of recursive calls applied to arguments *identical* (not equal in value) to previous ones – that is, arguments stored in the same place in memory. We impose more restriction so that and we can remove redundant recursive calls by calculation at compilation time rather than at execution time.

5.5 Conclusions

In this chapter, we put forward our idea of the cheap tupling transformation, obtaining efficient recursions without multiple traversals of the same data structures, redundant recursive calls, and unnecessary traversals of intermediate data structures. Our cheap tupling is "cheap" in the sense that it can be implemented very *cheaply* and hence more practical than previous studies. But this comes at price of less generality. Although we have demonstrated that combining of three rules can deal with more complicated cases (Section 5.3), there are still many recursions that our tupling cannot be applied. How to enlarge our application scope is one of our future work.

Chapter 6

Calculating Accumulations

The *accumulation strategy* consists of generalizing a function over an algebraic data structure by inclusion of an extra parameter, an *accumulating parameter*, for reusing and propagating intermediate results. It is one of the standard optimization techniques taught to functional programmers[Hen80].

We are faced with two difficulties in the accumulation strategy. One is to determine *where* and *when* to generalize the original function. By "where," we mean what part of the function should be generalized. We may have many alternatives such as generalizing a constant to a variable or generalizing an expression to a function[PP93]. By "when," we mean how many steps of *unfolding* are needed to find a suitable place for generalization, since a proper place usually comes out after several unfoldings. One general way, known as *forcing generalization*, is to do generalization in case *folding* cannot be done during unfold/fold transformations[Bir84, Fea87, PP93, PS87], although related studies remain in an ad-hoc level.

The other difficulty, surprisingly not yet receiving its worthy consideration, is how to manipulate accumulations. We believe this more important for the following reason. As we know, one advantage of functional programming is that it allows greatly improved modularity. Functions can be defined in terms of smaller and simpler functions which are "glued" together to give their result. So, for a rather complicated function whose accumulation is difficult to obtain, we may decompose it into several simpler ones whose accumulations are easier to find, and then re-compose the result.

In this chapter, we are trying to overcome these difficulties and to give a systematic study on the accumulation strategy. We shall formulate accumulations as *higher order catamorphisms*, and propose several general transformation rules for calculating accumulations (i.e., finding and manipulating accumulations) by *calculation-based* (rather than a search-based) program transformation methods. The main contributions of our work are as follows.

- We formulate accumulations as higher order catamorphisms facilitating program calculation.

- We have provided several general rules for calculating accumulations, i.e., finding and manipulating accumulations. Each application of the rule can be seen as a canned application of unfold/simplify/fold in the traditional transformational programming[BD77]. Our calculational approach can avoid the process of keeping track of function calls and the clever control to avoid infinite unfolding which always introduces substantial cost and complexity in search-based transformation.

- Instead of ad-hoc study on accumulation strategy, our study is systematic and some of them can be embedded in an automatic transformation system. On the other hand, our theory is built upon category theory which is more general. For example, our definition of accumulations is applicable to any type besides lists.

- Our study of generic theorems for higher order catamorphisms which return functions (rather than values), we believe, is also a particular valuable avenue to pursue since programs that deal with *state* are quite common and can be handled algebraically.

This Chapter is organized as follows. In Section 6.1, we formulate accumulations as higher order catamorphisms and demonstrate through many examples that higher order catamorphisms can describe accumulations effectively. In Section 6.2, we propose our *Generalization Theorem* for deriving accumulations. Section 6.3 proposes two general *Accumulation Promotion (Fusion) Theorems* for manipulating accumulations. An example of the derivation of an efficient algorithm for longest path problem are given in Section 6.4, and some related works and conclusions are described in Section 6.5.

6.1 Accumulations and Catamorphisms

An accumulation[Bir84] is a kind of computation which proceeds over an algebraic data structure while keeping some information in an accumulating parameter to be used as an intermediate result. We shall borrow the word "accumulation" to refer to the function which performs accumulating computation.

As a simple example, consider the following definition of *isum* which computes the initial prefix sums of a list, i.e., $isum\ [x_1, x_2, \cdots, x_n]\ 0\ =\ [0, x_1, x_1 + x_2, \cdots, x_1 + x_2 + \cdots + x_n]$.

$$\begin{array}{lcl} isum\ [\,]\ d & = & [d] \\ isum\ (x : xs)\ d & = & d : isum\ xs\ (d + x) \end{array}$$

In this definition, the second parameter of *isum* is the accumulating one that keeps partial sums for the later reuse, leading to an efficient linear algorithm.

Now by abstracting d in both equations, we obtain

$$\begin{array}{lcl} isum\ [\,] & = & \lambda d.[d] \\ isum\ (x : xs) & = & \lambda d.(d : isum\ xs\ (d + x)), \end{array}$$

which defines a catamorphism $(\![e \triangledown \otimes]\!)$ over cons lists where

$$\begin{array}{lcl} e & = & \lambda d.[d] \\ x \otimes p & = & \lambda d.(d : p\ (d + x)). \end{array}$$

This is a *higher order catamorphism* in the sense that it takes a list to yield a function. Generally, we can formulate accumulations with the use of higher order catamorphisms as in the following proposition.

Proposition 19 (Accumulation) Let the data type T be the initial F-algebra, i.e., $(T, in_F) = \mu F$. An accumulation, which inducts over T using an accumulating parameter of type A and yields a value of type B, can be represented by a higher order catamorphism $(\![\phi]\!) :: T \to A \to B$ where $\phi :: F(A \to B) \to (A \to B)$. □

Some points on this proposition are worth noting. Firstly, in spite of the restrictions of higher order catamorphisms, such accumulations have no loss in descriptive power. Recalling our example of *isum*, we can naturally rewrite it to be a higher order catamorphism by means of abstraction.

Secondly, there is no restriction on the type of the accumulating parameter, which may be either a function or a basic value. This helps to describe an accumulation concisely if a proper accumulation parameter is chosen. Consider the function *idif* that computes the initial differences of a list, e.g. $idif\ [5, 2, 1, 4] = [5, 5-2, 5-2-1, 5-2-1-4] = [5, 3, 2, -2]$. It can be defined efficiently as:

$$\begin{array}{lcl} idif\ xs & = & idif'\ xs\ id \\ idif' & = & (\![e \triangledown \otimes]\!) \\ \text{where} & & e = \lambda f.[\,] \\ & & x \otimes p = \lambda f.(f\ x : p\ ((f\ x)-)), \end{array}$$

Here, id denotes the identity function. In this definition, a function is used as the accumulating parameter. If we insisted on using non-function values, the algorithm would have become quite complicated because the subtraction is not associative.

Thirdly, our accumulations is valid for any freely constructed types such as lists, trees and so on. In fact, one of our motivations for associating higher order catamorphisms with accumulations was to find a method to make downwards tree accumulations manipulable and efficient. Gibbons[Gib92] proposed the problem and tried to solve it with the aid of catamorphisms as we do here. Catamorphisms that he referred to are first order ones, and after a complicated discussion certain conditions are turned to be necessary for downwards tree accumulation to be expressed as a catamorphism. As will be seen in Example 6, we can give a concise definition using higher order catamorphism and make Gibbons' conditions unnecessary (see [HIT94a] for detail).

Example 6 (Downwards tree accumulation) We shall demonstrate how to describe an accumulation over trees. We have shown that the type of binary trees with elements of type a can be defined as the initial F_T-algebra in Section 2.2.

Downwards tree accumulation passes information downwards, from the root towards the leaves; each element is replaced by some functions on its ancestors. We denote a downwards accumulation by $(f, \oplus, \otimes)^{\Downarrow} :: \mathit{Tree}\ a \rightarrow \mathit{Tree}\ b$, which depends on three operations $f :: a \rightarrow b$, $(\oplus) :: b \rightarrow a \rightarrow b$ and $(\otimes) :: b \rightarrow a \rightarrow b$. This function is defined by

$$\begin{array}{l} (f,\oplus,\otimes)^{\Downarrow}\ (\mathit{Leaf}\ a) = \mathit{Leaf}\ (f\ a) \\ (f,\oplus,\otimes)^{\Downarrow}\ (\mathit{Node}\ (a,x,y)) \\ \quad = \mathit{Node}\ (f\,a, (((f\,a)\oplus),\oplus,\otimes)^{\Downarrow}\,x, (((f\,a)\otimes),\oplus,\otimes)^{\Downarrow}\,y). \end{array}$$

For instance, $(id, +, +)^{\Downarrow}$ is a function which replaces each node with the sum of all its ancestors.

Obviously, the function $(f, \oplus, \otimes)^{\Downarrow}$ cannot be specified by a first order catamorphism if these three operations do not satisfy certain conditions. And looking for such conditions may lead to a very complicated discussion as Gibbons did[Gib92]. If we use a higher order catamorphism, we can describe it as follows.

$$\begin{array}{l} (f,\oplus,\otimes)^{\Downarrow}\ t = (\![l \triangledown n]\!)_{F_T}\ t\ f \\ \quad \text{where} \\ \qquad l\ a\ \rho = \mathit{Leaf}(\rho\ a) \\ \qquad n\ (b,u,v)\ \rho = \mathit{Node}\ (\rho\ b, u((\rho\ b)\oplus), v((\rho\ b)\otimes)) \end{array}$$

We have demonstrated its use in calculating efficient parallel tree algorithms in [HIT94b]. □

6.2 Derivation of Accumulations

In this section, we shall propose our Generalization Theorem for deriving an accumulation from a structured specification in catamorphisms.

Recall that our accumulations are sort of higher order catamorphisms. If the specification is an *first order catamorphism*, namely a catamorphism whose result is a value instead of a function, the derivation of an accumulation can be considered as a transformation from a first order catamorphism to a higher order one. The following lemma is useful in such transformation. Note that for notational convenience we abbreviate the equation

$$f \circ h = g \circ h$$

to

$$f = g \ \mathsf{mod} \ h.$$

Lemma 20 Let $(\![\phi]\!)_F$ be a first order catamorphism. If there exists a binary operator $\oplus$ with a right identity e such that

$$(\oplus) \circ \phi = \psi \circ F(\oplus) \ \mathsf{mod} \ F\,(\![\phi]\!)_F,$$

then

$$(\![\phi]\!)_F \ xs = (\![\psi]\!)_F \ xs \ e.$$

Proof:

$$\begin{array}{ll} & (\![\phi]\!)_F \ xs \\ = & \quad \{ \ e \text{ is a right identity of } \oplus \ \} \\ & (\oplus) \ ((\![\phi]\!)_F \ xs) \ e \\ = & \quad \{ \ \text{Function application and composition} \ \} \\ & ((\oplus) \circ (\![\phi]\!)_F) \ xs \ e \\ = & \quad \{ \ \text{Fusion Theorem} \ \} \\ & (\![\psi]\!)_F \ xs \ e \end{array}$$

□

Lemma 20 tells us that the transformation from a first order catamorphism to a higher order one can be considered to find a suitable binary operator for relating the original catamorphism with the newly introduced accumulating part. However, from Lemma 20, we know only what property it should obey, not how it should be derived. To see how to derive such a binary operator $\oplus$ and thus obtain ψ, let us compare the both sides of the equation of

$$(\oplus) \circ \phi = \psi \circ F(\oplus) \ \mathsf{mod} \ F\,(\![\phi]\!)_F,$$

and we can see that ϕ is expected to have the form of $g \circ F(\oplus)$ so that both sides can end with $F(\oplus)$. This suggests us to derive $\oplus$ from ϕ, and hence calculate ψ, which leads to our Generalization Theorem.

Theorem 21 (Generalization) Let $(\![\phi]\!)_F$ be the given first order catamorphism. If $\phi = g \circ F(\oplus)$ where $\oplus$ is a binary operator with right identity e, then

$$(\![\phi]\!)_F \ xs = (\![\psi]\!)_F \ xs \ e,$$

where $\psi = (\oplus) \circ g \ \mathsf{mod} \ F((\oplus) \circ (\![\phi]\!)_F)$.

Proof: According to Lemma 20, it is enough to prove that

$$(\oplus) \circ \phi = \psi \circ F\,(\oplus) \ \mathsf{mod} \ F(\![\phi]\!)_F.$$

This can be easily verified with the composite property of functor F, i.e., $F\,f \circ F\,g = F\,(f \circ g)$. □

In fact, the Generalization Theorem provides us a *generalization procedure* for deriving accumulations. In practical program calculation the given catamorphism is likely to have the form of $(\!|\phi_1 \triangledown \cdots \triangledown \phi_n|\!)_{F_1+\cdots+F_n}$.

1. Rewrite each ϕ_i in a given catamorphism $(\!|\phi_1 \triangledown \cdots \triangledown \phi_n|\!)_{F_1+\cdots+F_n}$ to a form of $g_i \circ F_i(\oplus)$ such that $\oplus$ is a binary operator with a right identity e.
2. Calculate ψ_i according to the equation
$$\psi_i = (\oplus) \circ g_i \ \mathsf{mod} \ F_i((\oplus) \circ (\!|\phi_1 \triangledown \cdots \triangledown \phi_n|\!)_{F_1+\cdots+F_n}).$$
3. Group ψ_i's to be $(\!|\psi_1 \triangledown \cdots \triangledown \psi_n|\!)_{F_1+\cdots+F_n}$. Obviously,
$$(\!|\phi_1 \triangledown \cdots \triangledown \phi_n|\!)_{F_1+\cdots+F_n} \ xs = (\!|\psi_1 \triangledown \cdots \triangledown \psi_n|\!)_{F_1+\cdots+F_n} \ xs \ e.$$

For convenience, we define that a parameter x_j in $\phi_i(x_1, ..., x_m) : T$ is said to be a *recursive parameter* if x_j has the type of T. In what follows, we would like to use p_j's instead of x_j's to explicitly denote recursive parameters. One property with recursive parameter is as follows.

Corollary 22 (Recursive parameter) Any recursive parameter p of function ψ_i obtained by the generalization procedure can be represented by $(p'\oplus)$ using another function p'.

Proof: A direct result from the equation:
$$\psi_i = (\oplus) \circ g_i \ \mathsf{mod} \ F_i((\oplus) \circ (\!|\phi_1 \triangledown \cdots \triangledown \phi_n|\!)_{F_1+\cdots+F_n}).$$
□

One use of this corollary can be seen in Example 7 when we derive ψ_2.

Example 7 (*isum*) By way of illustration, consider the function *isum* again. Suppose we are given the following first order catamorphism:

$$\begin{aligned} &isum = (\!|\phi_1 \triangledown \phi_2|\!)_{F_1+F_2} \\ &\quad \text{where} \quad \phi_1() = [0] \\ &\qquad\qquad\ \ \phi_2(x, p) = 0 : (x+) * p \end{aligned}$$

where $F_1 = !\mathbf{1}$ and $F_2 = !a \times I$. It is an inefficient quadratic algorithm, so we are trying to derive an efficient accumulation for it.

We begin by rewriting ϕ_1 and ϕ_2 to find g_1, g_2 and $\oplus$. It is trivial to see that
$$g_1 = \phi_1$$

from $\phi_1 = g_1 \circ F_1(\oplus)$ since $F_1\ f = id$. Now we hope to find g_2 and $\oplus$ satisfying the equation $\phi_2 = g_2 \circ F_2(\oplus)$ where $F_2\ f = id \times f$. This is the same to find g_2 and $\oplus$ satisfying $\phi_2(x, p) = g_2(x, (p\ \oplus))$. Since in the definition of ϕ_2 the operation on p is $(x+)*$, we may define $\oplus$ as

$$p \oplus y = (y+) * p,$$

and we have

$$g_2(x, p') = 0 : p'\ x.$$

Note that $\oplus$ has the right identity 0 since $p \oplus 0 = (0+) * p = p$.

Next, we turn to calculate ψ_1 and ψ_2.

$$\begin{array}{cl}
 & \psi_1\ ()\ y \\
= & \quad \{\ \text{Generalization Theorem}\ \} \\
 & ((\oplus) \circ g_1)\ ()\ y \\
= & \quad \{\ \text{Function composition}\ \} \\
 & g_1() \oplus y \\
= & \quad \{\ \text{Def. of } g_1 \text{ and } \oplus\ \} \\
 & (y+) * [0] \\
= & \quad \{\ \text{Map}\ \} \\
 & [y]
\end{array}$$

$$\begin{array}{cl}
 & \psi_2\ (x, (p\oplus))\ y \\
= & \quad \{\ \text{Generalization Theorem}\ \} \\
 & ((\oplus) \circ g_2)\ (x, (p\oplus))\ y \\
= & \quad \{\ \text{Function composition}\ \} \\
 & g_2(x, (p\oplus)) \oplus y \\
= & \quad \{\ \text{Def. of } g_2 \text{ and } \oplus\ \} \\
 & (y+) * (0 : (p \oplus x)) \\
= & \quad \{\ \text{Map}\ \} \\
 & y : ((y+) * (p \oplus x)) \\
= & \quad \{\ \text{Def. of } \oplus\ \} \\
 & y : ((y+) * ((x+) * p)) \\
= & \quad \{\ \text{Map and associativity of ``+''}\ \} \\
 & y : (((y + x)+) * p) \\
= & \quad \{\ \text{Def. of } \oplus\ \} \\
 & y : (p \oplus (y + x)) \\
= & \quad \{\ \text{Sectioning}\ \} \\
 & y : (p\oplus)(y + x)
\end{array}$$

According to the second step of our procedure, it follows that

$$\begin{array}{lcl}
\psi_1\ () & = & \lambda y.[y] \\
\psi_2\ (x, p') & = & \lambda y.(y : p'\ (y + x)).
\end{array}$$

Finally, according to the Generalization Theorem, we get

$$isum\ xs = (\![\psi_1 \triangledown \psi_2]\!)\ xs\ 0$$

which is the same as we introduced at the beginning of Section 6.1. □

The generalization procedure requires us to derive a suitable binary operator $\oplus$. But sometimes, we cannot find a right identity for such $\oplus$. Techniquely, in this case, we may create a virtual one as in the following example.

Example 8 (*subs*) Consider the function *subs* which accepts a cons list and yields the set of all its subsequences:

$$\begin{array}{lcl} subs\ [\,] & = & \{[\,]\} \\ subs\ (x : xs) & = & subs\ xs \cup (x :) * subs\ xs, \end{array}$$

i.e.,

$$\begin{array}{l} subs = (\![\phi_1 \triangledown \phi_2]\!) \\ \quad \text{where} \quad \phi_1() = \{[\,]\} \\ \qquad\qquad\quad \phi_2(x, p) = p \cup (x :) * p. \end{array}$$

With a similar derivation as for *isum*, we can define the binary operator for *subs* as

$$p \oplus y = (y :) * p.$$

The problem with such $\oplus$ is that it has no right identity. Nevertheless, we could assume a virtual right identity ϵ and continue the calculation. By the generalization procedure we can have

$$\begin{array}{l} subs\ xs = (\![\psi_1, \psi_2]\!)\ xs\ \epsilon \\ \quad \text{where} \quad \psi_1\ ()\ y = \{[y]\} \\ \qquad\qquad\quad \psi_2\ (x, p)\ y = p\ y \cup (y :) * (p\ x). \end{array}$$

Section 6.4 will show how this definition is used in practical program derivation. □

The following is an often-quoted example illustrating the role of accumulation. We shall give the derivation of such accumulation using our approach.

Example 9 (*rev*) Consider the *rev* function which reverses a list. The initial quadratic specification is

$$\begin{array}{l} rev = (\![\phi_1 \triangledown \phi_2]\!) \\ \quad \text{where} \quad \phi_1() = [\,] \\ \qquad\qquad\quad \phi_2\ (a, p) = p +\!\!+ [a] \end{array}$$

According to the generalization procedure, we see that the binary operator $\oplus$ can be instantiated to $+\!\!+$ whose right identity is $[\,]$. We omit other calculation but give the final result:

$$\begin{aligned} & rev\ xs = (\![\phi'_1 \triangledown \phi'_2]\!)\ xs\ [\,] \\ & \quad \text{where} \quad \phi'_1\ () = id \\ & \qquad\qquad\ \ \phi'_2\ (a, p) = p \circ (a\ :) \end{aligned}$$

which is the well-known efficient accumulation as that in [Hug86]. □

Sheard[SF93] also studied the generalization of structure programs, but he requires the associativity of $\oplus$ and puts many restrictions on the structure of $\phi'_i s$. On the contrary, we remove as many restrictions as possible in our theorem and give a much more general but practical generalization procedure. Our method covers Sheard's but not vice versa. For instance, our examples of the *isum* and the *subs* can not be dealt with by Sheard's.

6.3 Manipulating Accumulations

In this section, we propose several general rules for manipulating accumulations. By "manipulate accumulations" we mean the derivation of new accumulations from the old ones.

In compositional style of programming, it is usually the case where a function is composed with an accumulation. To find an accumulation for this composition, we propose the following theorem.

Theorem 23 (Accumulation Promotion (Fusion) 1) Let $(\![\phi]\!)_F$ and $(\![\psi]\!)_F$ be two accumulations. If

$$(h\circ) \circ \phi = \psi \circ F\,(h\circ)\ \ \mathsf{mod}\ \ F\,(\![\phi]\!)_F$$

then

$$h \circ ((\![\phi]\!)_F\ xs) = (\![\psi]\!)_F\ xs.$$

Proof: This can be easily proved by specializing the Fusion Theorem in which h is replaced by $(h\circ)$. □

Example 10 Suppose that we want to derive an accumulation algorithm for the composition of $length \circ rev$, where $length$ is a function computing the length of a list. Recall that we have got the accumulation algorithm $rev\ xs = (\![\phi'_1 \triangledown \phi'_2]\!)\ xs\ [\,]$ in Example 9, we can thus use Theorem 23 and calculate as follows.

$$\begin{array}{ll}
 & (length\circ) \circ \phi'_1 \\
= & \quad \{ \text{ Def. of } \phi'_1 \} \\
 & (length\circ) \circ \lambda().id \\
= & \quad \{ \text{ Calculation } \} \\
 & \lambda().length \\
= & \quad \{ \ F_1\ f = id, \text{ define } \psi_1 = \lambda().length \ \} \\
 & \psi_1 \circ F_1 (length\circ)
\end{array}$$

$$\begin{array}{ll}
 & (length\circ) \circ \phi'_2 \\
= & \quad \{ \text{ Def. of } \phi'_2 \} \\
 & (length\circ) \circ (\lambda(a,p).p \circ (a :)) \\
= & \quad \{ \text{ Calculation } \} \\
 & \lambda(a,p).length \circ p \circ (a :) \\
= & \quad \{ \ F_2 f = id \times f, \text{ define } \psi_2 = \lambda(a,p).p \circ (a :) \ \} \\
 & \psi_2 \circ F_2 (length\circ)
\end{array}$$

Therefore, by Theorem 23, we have

$$(length \circ rev)\ xs = (\![\psi_1 \triangledown \psi_2]\!)\ xs\ [\,].$$

□

Although we have successfully promoted *length* into *rev*, our derived accumulation is quite unsatisfactory. The reason is that Theorem 23 does not tell anything about manipulation on accumulating parameter, which is also very important. Our following theorem is for this purpose.

Theorem 24 (Accumulation Promotion (Fusion) 2) Let $(\![\phi]\!)_F$ and $(\![\psi]\!)_F$ be two accumulations. If

$$(\circ g) \circ \phi = \psi \circ F\,(\circ g) \ \ \mathsf{mod}\ \ F\,(\![\phi]\!)_F$$

then

$$((\![\phi]\!)_F\ xs) \circ g = (\![\psi]\!)_F\ xs$$

Proof: The proof is similar to that for Theorem 23 by specializing h to $(\circ g)$ in the Promotion (Fusion) Theorem. □

Example 11 Consider the point of the calculation in Example 10 where we have reached

$$(length \circ rev)\ xs = (\![\psi_1 \triangledown \psi_2]\!)\ xs\ [\,].$$

We are going to derive η_1, η_2 and g based on Theorem 24 such that

$$((\![\eta_1 \triangledown \eta_2]\!)\ xs) \circ g = (\![\psi_1 \triangledown \psi_2]\!)\ xs.$$

Observing that

$$\begin{array}{rcl}
(\circ g) \circ \eta_1 & = & \quad \{ \text{ Theorem 24 } \} \\
& & \psi_1 \circ F_1(\circ g) \\
& = & \quad \{ \ F_1 f = id, \text{ Def. of } \phi_1 \ \} \\
& & \lambda().length \\
& = & \lambda().(id \circ length) \\
& = & (\circ length) \circ (\lambda().id),
\end{array}$$

we may let $g = length$ and $\eta_1 = \lambda().id$. To derive η_2, we calculate as follows.

$$\begin{array}{rcl}
(\circ g) \circ \eta_2 & = & \quad \{ \text{ Theorem 24 } \} \\
& & \psi_2 \circ F_2(\circ length) \\
& = & \quad \{ \text{ Def. of } \psi_2 \text{ and } F_2 \ \} \\
& & (\lambda(a,p).p \circ (a\,:)) \circ (id \times (\circ length)) \\
& = & \lambda(a,p).p \circ length \circ (a\,:) \\
& = & \quad \{ \ length \circ (x\,:) = (1+) \circ length \ \} \\
& & \lambda(a,p).p \circ (1+) \circ length \\
& = & (\circ length) \circ (\lambda(a,p).p \circ (1+)) \\
& = & \quad \{ \text{ Def. of } g \ \} \\
& & (\circ g) \circ (\lambda(a,p).p \circ (1+))
\end{array}$$

It is immediate that $\eta_2 = \lambda(a,p).p \circ (1+)$. Therefore,

$$\begin{array}{rcl}
(length \circ rev)\ xs & = & (((\llbracket \eta_1 \triangledown \eta_2 \rrbracket\ xs) \circ length)\ [\,] \\
& = & \llbracket \eta_1 \triangledown \eta_2 \rrbracket\ xs\ (length\ [\,]) \\
& = & \llbracket \eta_1 \triangledown \eta_2 \rrbracket\ xs\ 0
\end{array}$$

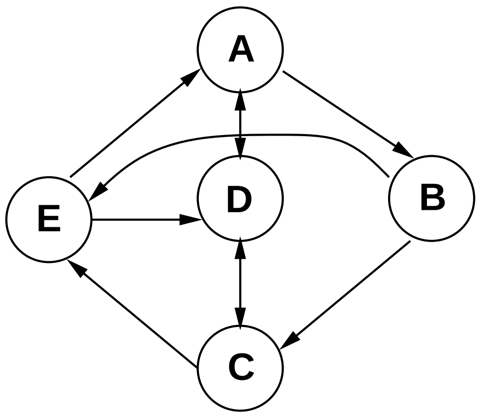
By in-lining the definition of catamorphisms, we get our familiar recursive definition:

$$\begin{array}{l}
(length \circ rev)\ xs = h\ xs\ 0 \\
\quad \text{where } \begin{array}{rcl} h\ [\,]\ y & = & y \\ h\ (x : xs)\ y & = & h\ xs\ (1 + y). \end{array}
\end{array}$$

$\square$

6.4 An Application

In this section, we explain how to apply our rules to a rather complicated example: calculating an efficient program for the *longest subsequences paths* problem. Bird[Bir84] proposed an impressive study on this example. We review it in order to show that some of the Bird's explanation of where to generalize and how to proceed program transformation can be made more systematic by program calculation in a theorem-driven manner. In other words, we proceed the derivation by repeatedly trying to calculate program to a form that meets the conditions of our theorems so that our theorems become applicable.



arc A B
arc A D
arc B C
arc B E
arc C D
arc C E
arc D A
arc D C
arc E A
arc E D

Figure 6.1: An Example of a Graph and its Representation

Beginning with a simple and straightforward specification of the problem in hand, our calculational method coerces the specification into an executable and acceptably efficient program in some given functional language such as Gofer.

6.4.1 Specification

Our problem is to determine the length of the longest subsequence of a given sequence of vertices that forms a connected path in a given directed graph G. For simplicity we suppose that G is presented through a predicate arc so that $arc\ a\ b$ is true just in the case that (a, b) is an arc of G from vertex a to vertex b. For example, Figure 6.1 gives a graph and its representation. If the input sequence is $[C, A, B, D, A, C, D, E, B, E]$. The length of the longest path sequence is 5, particular length for solutions of $[C, D, A, B, E]$ and $[A, B, C, B, E]$. Our definition of the problem reads:

$$\begin{aligned}
psp\ &= max \circ (length*) \circ (path\triangleleft) \circ subs \\
&\text{where}\quad path\ [\,] = True \\
&\qquad\qquad path\ [x] = True \\
&\qquad\qquad path\ (x_1 : x_2 : xs) = arc\ x_1\ x_2\ \wedge\ path\ (x_2 : xs)
\end{aligned}$$

Here $p\triangleleft$ is a function that takes a set and removes those elements not satisfying predicate p.

It is important to observe that the above does describe an algorithm to solve the problem, but it is not an efficient one. Clearly, the algorithm is exponential in the length of the given sequence.

6.4.2 Program Derivation

Our derivation of an accumulation for *psp* begins with finding an accumulation for *subs*, and then manipulates accumulations according to the Accumulation Promotion Theorems.

Deriving an accumulation for *subs*

This derivation has been already given in Example 8. The result is as follows.

$$\begin{array}{ll} subs\ xs = (\!|\psi_1 \triangledown \psi_2|\!)_{F_1+F_2}\ xs\ \epsilon & \\ \quad \text{where} \quad \psi_1\ ()\ y = \{[y]\} & \\ \qquad\qquad\ \ \psi_2\ (x,p)\ y = p\ y \cup (y\ :) * (p\ x) & \end{array}$$

where $F_1 = !\mathbf{1}$ and $F_2 = !a \times I$.

Manipulating $(path\triangleleft) \circ subs$

After obtaining the accumulation for *subs*, we step to derive an accumulation for the composition of the function *path*$\triangleleft$ with the *subs* based on Theorem 23.

Let $k = path\triangleleft$. We calculate ξ_1, ξ_2 from ψ_1, ψ_2 and k based on the condition in Theorem 23. Observing that

$$\begin{array}{rcl} (k\circ) \circ \psi_1 & = & (k\circ) \circ (\lambda().\lambda y.\{[y]\}) \\ & = & \lambda().\lambda y.(k\ \{[y]\}) \\ & = & \lambda().\lambda y.\{[y]\} \\ & = & (\lambda().\lambda y.\{[y]\}) \circ F_1(k\circ) \end{array}$$

we thus get ξ_1:

$$\xi_1\ ()\ y \ = \ \{[y]\}.$$

Now calculate ξ_2

$$\begin{array}{rcl} (k\circ) \circ \psi_2 & = & (k\circ) \circ (\lambda(x,p).\lambda y.(p\ y \cup (y\ :) * (p\ x))) \\ & = & \lambda(x,p).\lambda y.k\ (p\ y) \cup \underline{k((y\ :) * (p\ x))}). \end{array}$$

Before continuing the calculation, we break to look into ϵ (see Section 6.2), the right identity of $\oplus$. It should represent a vertex in the graph and satisfy $\epsilon : xs = xs$. Unfortunately, such vertex does not exist. To fix this problem, we consider ϵ as a virtual vertex with edges going to every vertex, i.e., $arc\ \epsilon\ v = True$ for each vertex v in graph G. To keep the graph's path soundness, we do not allow any edge going from any vertex of G to ϵ, i.e., $arc\ v\ \epsilon = False$. Now return to our calculation for the underlined part. In the following, "$p \to q; r$" is used to denote "if p then q else r".

$$
\begin{array}{ll}
 & k((y\,:) * (p\ x)) \\
= & \quad \{ \text{ By Corollary 22 let } p = (p'\oplus), \text{ def. of } k \ \} \\
 & path \lhd\ ((y\,:) * ((x\,:) * p')) \\
= & \quad \{ \text{ Map } \} \\
 & path \lhd\ (((y\,:) \circ (x\,:)) * p') \\
= & \quad \{ \text{ Def. of } path \text{ and } \lhd \ \} \\
 & arc\ y\ x \to (y\,:) * (path \lhd\ ((x\,:) * p')); path \lhd\ ((x\,:) * p') \\
= & \quad \{ \text{ Def. of } k\text{, and the above ``let'' nassumption } \} \\
 & arc\ y\ x \to (y\,:) * (k\ (p\ x)); k\ (p\ x) \\
= & \quad \{ \ \epsilon : xs = xs \ \} \\
 & arc\ y\ x \to (y \neq \epsilon \to (y\,:) * (k\ (p\ x)); k\ (p\ x)); k\ (p\ x) \\
= & \quad \{ \text{ Property of } \to \} \\
 & arc\ y\ x \wedge y \neq \epsilon \to (y\,:) * (k\ (p\ x)); k\ (p\ x)
\end{array}
$$

So

$$
\begin{array}{rcl}
(k\circ) \circ \psi_2 & = & \lambda(x,p).\lambda y.k\ (p\ y) \cup \\
 & & (arc\ y\ x \wedge y \neq \epsilon \to (y\,:) * (k\ (p\ x)); k\ (p\ x)) \\
 & = & (\lambda(x,p').\lambda y.(p'\ y) \cup \\
 & & (arc\ y\ x \wedge y \neq \epsilon \to (y\,:) * (p'\ x); (p'\ x)) \circ\ F_2(k\circ).
\end{array}
$$

Thus we get ξ_2 defined as

$$
\begin{array}{l}
\xi_2\ (x,p)\ y = (p\ y) \cup h \\
\quad \text{where} \quad h = arc\ y\ x \wedge y \neq \epsilon \to (y\,:) * (p\ x); (p\ x).
\end{array}
$$

Finally, according to Theorem 23, we obtain

$$((path\lhd) \circ subs)\ xs = (\![\xi_1 \triangledown \xi_2]\!)\ xs\ \epsilon.$$

Promoting $max \circ (length*)$ into the obtained accumulation

We continue promoting $max \circ length*$ into the derived accumulation according to the Accumulation Promotion Theorems, as we did above. We omit the detail calculation but give the last result.

$$
\begin{array}{l}
psp\ xs = (\![\eta_1 \triangledown \eta_2]\!)\ xs\ \epsilon \\
\quad \text{where} \quad \eta_1\ ()\ y = y = \epsilon \to 0; 1 \\
\qquad\qquad\quad \eta_2\ (x,p)\ y = max\ (p\ y)\ h \\
\qquad\qquad\qquad \text{where} \quad h = arc\ y\ x \wedge y \neq \epsilon \to 1 + p\ x; p\ x
\end{array}
$$

Our program

By in-lining the last derived accumulation in Gofer, we get the following program. Note that we assign a positive number as the identifier of each vertex, and we assume that ϵ has the identifier of 0.

```
type Vertex = Int
psp :: [Vertex] -> Int
psp xs = acc xs 0
acc [] y = if y==0 then 0 else 1
acc (x:xs) y = max (acc xs y) h
    where h = if ((arc y x)&&(y/=0))
              then (1+acc xs x)
              else (acc xs x)
arc 0 v = True
arc v 0 = False
```

Comparing with the initial specification, we can see that substantial progress has been made. Careful readers may have found that if the above program is implemented naively, it still requires exponential time to give its answer. But this is not a problem. Different from the initial program, our derived program is suitable to be made optimized by some standard techniques such as tabulation [Bir80] or memoisation[Mic68]. Since these discussions are beyond the scope of this paper, we omit it here. Alternatively, a Gofer system with embedded memoisation mechanism [Yam95] can give a direct efficient implementation.

As maybe easily verified, there are only $O(n^2)$ distinct values of *acc* requiring in the computation of *psp xs*, where $n = length(xs)$. So our final program brings the running time down to $O(n^2)$ if we ignore the time for manipulating memo-table.

6.5 Related Work and Conclusions

Much work has been devoted to the "forcing strategy" for the derivation of an accumulation. Pettorossi[PS87, PP93] showed, through many examples, how to generalize functions in case folding fails. All these studies are search-based program transformation rather than our calculational based transformation.

Our work was greatly inspired by Bird's[Bir84] pioneer work where he treated promotion as the guide strategy for achieving efficiency, and the parameter accumulation is the method by which promotion is effected. We improve Bird's work in three respects. First, we have extended his strategy from lists to any data type based on the categorical theory of data type. Secondly, we have shown that some of the Bird's explanation of where to generalize and how to proceed program transformation can be made more systematic by program calculation in a theorem-driven manner. Thirdly, we provides general rules for manipulating accumulations to obtain new ones.

This work concerning higher order catamorphisms was much influenced by Fokkinga and Meijer's specification of attribute grammar with catamorphisms [FJMM90]. But their interest is in specification while we are interested in calculation. Another interesting work is from Meijer[Mei92] who proposed the following promotion theorem for higher order catamorphisms for calculating compiler.

$$\frac{F(\circ g)\, a = F(f \circ)\, b \Rightarrow \phi\, a \circ g = f \circ \psi\, b}{f \circ (\![\psi]\!)\, xs = (\![\phi]\!)\, xs \circ g}$$

Comparing with our Accumulation Promotion Theorems, it has too many free parameters which make derivation difficult. In fact, the above theorem can be obtained from Theorem 23 and Theorem 24.

We are also related to Sheard's work [SF93] where he discussed to some extent about transformation of higher order catamorphisms. But his interest is in how to remove computations that are not amenable to his normalization algorithm. In addition, his promotion theorem for higher order catamorphisms are not so general as ours.

Chapter 7

Calculating Monadic Programs

Monads[Mog90, Wad92] are becoming an increasingly important tool for structural functional programming[JD93, KW93, LHJ95], because they provide a uniform framework for describing a wide range of programming features including, for example, state, I/O, continuations, exceptions, parsing and non-determinism, without leaving the framework of a purely functional language. By using monads we can separate the concern of exceptions or states or other features from program construction, leading to a program with clear structure (see Section 3.2 for an example). However, resulting programs may be so inefficient that further program transformation is necessary.

Many studies have been devoted to the transformation of programs without monads. One promising technique is to code the control structure of programs based on the input data type structure. It has been shown that each data type comes equipped with a catamorphism[MFP91, MH95] (i.e., a generalized fold[SF93]) which satisfies several laws being very useful for program transformation. Can this approach be applied to the transformation of programs using monads (which will be called *monadic programs* hereafter)?

It requires paying much attention to do so because the control structure of a monadic program depends not only on its input data type structure but also on its output data type structure (i.e., monadic structure). Theoretically, Fokkinga[Fok94] derived a sufficient assumption on monad under which there is a kind of so-called monadic catamorphisms which satisfy similar laws and can thus be used for transformation of monadic programs. Recently, Meijer and Jeuring[MJ95] discussed the use of monadic catamorphisms in practical functional programming. They convincingly demonstrated through many examples that by programming in this style, resulting monadic programs are of an astonishing clarity, conciseness and manipulability.

Categorically speaking, monadic catamorphisms are the lifting of normal catamorphisms to the Kleisli Category[Fok94]. This is also the approach taken in the Algebraic Design Language (ADL)[KL94]. Despite its mathematical elegance, Fokkinga's theory contains an assumption on monad that is not valid for several known monads, in particular it is not valid for state monad, I/O monad etc. For these monads, therefore the whole reasoning of Fokkinga's doesn't make sense (see the conclusion in his paper[Fok94]). Meijer and Jeuring[MJ95] solved this problem informally in their case example of deriving an efficient abstract G-machine from an initial naive monadic program related to the state monad; they defined their specific monadic catamorphism in terms of a normal catamorphism and derived the specific transformation rules based on the *fusion laws* (i.e., *promotion theorem*) equipped for the normal catamorphism. However, a general study on direct transformation of monadic catamorphisms was not given.

In this chapter, we put forward a new theory on monadic catamorphism by moving Fokkinga's assumption on monad to the condition of a map between monadic algebras so that our theory is valid for arbitrary monads including, for example, the state monad that is not allowed in Fokkinga's theory. Our theory covers Fokkinga's as a special case but can be applied to a wider class of monadic programs. Moreover, we shall show that Meijer and Jeuring's case study is actually an instance of our theory.

This chapter is organized as follows. We begin by demonstrating monadic programming in Section 7.1. In Section 7.2, we propose our construction of the category of monadic F-algebras, give the definition of monadic catamorphisms, and present the promotion theorem for the transformation of monadic programs. An example of an instance of our theory for calculating G-machine is given in Section 7.3.

7.1 Monadic Programming

In this section, we briefly review the basic concept of monads and show a simple example of how to write programs using monads.

7.1.1 Monads

Wadler[Wad92] defines a monad as a unary type constructor M together with two functions *result* and *bind* whose types are given by:

$$\begin{array}{lcl} result & :: & a \to M\ a \\ bind & :: & M\ a \to (a \to M\ b) \to M\ b. \end{array}$$

In addition, these two functions are required to satisfy a number of laws. The left-unit law and right-unit law remove occurrences of *result* from an expression.

$$\begin{array}{lcl} result\ a\ \text{`}bind\text{`}\ k & = & k\ a \\ m\ \text{`}bind\text{`}\ result & = & m \end{array}$$

Furthermore, *bind* has to be associative, i.e.,

$$(m\ \text{`}bind\text{`}\ \lambda x \to f\,x)\ \text{`}bind\text{`}\ g = m\ \text{`}bind\text{`}(\lambda x \to f\ x\ \text{`}bind\text{`}\ g).$$

For instance, the exception monad used to model programs with exception is defined by the type constructor $Maybe$

$$Maybe\ a = Just\ a \mid Nothing$$

with the two functions *result* and *bind* defined by

$$\begin{array}{lcl} result\ a & = & Just\ a \\ Nothing\ `bind`\ f & = & Nothing \\ Just\ a\ `bind`\ f & = & f\ a. \end{array}$$

An intersting use of monads is to model programs that make use of an internal state. Computation of this kind can be represented by function of type $s \to (s, a)$ (often referred to as *state transformer*) mapping an initial state to a pair containing the next state and the result. It can be defined as the monad $State\ s$ as follows.

$$\begin{array}{lcl} State\ s\ a & = & s \to (s, a) \\ result\ a & = & \lambda s \to (s, a) \\ m\ `bind`\ f & = & \lambda s \to \textbf{let}\ (s', a) = m\ s\ \textbf{in}\ f\ a\ s'. \end{array}$$

Definition 14 (M-monadic Function) Let M be a monad. A function is said to be *M-monadic* if its type is $a \to M\ b$ for some types a and b. □

If M is clear from the context, we may simply say monadic function instead of M-monadic function. As will be seen later, the well known *Kleisli Category* concerning monadic functions is very important in our theory.

7.1.2 An Example of Programming with Monads

To help understanding the use of monads in programming and the motivation of promotional transformation on monadic programs, consider the following simple problem. Given a binary tree in which each leaf has a name and given a list of pairs associating names with values, we are asked to calculate the sum of values associated with tree leaves. For example, if the tree is $Node\ (Leaf$ "good"$, Leaf$ "luck"$)$ and the associated list is [("good", 2), ("luck", 4)], we hope to give the result of 6. If there is a leaf whose associated value does not exist in the associated list, an error should be returned. We may solve the problem by defining a function *st*:

$$st :: Tree\ String \to [(String, Value)] \to Maybe\ Value.$$

Although the problem is simple, programming is not so easy because it usually has to carry the association list explicitly all the way and to pay much attention to any possibility of error happening. To avoid this, we may define a suitable monad so that programs can be constructed in a quite simple way over this monadic computation. So we define the monad $RM\ r$ as follows.

$$\begin{array}{lcl} RM\ r\ x & = & r \rightarrow Maybe\ x \\ result\ x & = & \lambda\_ \rightarrow Just\ x \\ m\ `bind`\ f & = & \lambda r \rightarrow m\ r\ `bind`\ \lambda a \rightarrow f\ a\ r \\ zero & = & \lambda\_ \rightarrow Nothing \end{array}$$

The monad $RM\ r$ is in fact a composition of two standard monads the *Reader Monad* and the *Exception Monad*[JD93]. Now we can rewrite the type of st to

$$st :: Tree\ String \rightarrow RM\ [(String, Value)]\ Value.$$

The st takes a tree of type $Tree\ String$ and yields a computation represented by the monad $RM\ [(String, Value)]$. The result of this computation is a value.

According to the specification of the problem, we define st concisely as follows.

$$\begin{array}{lcl} st\ t & = & subst\ t\ `bind`\ \lambda t' \rightarrow result\ (sumtr\ t') \\ \\ subst & :: & Tree\ String \rightarrow RM\ [(String, Value)]\ (Tree\ Value) \\ subst\ (Leaf\ a) & = & lookup\ a\ `bind`\ \lambda b \rightarrow result\ (Leaf\ b) \\ subst\ (Node\ (l, r)) & = & subst\ l\ `bind`\ \lambda l' \rightarrow subst\ r\ `bind`\ \lambda r' \rightarrow \\ & & result\ (Node\ (l', r')) \\ \\ sumtr & = & Tree\ Value \rightarrow Value \\ sumtr & = & (\!|id, (+)|\!) \end{array}$$

It uses two functions: $subst$ for replacing each leaf in the tree with its associated value and $sumtr$ for summing up all leaves. Besides, it uses a monadic function

$$lookup :: String \rightarrow RM\ [(String, Value)]\ Value$$

which returns $result \circ v$ if the name has associated value v and returns $zero$ otherwise. It is worth noting here that we do not need to care about error happening and do not need to carry explicitly the association list all the way in the definition of st.

One big problem with functional programs is that some intermediate results might be produced and then be consumed, resulting in inefficient programs[TM95, Wad88]. Monadic programs are not an exception. For example, the definition of st has two parts composed with $bind$; $subst\ t$ produces a whole tree which is then consumed by $result \circ sumtr$. Can we fuse them together to make it efficient?

Fokkinga[Fok94] solved this problem by structuring monadic programs with monadic catamorphisms for which there is a general promotion rule for fusing monadic programs. Although we can make st efficient by his theory, it contains an assumption on monads that is not valid for manipulating many monadic programs (see Section 5). In this paper, we aim to give a more general theory for fusing monadic programs.

7.2 Theory for Manipulating Monadic Programs

We aim at a general theory on promotional transformation of *monadic* programs. Before doing so, let's recall how we have done for *normal* programs as in Section 2. First of all, we have the base category C. We then build the category of F-algebras upon C. Finally, we define catamorphisms as homomorphisms from the initial F-algebra to another F-algebra and derive its Promotion Theorem. In this section, we shall follow this train of thought, defining monadic catamorphisms and constructing a general theory for manipulating monadic programs. In what follows, we assume that F is an endofunctor from C to C, and that (T, in_F) is the initial F-algebra.

7.2.1 The Base Category for Monadic Catamorphisms

From the viewpoint of typing, a so-called monadic catamorphism[Fok94, MJ95], denoted by $\langle\!\lbrack \_ \rbrack\!\rangle$, should act on *monadic functions*:

$$\frac{\phi :: F\,X \to M\,X}{\langle\!\lbrack \phi \rbrack\!\rangle :: T \to M\,X}$$

Thus the base category for monadic catamorphisms would be naturally *Kleisli Category* whose morphisms are monadic functions.

Definition 15 (Base Category C^M) Let M be a monad with *bind* and *result* operations. The base category for monadic catamorphisms, denoted as C^M, is defined as a Kleisli category[BW90]:

- Whose objects are sets;
- Whose morphisms are monadic functions, i.e., given two objects X and Y, the morphism from X to Y is the monadic function with type $X \to M\,Y$;
- Whose associative composition operator is @, defined by

$$f \,@\, g = \lambda x \to g\ x\ \text{`}bind\text{`}\ f;$$

- Whose identity is the monadic function *result*. □

Notice the difference between morphisms and monadic functions in C^M. The monadic function of type $X \to M\,Y$ denotes a morphism from object X to object Y instead of a morphism from X to $M\,Y$. To make this difference clear, we may use :: for typing of functions and and : for mapping of morphisms.

Recall that our default category C has as objects sets and has as morphisms functions from one type to another. Now, we know that C^M has as objects sets but has as morphisms monadic functions. In fact, there is an adjunction[BW90] between C and C^M, which is quite important since it provides means to discuss properties of C^M in terms of C.

7.2.2 Category of Monadic F-Algebras

To define monadic catamorphisms and the Promotion (Fusion) Theorem, we need to construct a category with monadic algebras as objects and monadic catamorphisms as some of its morphisms. To this end, first we define monadic algebras.

Definition 16 (Monadic F-algebra) A monadic F-algebra is a pair (X, ϕ), where X is an object in C^M and ϕ is a monadic function denoting a morphism in C^M from object $F\,X$ to X. $\square$

For example, $(T, result \circ in)$ is a monadic F-algebra. Note that in the above definition we can apply F on an object of category C^M because we know C and C^M have the same objects.

Second, we set about to define the structure-preserving map between monadic F-algebras. Since we hope such a map could be a morphism in our future category of monadic algebras ($Alg(F,*)$), we have to define it carefully while imposing as less restriction as possible. To capture the structure of monadic F-algebras, we define a derivation of F, denoted by F^*, inducting over the construction of functor F. It maps a monadic function to another one, i.e., $F^* :: (X \to M\,Y) \to (F\,X \to M\,(F\,Y))$.

Definition 17 (F^*)

$$\begin{array}{llll}
F = F_1 + F_2 & \Rightarrow & F^*\,f & = \; F_1^*\;f + F_2^*\;f \\
F = F_1 \times F_2 & \Rightarrow & F^*\,f\;(x_1,x_2) & = \; F_1^*\;f\;x_1\;\text{`}bind\text{`}\;\lambda y_1 \to \\
 & & & \; F_2^*\;f\;x_2\;\text{`}bind\text{`}\;\lambda y_2 \to result\;(y_1,y_2) \\
F = I & \Rightarrow & F^*\,f & = \; f \\
F = !a & \Rightarrow & F^*\,f & = \; result
\end{array}$$

$\square$

Note the definition of F^* for the case of $F = F_1 \times F_2$. It determines a computing order from $F^*\,f\;x_1$ to $F^*\,f\;x_2$. As a matter of fact, we may change this order to give other proper definitions of F^*. The two properties that F^* has to possess are:

- **Identity property**: $F^*\,result = result$;
- **Separable property**: there exists a unique d_{F^*} such that $F^*\,f = d_{F^*} \circ F\,f$.

Obviously, the above definition of F^* satisfies the identity property. For the separable property, $d_{F^*} = F^*\,id$, which can be in-lined as follows.

$$\begin{array}{llll}
F = F_1 + F_2 & \Rightarrow & d_{F^*}\,(x_1 + x_2) & = \; d_{F^*}\,x_1 + d_{F^*}\,x_2 \\
F = F_1 \times F_2 & \Rightarrow & d_{F^*}\,(x_1,x_2) & = \; d_{F^*}\,x_1\;\text{`}bind\text{`}\;\lambda y_1 \to \\
 & & & \; d_{F^*}\,x_2\;\text{`}bind\text{`}\;\lambda y_2 \to result\;(y_1,y_2) \\
F = I & \Rightarrow & d_{F^*}\,x & = \; x \\
F = !a & \Rightarrow & d_{F^*}\,x & = \; result\;x
\end{array}$$

It should be noted that F^* is not necessary to be a functor satisfying $F^*(f \mathbin{@} g) = F^*f \mathbin{@} F^*g$ for any f and g, the assumption in Fokkinga's theory[Fok94]. It is this assumption that makes his theory invalid for several known monads. Instead, we shift his assumption to the condition of a structure-preserving map (see Definition 18 below) and make our theory valid for arbitrary monads. We will return to this point later.

Example 12 (F_T^*) Consider the functor F_T defined in Section 2.3, the F_T^* is as follows.

$$\begin{array}{rcl} F_T^*\ f & = & \mathit{result} + \lambda(x_1, x_2) \to f\ x_1\ \text{`}\mathit{bind}\text{`}\ \lambda y_1 \to \\ & & \qquad f\ x_2\ \text{`}\mathit{bind}\text{`}\ \lambda y_2 \to \mathit{result}\ (y_1, y_2) \end{array}$$

$\square$

After obtaining F^*, we can define the structure-preserving map between two monadic F-algebras.

Definition 18 (Structure-preserving Map) Let (X, ϕ) and (Y, ψ) be two monadic F-algebras. A structure-preserving map from (X, ϕ) to (Y, ψ) is a morphism h from object X to object Y in the category C^M satisfying

$$\begin{array}{llcll} (1) & h \mathbin{@} \phi & = & \psi \mathbin{@} F^*h & \\ (2) & F^*h \mathbin{@} F^*g & = & F^*(h \mathbin{@} g) & \text{for any } g \end{array}$$

$\square$

Our definition of structure-preserving map has an additional condition (2), compared with Fokkinga's. At the first glance, it seems that we impose more restriction. On the contrary, we have less restriction, because Fokkinga actually required that $F^*h \mathbin{@} F^*g = F^*(f \mathbin{@} g)$ for any monadic functions h and g, while we only require that this equation for a *specific* monadic function h holds for any monadic function g.

Finally, we construct the category of monadic F-algebras with respect to a monad M.

Theorem 25 (Category of monadic F-algebras) The following is the construction of monadic F-algebras, denoted by $Alg(F, *)$.

- It has as objects monadic F-algebras.
- It has as morphisms structure-preserving maps.
- The composition of morphisms is @.
- The identity morphism is *result*.

Proof: To see that the above construction establishes a category, we have to prove that (a) The *result* is a morphism, i.e., a structure-preserving map; (b) Given two morphisms $h_1 : (X, \phi) \to (Y, \psi)$ and $h_2 : (Y, \psi) \to (Z, \eta)$, then $h_2 \mathbin{@} h_1$ is a morphism from (X, ϕ) to (Z, η); (c) The composition @ is associative; (d) $\mathit{result} \mathbin{@} h = h \mathbin{@} \mathit{result} = h$.

Since our category is built upon C^M, (c) and (d) obviously hold. To prove (a), we have to show that *result* satisfies two conditions for being a morphism, as is easily verified. To prove (b), we proceed the following two calculations to show that $h_2 \,@\, h_1$ satisfies two conditions for being a morphism.

$$\begin{array}{ll} & h_2 \,@\, h_1 \,@\, \phi \\ = & \quad \{ \text{ condition (1) for } h_1 \text{ being a morphism } \} \\ & h_2 \,@\, \psi \,@\, F^* h_1 \\ = & \quad \{ \text{ condition (1) for } h_2 \text{ being a morphism } \} \\ & \eta \,@\, F^* h_2 \,@\, F^* h_1 \\ = & \quad \{ \text{ condition (2) for } h_2 \text{ being a morphism } \} \\ & \eta \,@\, F^*(h_2 \,@\, h_1) \end{array}$$

$$\begin{array}{ll} & F^*(h_2 \,@\, h_1) \,@\, F^* g \\ = & \quad \{ \text{ condition (2) for } h_2 \text{ being a morphism } \} \\ & F^* h_2 \,@\, F^* h_1 \,@\, F^* g \\ = & \quad \{ \text{ condition (2) for } h_1 \text{ being a morphism } \} \\ & F^* h_2 \,@\, F^*(h_1 \,@\, g) \\ = & \quad \{ \text{ condition (2) for } h_2 \text{ being a morphism } \} \\ & F^*(h_2 \,@\, h_1 \,@\, g) \end{array}$$

□

7.2.3 Monadic Catamorphisms

In this section, we shall define monadic catamorphisms and give the Promotion (Fusion) Theorem for them.

Proposition 26 (Monadic Catamorphism) Let (T, in) be the initial F-algebra, F^* be a derivation of F with respect to a monad M. For any monadic F-algebra (X, ϕ), there exists a unique monadic function $h :: T \to M\, X$ satisfying

$$h \circ in = \phi \,@\, F^* h$$

Here h will be called *monadic catamorphism* and will be denoted by $\langle\!\lbrack \phi \rbrack\!\rangle_{F^*}$. If F^* is clear from the context, we may omit the subscript F^* in $\langle\!\lbrack \_ \rbrack\!\rangle_{F^*}$.

Proof: To prove it, we calculate the equation and find the explicit definition of h as follows.

$$\begin{array}{ll} & h \circ in = \phi \,@\, F^* h \\ \equiv & \quad \{ \text{ Separable property of } F^* \ \} \\ & h \circ in = \phi \,@(d_{F^*} \circ F\, h) \\ \equiv & \quad \{ \text{ by the fact } f \,@(g \circ h) = (f \,@\, g) \circ h \ \} \\ & h \circ in = (\phi \,@\, d_{F^*}) \circ F\, h \\ \equiv & \quad \{ \text{ uniqueness of the normal catamorphism } \} \\ & h = (\!\lbrack \phi \,@\, d_{F^*} \rbrack\!)_F \end{array}$$

□

Example 13 (Monadic catamorphism over tree) The *subst* given in Section 3.2 is a monadic catamorphism over tree data structure.

$$\begin{array}{lll} subst & = & (\!|\phi_1, \phi_2|\!)_{F_T^*} \\ & & \phi_1\ a\ = lookup\ a\ \text{`}bind\text{`}\ \lambda b \rightarrow result(Leaf\ b) \\ & & \phi_2\ (l, r)\ =\ result\ (Node\ (l, r)) \end{array}$$

□

The promotion (fusion) theorem for manipulating monadic catamorphisms is as follows.

Theorem 27 (Monadic Promotion (Fusion)) If h is a morphism from (X, ϕ) to (Y, ψ) in the category $Alg(F, *)$, i.e.,

$$\begin{array}{llll} (1) & h \,@\, \phi & = & \psi \,@\, F^* h \\ (2) & F^*\, h\ @\ F^*\, g & = & F^*(h \,@\, g) \quad \text{for any morphism } g \end{array}$$

then

$$h \,@ (\!|\phi|\!) = (\!|\psi|\!).$$

Proof: We prove the theorem by the following calculation.

$$\begin{array}{ll} & h \,@ (\!|\phi|\!) = (\!|\psi|\!) \\ \equiv & \quad \{\ \text{Proposition 26}\ \} \\ & h \,@ (\!|\phi|\!) \circ in = \psi \,@\, F^*(h \,@ (\!|\phi|\!)) \\ \equiv & \quad \{\ \ \text{Condition (2)}\ \ \} \\ & h \,@ (\!|\phi|\!) \circ in = \psi \,@\, F^* h \,@\, F^* (\!|\phi|\!) \\ \equiv & \quad \{\ \text{Proposition 26}\ \} \\ & h \,@\, \phi \,@\, F^* (\!|\phi|\!) = \psi \,@\, F^* h \,@\, F^* (\!|\phi|\!) \\ \Leftarrow & \quad \{\ \ \text{trivial}\ \ \} \\ & h \,@\, \phi = \psi \,@\, F^* h \end{array}$$

□

Why is our promotion theorem able to be applied to a wider class of monadic functions than Fokkinga's? Recall that his assumption, $F^* h \,@\, F^* g = F^*(h \,@\, g)$ for any monadic functions h and g, is an implicit condition in his promotion theorem. We believe that h is not necessary to range over all functions. In practical program calculation, h is usually given and asked to be promoted into a monadic catamorphism. Therefore, as in our promotion theorem, we only requires that for a *specific* monadic function h and any monadic function g, $F^* h \,@\, F^* g = F^*(h \,@\, g)$.

Example 14 (Promotional Transformation on *st*) Recall the definition of *st* in Section 3.2:

$$st\ t = subst\ t\ \text{`}bind\text{`}\ \lambda t' \rightarrow result\ (sumtr\ t').$$

We know, as in Example 13, that *subst* is a monadic catamorphism. So we can rewrite *st* to the following.

$$st = (result \circ sumtr) \,@\, (\!|\phi_1 \triangledown \phi_2|\!)_{F_T^*}$$

Based on Theorem 27, we can derive the following efficient monadic program[HIT95b]:

$$\begin{array}{rcl} st & = & (\!|\psi_1, \psi_2|\!) \\ & & \psi_1 = result \,@\, lookup \\ & & \psi_2 = result \circ +. \end{array}$$

□

7.3 An Example

In this section, we adopt the impressive example provided by Meijer and Jeuring[MJ95], which calculates efficient G-machine. We intend to show their four promotable conditions, informally provided and acted as the kernel strategy in their calculation, are just an instance of our promotion theorem. We will not address the calculation already done by Meijer and Jeuring[MJ95] and only explain our main idea.

G-Machine

In the definition of G-machine, a state monad is used. Sharing and graph manipulation that goes on in a real graph reduction implementation can be modeled using the state monad whose state is a graph and whose computation result is a pointer (represented by *Int*) to a node of the graph. An naive G-machine evaluates an expression e by first building a graph for e and then evaluating the resulting graph:

$$\begin{array}{lcl} machine & :: & [Int] \to Expr \to State\ Graph\ Int \\ machine\ ps\ e & = & (eval \,@(build\ ps))\ e \end{array}$$

Evaluating an expression is rather inefficient: *build ps* builds a complete graph, which is subsequently consumed by function *eval*.

Since the monadic program *machine* is defined over the state monad, Fokkinga's theory cannot be used to calculate it to an efficient one. In the following we shall demonstrate an instance of our theory for this specific problem, giving the definition of monadic catamorphisms over *Expr* and proposing the corresponding promotion theorem.

Expressing *build ps* by a Monadic Catamorphism over *Expr*

Consider a tiny first order language[MJ95] in which all values are integers, and the only operator is addition. Formal parameters are encoded by de Bruijn indices. The expressions are elements of the data type *Expr*:

$$\begin{array}{rcll} Expr & = & Var\ Int & \text{Variable} \\ & \mid & Con\ Int & \text{Constant} \\ & \mid & Add\ (Expr, Expr) & \text{Addition} \\ & \mid & App\ (Name, Expr) & \text{Function application} \end{array}$$

This data type can be defined as the initial F-algebra $(Expr, in_F)$ where

$$in_F = Var \triangledown Con \triangledown Add \triangledown App$$

and F is an endofunctor defined by

$$F \ = !Int + !Int \ + \ I \times I + !Name \times I.$$

Now we can define a proper derivation of F as follows.

$$\begin{array}{rcl} F^* f & = & result \ \triangledown \\ & & result \ \triangledown \\ & & \lambda(e_1, e_2) \rightarrow f\ e_1\ \text{`}bind\text{`}\ \lambda e_1' \rightarrow \\ & & \qquad\qquad f\ e_2\ \text{`}bind\text{`}\ \lambda e_2' \rightarrow result\ (e_1', e_2') \quad \triangledown \\ & & \lambda(n, es) \rightarrow f\ es\ \text{`}bind\text{`}\ \lambda es' \rightarrow result\ (n, es') \end{array}$$

According to Proposition 26, a monadic catamorphism over *Expr* is defined by

$$\begin{array}{lcl} (\!|\phi_1 \triangledown \phi_2 \triangledown \phi_3 \triangledown \phi_4|\!) & :: & Expr \rightarrow M\ Y \\ (\!|\phi_1 \triangledown \phi_2 \triangledown \phi_3 \triangledown \phi_4|\!)\ (Var\ v) & = & \phi_1\ v \\ (\!|\phi_1 \triangledown \phi_2 \triangledown \phi_3 \triangledown \phi_4|\!)\ (Con\ c) & = & \phi_2\ c \\ (\!|\phi_1 \triangledown \phi_2 \triangledown \phi_3 \triangledown \phi_4|\!)\ (Add\ (e_1, e_2)) & = & (\!|\phi_1 \triangledown \phi_2 \triangledown \phi_3 \triangledown \phi_4|\!)\ e_1\ \text{`}bind\text{`}\ \lambda e_1' \rightarrow \\ & & (\!|\phi_1 \triangledown \phi_2 \triangledown \phi_3 \triangledown \phi_4|\!)\ e_2\ \text{`}bind\text{`}\ \lambda e_2' \rightarrow \\ & & \phi_3\ (e_1', e_2') \\ (\!|\phi_1 \triangledown \phi_2 \triangledown \phi_3 \triangledown \phi_4|\!)\ (App\ (n, e)) & = & (\!|\phi_1 \triangledown \phi_2 \triangledown \phi_3 \triangledown \phi_4|\!)\ e\ \text{`}bind\text{`}\ \lambda e' \rightarrow \\ & & \phi_4\ (n, e') \end{array}$$

The above monadic catamorphism over *Expr* is exactly the same as what Meijer and Jeuring gave except that it is defined directly rather than in terms of a normal catamorphism. In fact, *build ps* can be defined as such a monadic catamorphism[MJ95]. We omit the detail but assume that

$$build\ ps = (\!|\phi_1 \triangledown \phi_2 \triangledown \phi_3 \triangledown \phi_4|\!).$$

Promoting *eval* into *build ps*

Now we are faced with a monadic program *eval* @(*build ps*), and we hope to promote *eval* into *build ps* to obtain an efficient one. As we know $build\ ps = (\!|\phi_1 \triangledown \phi_2 \triangledown \phi_3 \triangledown \phi_4|\!)$, we can derive the general transformation rule for this calculation by instantiating Theorem 27.

Our rule says that $eval\,@(build\ ps) = (\!|\psi_1 \triangledown \psi_2 \triangledown \psi_3 \triangledown \psi_4|\!)$, provided that

$$\begin{array}{lll}
(a) & eval\,@\,\phi_1 & = \psi_1 \\
(b) & eval\,@\,\phi_2 & = \psi_2 \\
(c_1) & eval\,@\,\phi_3 & = \lambda(e_1, e_2) \to eval\ e_1\ \text{`}bind\text{`}\ \lambda e_1' \to \\
 & & \qquad\qquad eval\ e_2\ \text{`}bind\text{`}\ \lambda e_2' \to \\
 & & \qquad\qquad \psi_3\ (e_1', e_2') \\
(c_2) & F^* eval\,@\,F^*\,g & = F^*\,(eval\,@\,g) \quad \text{for any } g \\
(d) & eval\,@\,\phi_4 & = \lambda(n, e) \to eval\ e\ \text{`}bind\text{`}\ \lambda e' \to \psi_4\ (n, e'),
\end{array}$$

which is essentially the same as the four conditions informally provided by Meijer and Jeuring[MJ95]. Note that we have (c_1) and (c_2) which implies Meijer and Jeuring's corresponding one condition.

Using the above rule, we can go on deriving ψ_i's and thus obtain an efficient monadic program for G-machine[MJ95].

7.4 Conclusion and Future Work

In this chapter, we propose a new theory ([HI95] for monadic catamorphism by moving Fokkinga's assumption on the monad to the condition on a map between monadic algebras so that our theory is valid for arbitrary monads including, for example, the state monad that is not allowed in Fokkinga's theory. Our theory covers Fokkinga's as a special case but can be applied to a wider class of monadic programs.

So far, we have only studied the transformation of monadic programs with respect to one monad. We are trying to fuse monadic programs with respect to different monads. Besides, we believe that our work on monadic catamorphisms can be extended to its dual, *monadic anamorphism*.

Chapter 8

Calculating Parallel Functional Programs

This chapter shows that by constructing efficient catamorphisms which have high inherited parallelism one can succeed in deriving efficient parallel functional programs. Rather than giving a general study, we focus ourselves on the construction of a special catamorphism [1], called *list homomorphism*, based on the calculational approach.

List homomorphisms [Bir87] are those functions on finite lists that *promote* through list concatenation – that is, functions h for which there exists a binary operator $\oplus$ such that, for all finite lists xs and ys,

$$h\ (xs \mathbin{+\!\!\!+} ys) = h\,xs \oplus h\,ys$$

where $\mathbin{+\!\!\!+}$ denotes list concatenation. Examples of list homomorphisms are simple functions, such as *sum* and *max* which return the sum and the largest of the elements of a list respectively.

Intuitively, the definition of list homomorphisms implies that the computations of $h\,xs$ and $h\,ys$ are independent each other and can be carried out in parallel, which can be viewed as expressing the well-known divide-and-conquer paradigm of parallel programming. Therefore, the implications for parallel program derivation are clear; if the problem is a list homomorphism, then it only remains to define an efficient $\oplus$ in order to produce a highly parallel solution.

However, there are some useful and interesting list functions that are not list homomorphisms. One example is the function *mss*, which finds the sum of contiguous segment within a list whose members have the largest sum among all segments. For example, we have

$$mss\ [3, -4, 2, -1, 6, -3] = 7$$

where the result is contributed by the segment $[2, -1, 6]$. The *mss* is not a list homo-

[1] Note it is indeed a catamorphism over append lists.

morphism, since knowing $mss\ xs$ and $mss\ ys$ is not enough to allow computation of $mss\ (xs +\!\!+\, ys)$.

To solve this problem, Cole [Col93] proposed an informal approach showing how to embed these functions into list homomorphisms. His method consists of constructing a homomorphism as a tuple of functions where the original function is one of the components. The main difficulty is to guess which functions must be included in a tuple in addition to the original function and to prove that the constructed tuple is indeed a list homomorphism. The examples given by Cole show that this usually requires a lot of ingenuity from the program developer. The purpose of this chapter is to give a formal derivation of such list homomorphisms containing the original non-homomorphic function as its component. Our main contributions are as follows.

- First, unlike Cole's informal study, we propose a *systematic* way of discovering the extra functions which are to be tupled with the original function to construct a list homomorphism. We give two main theorems, the Tupling Theorem and the Almost Fusion Theorem, showing how to derive a "true" list homomorphism from recursively defined functions by means of tupling and how to calculate a new homomorphism from the old by means of fusion.

- Second, our main theorems for tupling and fusion are given in a *calculational* style [MFP91, TM95, HIT96d] rather than being based on the fold/unfold transformation [Chi92, Chi93]. Therefore, infinite unfoldings, once inherited in the fold/unfold transformation, can be definitely avoided by the theorems themselves. Furthermore, although we restrict ourselves to list homomorphisms, our theorems could be extended naturally for homomorphisms of arbitrary data structures (e.g., trees) with the theory of *constructive algorithmics* [Fok92a].

- Third, our derivation of parallel program proceeds in a *formal* way, leading to a *correct* solution with respect to the initial specification. We start with a simple, and "obviously" correct, but possibly inefficient solution to the problem, and then transform it based on our rules and algebraic identities into a semantically equivalent list homomorphism. We demonstrate our method through a non-trivial example: maximum segment sum problem, once informally discussed by Cole [Col93]. As a matter of fact, our approach can be applied to a wide class of interesting problems, e.g., the 2-dimensional maximum segment sum problem [HIT96e].

This chapter is organized as follows. In Section 8.1, we explain the concept of Cole's almost homomorphism and show the difficulty in deriving almost homomorphisms. After showing our specifications in Section 8.2, we focus ourselves on the derivation of parallel programs from specifications with two important theorems, namely the tupling and the fusion theorems, in Section 8.3. Finally, we give the concluding remarks with related work in Section 8.4.

8.1 Almost Homomorphisms

As stated in the introduction, quite a few useful functions are not list homomorphisms. Cole argued informally that some of them can be converted into so-called *almost homomorphisms* [Col93] by tupling them with some extra functions. An almost homomorphism is a composition of a projection function and a list homomorphism.

In fact, it may be surprising to see that every function can be represented in terms of an almost homomorphism[Gor95]. Let k be a non-homomorphic function. Consider a new function g such that $g\,x = (x, k\,x)$. The tuple-function g is homomorphic, i.e., $g\,(xs +\!\!+\, ys) = (xs +\!\!+\, ys,\ k\,(xs +\!\!+\, ys)) = g\,xs \oplus g\,ys$, where $(xs, a) \oplus (ys, b) = (xs +\!\!+\, ys,\ k\,(xs+$ $+\,ys))$, and we have the almost homomorphism for k defined by $k = \pi_2 \circ g = \pi_2 \circ ([g \circ [\cdot], \oplus])$. However, it is quite expensive and meaningless in that it does not make use of the previously computed values a and b) and computes k from scratch! In this sense, we say it is not an expected "true" almost homomorphism.

In order to derive a "true" almost homomorphism, a suitable tuple-function should be carefully defined, making full use of previously computed values. Cole reported several case studies of such derivation with parallel algorithms as a result, and stressed that in each case, the derivation requires a lot of intuition [Col93]. In this chapter, we shall propose a systematic approach to this derivation.

8.2 Specification

We aim at a formal derivation of parallel programs by constructing list homomorphisms including the original problems as its component (i.e., almost homomorphisms). To talk about parallel program derivation, we should be clear about specifications. It is strongly advocated by Bird [Bir87] that specifications should be direct solutions to problems. Therefore, our specification for a problem p will be a simple, and "obviously" correct, but possibly inefficient solution with the form of

$$p = p_n \circ \cdots \circ p_2 \circ p_1 \qquad (8.1)$$

where each p_i is a (recursively defined) function. This reflects our way of solving problems; a (big) problem p may be solved through multiple passes in which a simpler problem p_i is solved by a recursion.

We shall use the maximum segment sum problem *mss* as our running example. This problem is of interest because there are efficient but non-obvious algorithms to compute it, both in sequential [Bir87] and in parallel[CS92, Col93]. An obviously correct solution to the problem is:

$$mss = max^s \circ (sum *^s) \circ segs \qquad (8.2)$$

which is implemented by three passes: computing the set including all of the contiguous segments by *segs*, summing the elements of each by $sum *^s$, and selecting from the set the largest value of the sums by max^s. Clearly, max^s and $sum *^s$ should be functions over *sets*. Similar to lists, a set is either empty $\{\}$, s singleton $\{x\}$, or the union of two sets $s_1 \cup s_2$. The difference lies in that the union operation is not only associative but also commutative and idempotent. The definition style of functions over sets is quite similar to those over lists. For instance, max^s can be defined directly as follows.

$$\begin{array}{lcl} max^s\ \{\} & = & -\infty \\ max^s\ \{x\} & = & x \\ max^s\ (s_1 \cup s_2) & = & max^s\, s_1 \uparrow max^s\, s_2 \end{array}$$

Like the list map operator, the *map* operator over sets, denoted by $*^s$, applies another function to every items in a set (e.g., $sum *^s$ in our case). In the following, we shall focus on defining *segs*.

The *segs*, the function computing the set of all (contiguous) segments of a list, can be recursively defined by:

$$\begin{array}{lcl} segs\ [\,] & = & \{\} \\ segs\ [x] & = & \{[x]\} \\ segs\ (xs +\!\!+\, ys) & = & segs\ xs \cup segs\ ys \cup (tails\ xs\ \mathcal{X}_{+\!\!+}\, inits\ ys). \end{array}$$

The last equation says that all segments of the concatenation of two lists *xs* and *ys* can be obtained from segments in both *xs* and *ys* and the segments produced crosswisely by concatenating all tail segments of *xs* (i.e., the segments in *xs* ending with the last element) with initial segments of *ys* (i.e., the segments in *ys* starting with the the first element). Note that the *inits*, *tails*, and $\mathcal{X}_{+\!\!+}$ are considered as standard functions in [Bir87] (though our definitions are slightly different). The *inits* is a function returning all initial segments of a list, while the *tails* is a function returning all tail segments. They can be defined directly by:

$$\begin{array}{lcl} inits\ [\,] & = & [\,] \\ inits\ [x] & = & [[x]] \\ inits\ (xs +\!\!+\, ys) & = & inits\ xs +\!\!+\, (xs +\!\!+\,) * (inits\ ys) \end{array}$$

and

$$\begin{array}{lcl} tails\ [\,] & = & [\,] \\ tails\ [x] & = & [[x]] \\ tails\ (xs +\!\!+\, ys) & = & (+\!\!+\, ys) * (tails\ xs) +\!\!+\, tails\ ys. \end{array}$$

The operator $\mathcal{X}_\oplus$ is usually called *cross* operator, defined informally by

$$[x_1, \cdots, x_n]\ \mathcal{X}_\oplus\, [y_1, \cdots, y_m] = \{x_1 \oplus y_1, \cdots, x_1 \oplus y_m,\ \cdots,\ x_n \oplus y_1, \cdots, x_n \oplus y_m\},$$

which crosswisely combines elements in two lists with operator $\oplus$. An obvious property with cross operator is

$$(f *^s) \circ \mathcal{X}_\oplus = \mathcal{X}_{f \circ \oplus}. \tag{8.3}$$

So much for the specification of the *mss* problem. It is a naive correct solution to the problem without concerning efficiency and parallelism at all. It will be shown that our method can derive a correct $O(\log n)$ parallel time algorithm by constructing a "true" almost homomorphism.

8.3 Derivation

Our derivation of a "true" almost homomorphism from the specification (8.1) is carried out in the following way: we derive an almost homomorphism for p_1 (a recursion) first, then fuse p_2 with the derived almost homomorphism to obtain another almost homomorphism, and repeat this fusion until p_n is fused. We are confronted with two problems here: (a) How can a "true" almost homomorphism be derived from a recursive definition? (b) How can a new almost homomorphism be calculated from a composition of another function and an old one?

8.3.1 Deriving Almost Homomorphisms

Although some functions cannot be described directly as list homomorphisms, they may be easily described by (mutual) recursive definitions while some other functions might be used (see *segs* in Section 8.2 for an example) [Fok92a]. In this section, we propose a way of deriving almost homomorphisms from such (mutual) recursive definitions, systematically discovering extra functions that should be tupled with the original function to turn it into a "true" list homomorphism. The "true" list homomorphism must fully reuse the previously computed values, as discussed in Section 8.1.

Our approach is based on the following theorem. For notational convenience, we define

$$\Delta_1^n f_i = f_1 \vartriangle f_2 \vartriangle \cdots \vartriangle f_n.$$

Theorem 28 (Tupling) Let $h_1, \cdots, h_n$ be mutually defined as follows.

$$\begin{array}{rcl} h_i\ [\,] & = & \iota_{\oplus_i} \\ h_i\ [x] & = & f_i\, x \\ h_i\ (xs \mathbin{+\!\!+} ys) & = & ((\Delta_1^n h_i)\ xs) \oplus_i ((\Delta_1^n h_i)\ ys) \end{array} \tag{8.4}$$

Then,

$$\Delta_1^n h_i = (\![\Delta_1^n f_i,\ \Delta_1^n \oplus_i]\!)$$

and $(\iota_{\oplus_1}, \cdots, \iota_{\oplus_n})$ is the unit of $\Delta_1^n \oplus_i$.

Proof: According to the definition of list homomorphisms, it is sufficient to prove that

$$\begin{array}{lcl} (\Delta_1^n h_i)\ [\,] & = & (\iota_{\oplus_1}, \cdots, \iota_{\oplus_n}) \\ (\Delta_1^n h_i)\ [x] & = & (\Delta_1^n f_i)\ x \\ (\Delta_1^n h_i)\ (xs \mathbin{+\!\!+} ys) & = & ((\Delta_1^n h_i)\ xs)\ (\Delta_1^n \oplus_i)\ ((\Delta_1^n h_i)\ ys). \end{array}$$

The first two equations are trivial. The last can be proved by the following calculation.

$$
\begin{array}{cl}
 & LHS \\
= & \quad \{ \text{ Def. of } \Delta \text{ and } \triangle \ \} \\
 & (h_1(xs +\!\!+\, ys), \cdots, h_n(xs +\!\!+\, ys)) \\
= & \quad \{ \text{ Def. of } h_i \ \} \\
 & (((\Delta_1^n h_i)\ xs) \oplus_1 ((\Delta_1^n h_i)\ ys),\ \cdots,\ ((\Delta_1^n h_i)\ xs) \oplus_n ((\Delta_1^n h_i)\ ys)) \\
= & \quad \{ \text{ Def. of } \triangle \text{ and } \Delta \ \} \\
 & RHS
\end{array}
$$

□

Theorem 28 says that if h_1 is mutually defined with other functions (i.e., $h_2, \cdots h_n$) which *traverse over the same lists* in the *specific form* of (8.4), then tupling $h_1, \cdots, h_n$ will definitely give a list homomorphism. It follows that h_1 is an almost homomorphism: the projection function π_1 composed with the list homomorphism for the tuple-function. It is worth noting that this style of tupling can avoid repeatedly redundant computations of $h_1, \cdots, h_n$ in the computation of the list homomorphism of $\Delta_1^n h_i$ [Tak87]. That is, all previous computed results by $h_1, \cdots, h_n$ can be fully reused, as expected in "true" almost homomorphisms.

Practically, not all recursive definitions are in the form of (8.4). They, however, can be turned into such form by a simple transformation. Let's see how the tupling theorem works in deriving a "true" almost homomorphism from the definition of *segs* given in Section 8.2.

First, we determine what functions are to be tupled, i.e., $h_1, \cdots, h_n$. As explained above, the functions to be tupled are those which traverse over the same lists in the definitions. So, from the definition of *segs*:

$$segs\ (xs +\!\!+\, ys) \;=\; \underline{segs\ xs} \cup \underline{segs\ ys} \cup (\underline{tails\ xs}\ \mathcal{X}_{+\!\!+}\ \underline{inits\ ys})$$

we know that *segs* needs to be tupled with *tails* and *inits*, because *segs* and *inits* traverse the same list *xs* whereas *segs* and *tails* traverse the same list *ys* as underlined. Going to the definition of *inits*,

$$inits\ (xs +\!\!+\, ys) \;=\; \underline{inits\ xs} +\!\!+ (\underline{xs} +\!\!+) * (inits\ ys)$$

we find that the *inits* needs to be tupled with *id*, the identity function, since $xs = id\ xs$. Similarly, The *tails* needs to be tupled with *id*. Note that *id* is the identity function over lists defined by

$$
\begin{array}{lcl}
id\ [\,] & = & [\,] \\
id\ [x] & = & [x] \\
id\ (xs +\!\!+\, ys) & = & id\ xs +\!\!+ id\ ys
\end{array}
$$

To summarize the above, the functions to be tupled are *segs*, *inits*, *tails*, and *id*, i.e., our tuple function will be $segs \triangle inits \triangle tails \triangle id$.

Next, we rewrite the definitions of the functions in the above tuple to the form of (8.4), i.e., deriving $f_1, \oplus_1$ for *segs*, $f_2, \oplus_2$ for *inits*, $f_3, \oplus_3$ for *tails*, and $f_4, \oplus_4$ for *id*. In fact, this is straightforward: just selecting the corresponding recursive calls from the tuples. From the definition of *segs*, we have

$$\begin{array}{lcl} f_1\ x & = & \{[x]\} \\ (s_1, i_1, t_1, d_1) \oplus_1 (s_2, i_2, t_2, d_2) & = & s_1 \cup s_2 \cup (t_1 \mathcal{X}_{+\!\!+}\, i_2). \end{array}$$

It would be helpful for understanding the above derivation if we notice the following correspondences: s_1 to *segs xs*, i_1 to *inits xs*, t_1 to *tails xs*, d_1 to *id xs*, s_2 to *segs ys*, i_2 to *inits ys*, t_2 to *tails ys*, d_2 to *id ys*. Similarly, for *inits*, *tails* and *id*, we have

$$\begin{array}{lcl} f_2\ x & = & [[x]] \\ (s_1, i_1, t_1, d_1) \oplus_2 (s_2, i_2, t_2, d_2) & = & i_1 +\!\!+ (d_1 +\!\!+) * i_2 \\ f_3\ x & = & [[x]] \\ (s_1, i_1, t_1, d_1) \oplus_3 (s_2, i_2, t_2, d_2) & = & (+\!\!+\, d_2) * t_1 +\!\!+ t_2 \\ f_4\ x & = & [x] \\ (s_1, i_1, t_1, d_1) \oplus_4 (s_2, i_2, t_2, d_2) & = & d_1 +\!\!+ d_2 \end{array}$$

Finally, we apply Theorem 28 and get the following list homomorphism.

$$segs \vartriangle inits \vartriangle tails \vartriangle id = (\!\![\Delta_1^4 f_i, \Delta_1^4 \oplus_i]\!\!)$$

And our almost homomorphism for *segs* is thus obtained:

$$segs = \pi_1 \circ (\!\![\Delta_1^4 f_i, \Delta_1^4 \oplus_i]\!\!). \tag{8.5}$$

It would be intersting to see that the above derivation is practically *mechanical*. Note that the derivation of the unit of the new binary operator (e.g., $\Delta_1^4 \oplus_i$) is omitted because this is trivial; the new tuple function applying to empty list will give exactly this unit (e.g., $(segs \vartriangle inits \vartriangle tails \vartriangle id)\,[\,]$). The derivation of units will be omitted in the rest of this chapter as well.

8.3.2 Fusion with Almost Homomorphisms

In this section, we show how to fuse a function with an almost homomorphism, the second problem (b) listed at the beginning of Section 8.3.

It is well known that list homomorphisms are suitable for program transformation in that there is a general rule called *Fusion Theorem* [Bir87], showing how to fuse a function with a list homomorphism to get another list homomorphism.

Theorem 29 (Fusion) Let h and $(\!\![f, \oplus]\!\!)$ be given. If there exists $\otimes$ such that $\forall x, y.\ h\ (x \oplus y) = h\,x \otimes h\,y$, then $h \circ (\!\![f, \oplus]\!\!) = (\!\![h \circ f, \otimes]\!\!)$. □

This fusion theorem, however, cannot be used directly for our purpose. As seen in (8.5), we usually derive an almost homomorphism and we hope to know how to fuse functions with almost homomorphisms, namely we want to deal with the following case:

$$h \circ (\pi_1 \circ ([\![\Delta_1^n f_i, \Delta_1^n \oplus_i]\!])).$$

We'd like to shift π_1 left and promote h into the list homomorphism. Our fusion theorem for this purpose is given below.

Theorem 30 (Almost Fusion) Let h and $([\![\Delta_1^n f_i,\ \Delta_1^n \oplus_i]\!])$ be given. If there exist $\otimes_i$ $(i = 1, \cdots, n)$ and a map $H = h_1 \times \cdots \times h_n$ where $h_1 = h$ such that for all j,

$$\forall x, y.\ h_i\,(x \oplus_i y) = H\,x \otimes_i H\,y \tag{8.6}$$

then

$$h \circ (\pi_1 \circ ([\![\Delta_1^n f_i, \Delta_1^n \oplus_i]\!])) = \pi_1 \circ ([\![\Delta_1^n (h_i \circ f_i), \Delta_1^n \otimes_i]\!])$$

Proof: We prove it by the following calculation.

$$\begin{array}{ll} & h \circ (\pi_1 \circ ([\![\Delta_1^n f_i, \Delta_1^n \oplus_i]\!])) \\ = & \quad \{\ \text{By } \pi_1 \text{ and } H\ \} \\ & \pi_1 \circ H \circ ([\![\Delta_1^n f_i, \Delta_1^n \oplus_i]\!]) \\ & \quad \{\ \text{Theorem 29, and the proves below}\ \} \\ & \pi_1 \circ ([\![\Delta_1^n (h_i \circ f_i), \Delta_1^n \otimes_i]\!]) \end{array}$$

To complete the above proof, we need to show that for any x and y,

$$\begin{array}{lcl} H\,(x\,(\Delta_1^n \oplus_i)\,y) & = & (H\,x)\,(\Delta_1^n \otimes_i)\,(H\,y) \\ H \circ (\Delta_1^n f_i) & = & \Delta_1^n (h_i \circ f_i) \end{array}$$

The second equation is easy to prove. For the first, we argue that

$$\begin{array}{ll} & LHS \\ = & \quad \{\ \text{Expanding } \Delta, \text{Def. of } \vartriangle\ \} \\ & H\,(x \oplus_1 y, \cdots, x \oplus_n y) \\ = & \quad \{\ \text{Expanding } H, \text{Def. of } \times\ \} \\ & (h_1 (x \oplus_1 y), \cdots, h_n\,(x \oplus_n y)) \\ = & \quad \{\ \text{Assumption}\ \} \\ & (H\,x \otimes_1 H\,y, \cdots, H\,x \otimes_n H\,y) \\ = & \quad \{\ \text{Def. of } \vartriangle, \Delta\ \} \\ & RHS \end{array}$$

□

Theorem 30 suggests a way of fusing a function h with the almost homomorphism $\pi_1 \circ ([\![\Delta_1^n f_i,\ \Delta_1^n \oplus_i]\!])$ in order to get another almost homomorphism; trying to find $h_2, \cdots, h_n$ together with $\oplus_1, \cdots, \oplus_n$ that meet the equation (8.6). Note that, without lose of generality we restrict the projection function of our almost homomorphisms to π_1 in the theorem.

Returning to our running example, recall that we have reached the point:

$$mss = max^s \circ (sum *^s) \circ (\pi_1 \circ (\![\Delta_1^4 f_i, \Delta_1^4 \oplus_i]\!)).$$

We demonstrate how to fuse $sum *^s$ with $\pi_1 \circ (\![\Delta_1^4 f_i, \Delta_1^4 \oplus_i]\!)$ by Theorem 30. Let $H = (sum *^s) \times h_2 \times h_3 \times h_4$ where h_2, h_3, h_4 await to be determined. In addition, we need to derive $\otimes_1$, $\otimes_2$, $\otimes_3$, and $\otimes_4$ based on the following equations according to Theorem 30:

$$\begin{array}{l} sum *^s ((s_1, i_1, t_1, d_1) \oplus_i (s_2, i_2, t_2, d_2)) = \\ \quad (sum *^s s_1, h_2\, i_1, h_3\, t_1, h_4\, d_1) \otimes_i (sum *^s s_2, h_2\, i_2, h_3\, t_2, h_4\, d_2) \quad (i = 1, \cdots, 4). \end{array}$$

Now the derivation procedure becomes clear; calculating each LHS of the above equations to promote $sum *^s$ into s_1 and s_2, and determining the unknown functions (h_i and $\otimes_i$) by matching with its RHS. As an example, consider the following calculation of the LHS of the the equation for $i = 1$.

$$\begin{array}{ll} & (sum *^s)\, ((s_1, i_1, t_1, d_1) \oplus_1 (s_2, i_2, t_2, d_2)) \\ = & \quad \{ \text{ Def. of } \oplus_1 \ \} \\ & (sum *^s)\, (s_1 \cup s_2 \cup (t_1 \mathcal{X}_{+\!\!+}\, i_2)) \\ = & \quad \{ \ f *^s (s1 \cup s2) = f *^s s1 \cup f *^s s2 \ \} \\ & sum *^s s_1 \cup sum *^s s_2 \cup sum *^s (t_1 \mathcal{X}_{+\!\!+}\, i_2) \\ = & \quad \{ \ (8.3) \ \} \\ & (sum *^s s_1 \cup sum *^s s_2 \cup (t_1 \mathcal{X}_{sum \circ +\!\!+}\, i_2)) \\ = & \quad \{ \text{ cross operator, } sum \ \} \\ & (sum * s1 \cup sum * s_2 \cup ((sum * t_1) \mathcal{X}_{+} (sum * i_2))) \end{array}$$

Matching the last expression with its corresponding RHS:

$$(sum *^s s_1,\ h_2\, i_1,\ h_3\, t_1,\ h_4\, d_1) \otimes_1 (sum *^s s_2,\ h_2\, i_2,\ h_3\, t_2,\ h_4\, d_2)$$

will give

$$\begin{array}{l} h_2 = h_3 = sum* \\ (s_1, i_1, t_1, d_1) \otimes_1 (s_2, i_2, t_2, d_2) = s_1 \cup s_2 \cup (t_1 \mathcal{X}_{+} i_2). \end{array}$$

The g1 others can be similarly derived.

$$\begin{array}{lcl} h_4 & = & sum \\ (s_1, i_1, t_1, d_1) \otimes_2 (s_2, i_2, t_2, d_2) & = & i_1 +\!\!+ (d_1 +) * i_2 \\ (s_1, i_1, t_1, d_1) \otimes_3 (s_2, i_2, t_2, d_2) & = & (+ d_2) * t_1 +\!\!+ t_2 \\ (s_1, i_1, t_1, d_1) \otimes_4 (s_2, i_2, t_2, d_2) & = & d_1 + d_2 \end{array}$$

To use Theorem 30, we also need to consider the f part whose results are as follows.

$$\begin{array}{lcll} f_1'\ x & = & ((sum *^s) \circ f_1)\, x & = \{x\} \\ f_2'\ x & = & ((sum *) \circ f_2)\, x & = [x] \\ f_3'\ x & = & ((sum *) \circ f_3)\, x & = [x] \\ f_4'\ x & = & (sum \circ f_1)\, x & = x \end{array}$$

According to Theorem 30, we soon have

$$(sum *^s) \circ segs = \pi_1 \circ ([\![\Delta_1^4 f_i', \Delta_1^4 \otimes_i]\!]).$$

Again, we can fuse max^s with the above almost homomorphism (in this case, $H = max^s \times max \times max \times id$) and get the following almost homomorphism for mss, the final result for mss.

$$mss = \pi_1 \circ ([\![id, \Delta_1^4 \otimes_i']\!])$$

where

$$\begin{array}{l} (s_1, i_1, t_1, d_1) \otimes_1' (s_2, i_2, t_2, d_2) = s1 \uparrow s_2 \uparrow (t_1 + i_2) \\ (s_1, i_1, t_1, d_1) \otimes_2' (s_2, i_2, t_2, d_2) = i_1 \uparrow (d_1 + i_2) \\ (s_1, i_1, t_1, d_1) \otimes_3' (s_2, i_2, t_2, d_2) = (t_1 + d_2) \uparrow t_2 \\ (s_1, i_1, t_1, d_1) \otimes_4' (s_2, i_2, t_2, d_2) = d_1 + d_2 \end{array}$$

Thus we got the same result as informally given by Cole [Col93]. In practical terms, the algorithm looks so promising that on many architectures we can expect an $O(\log n)$ algorithm with $O(n/(\log n))$ processors.

8.4 Concluding Remarks and Related Work

In this chapter, we propose a formal and systematic approach to the derivation of efficient parallel programs from specifications of problems via *manipulation* of almost homomorphisms, namely the construction of almost homomorphisms from recursive definitions (Theorem 28) and the fusion of a function with almost homomorphisms (Theorem 30). It is different from Cole's informal way[Col93].

Tupling and fusion are two well-known techniques for improving programs. Chin [Chi92, Chi93] gave an intensive study on it. His method tries to fuse and/or tuple *arbitrary* functions by *fold/unfold* transformations while keeping track of function calls and using clever control to avoid infinite unfolding. In contrast to his costly and complicated algorithm to keep out of non-termination, our approach makes use of structural knowledge of list homomorphisms and constructs our tupling and fusion rules in a calculational style without worrying about infinite unfoldings.

Our approach to the tupling of mutual recursive definitions is much influenced by the *generalization algorithm* [Tak87, Fok92a]. Takeichi showed how to define a higher order function common to all functions mutually defined so that multiple traversals of the same data structures in the mutual recursive definition can be eliminated. Because higher order functions are suitable for partial evaluation but not good for program derivation, we employ tuple-functions and develop the corresponding fusion theorem.

Construction of list homomorphisms has gained great interest because of its importance in parallel programming. Barnard et.al. [BSS91] applied the Third Homomorphism

Theorem [Gib94] for the language recognition problem. The Third Homomorphism Theorem says that an algorithm h which can be formally described by two specific sequential algorithms (*leftwards* and *rightwards reduction* algorithms) is a list homomorphism. Although the existence of an associative binary operator is guaranteed, the theorem does not address the question of the existence – let alone the construction – of a direct and efficient way of calculating it. To solve this problem, Gorlatch [Gor95] imposed additional restrictions, left associativity and right associativity, on the leftwards and rightwards reduction functions so that an associative binary operator $\oplus$ could be derived in a systematic way. However, finding left-associative binary operators is usually not easier than finding associative operators. In comparison, our derivation is more constructive: we derive list homomorphism directly from mutual recursive representations and then fuse it with other functions.

Chapter 9

Concluding Remarks

Functional programming is claimed to be suitable for writing clear programs, but the clarity of functional programs usually comes at the price of being inefficiency compared with imperative ones. In this thesis, we propose a new method based on the theory of Constructive Algorithmics for the systematic and practical study on optimization of functional programs. By means of program calculation based on several novel transformation laws as well as some automatic algorithms newly developed, we can successfully improve general functional programs clearly written in a compositional and modular way. These laws and algorithms are significant because they make the fusion and the tupling, two most general transformations in optimizing functional programs, be effectively and practically applied. In addition, we demonstrate how to optimize specific functional programs, i.e., functional programs in parallel and functional programs with external effects. All the algorithms and transformation laws have been experimentally implemented and tested intensively, showing its power and its promise.

In the following, we would like to highlight some interesting future work.

First of all, we believe that we are in a position to implement a program calculator system for the optimization of functional programs in a more practical way. Actually, we are now working on the implementation of the *HYLO* system, a program calculator, for the purpose of calculating efficient functional programs. It should have at least the following characteristics.

- *HYLO* treats functional programs as first class objects and calculate efficient ones via transformation in calculational way. It keeps all advantages of the transformation in calculational way, such as termination property.

- Unlike ADL (Algebraic Design Language) system [KL94] which require users to describe computations in terms of catamorphisms (over inductive data types) and/or anamorphisms (over co-inductive data types), *HYLO* relieves programmers from these restrictions. Instead, it employs an algorithm automatically transforming almost all computations of recursive definitions over data types into hylomorphisms [HIT96d].

- *HYLO* adopts hylomorphisms as its basic recursive forms which are more general than those in the already existed systems. It should automatically perform the fusion transformation and the tupling transformation based on our algorithms which have been or will be newly developed for hylomorphisms [SF93, TM95, HIT96d, HIT96c].
- The functional programs that *HYLO* manipulates can be accepted by the Haskell (core) language [Jon96]. The merit of doing so is that the verification of the improvement of functional programs after transformation can be done easily by running them on Haskell system, and that *HYLO* is hoped to be embedded in the practical Haskell system.
- Although the interpreters of Charity system and ADL system have been reported, as far as we are aware, no literature has been published on how to calculate efficient programs with these systems. This is another reason why we'd like to implement the *HYLO* system.

Another intersting future work is related to programming methodology. There is no reason that calculational approach should be limited to the design of efficient *functional* programs. It is possible to apply it to the design of efficient *imperative* programs. For example, in order to write an efficient imperative program for a problem, we may write a clear functional program in a compositional style, merge as many compositions as possible by fusion transformation, optimize for each recursive function, and finally map recursive definitions to imperative loops. Here the only remained difficulty is in the last step where we have to find an efficient way for this mapping. There have been some studies being devoted to this work. For example, the programming group in Oxford University is trying to produce industrial-strength programs for specific optimization problems such as 0/1 knapsack problem, by transforming functional programs into efficient C code while preserving the lazy behavior of the original functional program. But this work is just a start, no literature being published as far as we know.

We close this thesis with my love maxim: `0Y3XF|1W!"0YF;F|B;` (Lao Tzu, Tao Te Ching, ch.48, about 600 B.C.), i.e.,

To obtain knowledge, add thinks everyday.
To obtain wisdom, remove things everyday.

Bibliography

[Bac78] J. Backus. Can programming be liberated from the von Newmann style? a functional style and its algebra of programs. *Commun. ACM*, 12(8), 1978.

[Bac89] R. Backhouse. An exploration of the Bird-Meertens formalism. In *STOP Summer School on Constructive Algorithmics, Abeland*, September 1989.

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

[BdM94] R.S. Bird and O. de Moor. Relational program derivation and context-free language recognition. In A.W. Roscoe, editor, *A Classical Mind*, pages 17–35. Prentice Hall, 1994.

[Bir80] R. Bird. Tabulation techniques for recursive programs. *ACM Computing Surveys*, 12(4):403–417, 1980.

[Bir84] R. Bird. The promotion and accumulation strategies in transformational programming. *ACM Transactions on Programming Languages and Systems*, 6(4):487–504, 1984.

[Bir87] R. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, pages 5–42. Springer-Verlag, 1987.

[Bir89] R. Bird. Constructive functional programming. In *STOP Summer School on Constructive Algorithmics, Abeland*, 9 1989.

[BSS91] D. Barnard, J. Schmeiser, and D. Skillicorn. Seriving associative operators for language recognition. In *Bulletin of* EATCS (43), pages 131–139, 1991.

[BW88] R. Bird and P. Waddler. *Introduction to Functional Programming.* Prentice Hall, 1988.

[BW90] M. Barr and C. Wells, editors. *Category Theory for Computing Science.* Prentice Hall, 1990.

[CF92] R. Cockett and T. Fukushima. About Charity. Technical report, Dept. of Computer Science, University of Calgary, 1992.

[Chi92] W. Chin. Safe fusion of functional expressions. In *Proc. Conference on Lisp and Functional Programming*, San Francisco, California, June 1992.

[Chi93] W. Chin. Towards an automated tupling strategy. In *Proc. Conference on Partial Evaluation and Program Manipulation*, pages 119–132, Copenhagen, June 1993. ACM Press.

[Chi95] W. Chin. Fusion and tupling transformations: Synergies and conflits. In *Proc. Fuji International Workshop on Functional and Logic Programming*, pages 106–125, Susono, Japan, July 1995. World Scientific Publisher.

[Coh83] N.H. Cohen. Eliminating redundant recursive calls. *ACM Transaction on Programming Languages and Systems*, 5(3):265–299, July 1983.

[Col93] M. Cole. Parallel programming, list homomorphisms and the maximum segment sum problems. Report CSR-25-93, Department of Computing Science, The University of Edinburgh, May 1993.

[CS92] W. Cai and D.B. Skillicorn. Calculating recurrences using the Bird-Meertens Formalism. Technical report, Department of Computing and Information Science, Queen's University, 1992.

[Dar81] J. Darlington. An experimental program transformation system. *Artificial Intelligence*, 16:1–46, 1981.

[dB89] P.J. de Bruin. Naturalness of polymorphism. Technical report, Department of Mathematics and Computing Science, University of Groningen, Groningen, The Netherlands, 1989.

[dM92] O. de Moor. *Categories, relations and dynamic programming.* Ph.D thesis, Programming research group, Oxford Univ., 1992. Technical Monograph PRG-98.

[Fea87] M.S. Feather. A survey and classification of some program transformation techniques. In *TC2 IFIP Working Conference on Program Specification and Transformation*, pages 165–195, Bad Tolz (Germany), 1987. North Holland.

[FJMM90] M. Fokkinga, J. Jeuring, L. Meertens, and E. Meijer. A translation from attribute grammers to catamorphisms. *Squiggolist*, pages 1–6, November 1990.

[Fok92a] M. Fokkinga. A gentle introduction to category theory — the calculational approach —. Technical Report Lecture Notes, Dept. INF, University of Twente, The Netherlands, September 1992.

[Fok92b] M. Fokkinga. *Law and Order in Algorithmics.* Ph.D thesis, Dept. INF, University of Twente, The Netherlands, 1992.

[Fok94] M. Fokkinga. Monadic maps and folds for arbitrary datatypes. Technical report, Dept. INF, University of Twente, The Netherlands, June 1994.

[FST94] L. Fegaras, T. Sheard, and T.Zhou. Improving programs which recurse over multiple inductive structures. Technical Report 94-005, Dept. of Computer Science and Engineering, Oregon Graduate Institution of Science and Technology, 1994.

[FSZ94] L. Fegaras, T. Sheard, and T. Zhou. Improving programs which recurse over multiple inductive structures. In *Proc. PEPM'94*, June 1994.

[Gib92] J. Gibbons. Upwards and downwards accumulations on trees. In *Mathematics of Program Construction* (LNCS 669), pages 122–138. Springer-Verlag, 1992.

[Gib94] J. Gibbons. The third homomorphism theorem. Technical report, University of Auckland, 1994.

[GLJ93] A. Gill, J. Launchbury, and S.P. Jones. A short cut to deforestation. In *Proc. Conference on Functional Programming Languages and Computer Architecture*, pages 223–232, Copenhagen, June 1993.

[Gor95] S. Gorlatch. Constructing list homomorphisms. Technical Report MIP-9512, Fakultät für Mathematik und Informatik, Universität Passau, August 1995.

[Hag87] T. Hagino. *Category Theoretic Approach to Data Types.* Ph.D thesis, University of Edinburgh, 1987.

[Hen80] P. Henderson. *Functional Programming : Application and Implementation.* Prentice Hall International, 1980.

[HI95] Z. Hu and H. Iwasaki. Promotional transformation of monadic programs. In *Proc. Fuji International Workshop on Functional and Logic Programming*, pages 196–210, Susono, Japan, July 1995. World Scientific.

[HIT94a] Z. Hu, H. Iwasaki, and M. Takeichi. Catamorphism-based transformation of functional programs. Technical Report METR 94–06, Faculty of Engineering, University of Tokyo, June 1994.

[HIT94b] Z. Hu, H. Iwasaki, and M. Takeichi. Promotion strategies for parallelizing tree algorithms. In *11st Conf. Proc. Jpn Soc. for Software Sci. and Technical (JSSST '94)*, pages 421–424, Osaka, Japan, November 1994.

[HIT95a] Z. Hu, H. Iwasaki, and M. Takeichi. Making recursions manipulable by constructing medio-types. Technical Report METR 95–04, Faculty of Engineering, University of Tokyo, 1995.

[HIT95b] Z. Hu, H. Iwasaki, and M. Takeichi. Promotional transformation of monadic programs. Technical Report METR 95–05, Faculty of Engineering, University of Tokyo, June 1995.

[HIT96a] Z. Hu, H. Iwasaki, and M. Takeichi. Caculating accumulations. Technical Report METR 96–03, Faculty of Engineering, University of Tokyo, March 1996.

[HIT96b] Z. Hu, H. Iwasaki, and M. Takeichi. Cheap tupling in calculational form (poster abstract). In *8th International Symposium on Programming Languages, Implementations, Logics, and Programs*, Aachen, Germany, September 1996. Springer-Verlag Lecture Notes in Computer Science.

[HIT96c] Z. Hu, H. Iwasaki, and M. Takeichi. Construction of list homomorphisms via tupling and fusion. In *21st International Symposium on Mathematical Foundation of Computer Science, LNCS 1113*, pages 407–418, Cracow, September 1996. Springer-Verlag.

[HIT96d] Z. Hu, H. Iwasaki, and M. Takeichi. Deriving structural hylomorphisms from recursive definitions. In *ACM SIGPLAN International Conference on Functional Programming*, pages 73–82, Philadelphia, PA, May 1996. ACM Press.

[HIT96e] Z. Hu, H. Iwasaki, and M. Takeichi. Formal derivation of parallel program for 2-dimensional maximum segment sum problem. In *Annual European Conference on Parallel Processing, LNCS 1123*, pages 553–562, LIP, ENS Lyon, France, August 1996. Springer-Verlag.

[Hug85] J. Hughes. Lazy memo-functions. In *Proc. Conference on Functional Programming Languages and Computer Architecture* (LNCS 201), pages 129–149, Nancy, France, September 1985. Springer-Verlag, Berlin.

[Hug86] R. J. M. Hughes. A novel representation of lists and its application to the function reverse. *Information Processing Letters*, 22(3):141–144, March 1986.

[Hug89] R. J. M. Hughes. Why functional programming matters. *The Computer Journal*, 32(2):98–107, April 1989.

[HW90] P. Hudak and P. Wadler. Report on the programming language haskell. Report yaleu/dcs/rr-777, Department of Computer Science, Yale University, April 1990.

[JD93] M.P. Jones and L. Duponcheel. Composing monads. Report yaleu/dcs/rr-1004, Department of Computer Science, Yale University, December 1993.

[Jeu93] J. Jeuring. *Theories for Algorithm Calculation.* Ph.D thesis, Faculty of Science, Utrecht University, 1993.

[JL92] S.L. Peyton Jones and D.R. Lester. *Implementing functional languages: a tutorial.* Prentice Hall, 1992.

[Jon88] S.L. Peyton Jones. *The implementation of functional programming languages.* Prentice-Hall, 1988.

[Jon91] Mark P. Jones. Introduction to Gofer. Technical Report Technical draft, PRG, Oxford University, 1991.

[Jon96] S. P. Jones. Compiling haskell by program transformation: A report from the trenches. In *Proc. ESOP.* Springer Verlag LNCS, 1996.

[KL94] R.B. Kieburtz and J. Lewis. Algebraic design language. Technical Report 94-002, Dept. of Computer Science and Engineering, Oregon Graduate Institution of Science and Technology, 1994.

[KW93] D.J. King and P. Wadler. Combining monad. In *Proc. Glasgow Workshop on Functional Programming.* Springer-Verlag, 1993.

[LHJ95] S. Liang, P. Hudak, and M.P. Jones. Monad transformers and modular interpreters. In *Proc. ACM symposium on principles of programming languages*, pages 333–343, San Francisco, 1995.

[LS95] J. Launchbury and T. Sheard. Warm fusion: Deriving build-catas from recursive definitions. In *Proc. Conference on Functional Programming Languages and Computer Architecture*, pages 314–323, La Jolla, California, June 1995.

[Mal90] G. Malcolm. Data structures and program transformation. *Science of Computer Programming*, (14):255–279, August 1990.

[Mei92] E. Meijer. *Calculating Compilers.* PhD thesis, University of Nijmegen, Toernooiveld, Nijmegen, The Netherlands, 1992.

[MFP91] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Proc. Conference on Functional Programming Languages and Computer Architecture* (LNCS 523), pages 124–144, Cambridge, Massachuetts, August 1991.

[MH95] E. Meijer and G. Hutton. Bananas in space: Exteding fold and unfold to exponential types. In *Proc. Conference on Functional Programming Languages and Computer Architecture*, pages 324–333, La Jolla, California, June 1995.

[Mic68] D. Michie. Memo functions and machine learning. *Nature*, 218:19–22, 1968.

[MJ95] E. Meijer and J. Jeuring. Merging monads and folds for functional programming. In *Proc. 1st International Springschool on Advanced Functional Progamming Techniques* (LNCS 925), May 1995.

[Mog90] E. Moggi. An abstract view of programming languages. Technical Report ECS–LSCS–90–113, LFCS, University of Edinburgh, Edingurgh, Scotland, 1990.

[MTH90] R. Milner, M. Tofte, and R. Harper, editors. *The Definition of Standard ML*. MIT, 1990.

[PP93] A. Pettorossi and M. Proietti. Rules and strategies for program transformation. In *IFIP TC2/WG2.1 State-of-the-Art Report*, pages 263–303. LNCS 755, 1993.

[PS87] A. Pettorossi and A. Skowron. Higher-order generalization in program derivation. In *Conf. on Theory and Practice of Software Development*, pages 182–196, Pisa, Italy, 1987. Springer Verlag (LNCS 250).

[SF93] T. Sheard and L. Fegaras. A fold for all seasons. In *Proc. Conference on Functional Programming Languages and Computer Architecture*, pages 233–242, Copenhagen, June 1993.

[SF94] T. Sheard and L. Fegaras. Optimizing algebraic programs. Technical Report Technical Report 94-004, Dept. of Computer Science and Engineering, Oregon Graduate Institution of Science and Technology, 1994.

[Tak87] M. Takeichi. Partial parametrization eliminates multiple traversals of data structures. *Acta Informatica*, 24:57–77, 1987.

[TM95] A. Takano and E. Meijer. Shortcut deforestation in calculational form. In *Proc. Conference on Functional Programming Languages and Computer Architecture*, pages 306–313, La Jolla, California, June 1995.

[Tur85] D.A. Turner. Miranda: A non-strict functional langauge with polymorphic types. In Jouannaud J-P, editor, *Proc. Functional Programming Languages and Computer Architecture*, pages 1–16, Nancy, France, September 1985. Springer Verlag (LNCS 201).

[Wad88] P. Wadler. Deforestation: Transforming programs to eliminate trees. In *Proc. ESOP (LNCS 300)*, pages 344–358, 1988.

[Wad89a] P. Wadler. Editorial - lazy functional programming. *The Computer Journal*, 32(2):97, April 1989.

[Wad89b] P. Wadler. Theorems for free. In *Proc. Conference on Functional Programming and Computer Architecture*, pages 347–359, 1989.

[Wad92] P. Wadler. The essence of functional programming. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 1–14, 1992.

[Yam95] N. Yamashita. *Build-in Memoisation Mechanism for Functional Programs.* Master thesis, Dept. of Information Engineering, University of Tokyo, 1995.

Publication Lists (1996–1994)

Refereed Papers

1. Z. Hu, H. Iwasaki, and M. Takeichi. Construction of list homomorphisms via tupling and fusion. In *21st International Symposium on Mathematical Foundation of Computer Science* (MFCS'96), LNCS 1113, pages 407–418. Cracow, September 1996. Springer-Verlag.

2. Z. Hu, H. Iwasaki, and M. Takeichi. Cheap tupling in calculational form (Poster Abstract). In *8th International Symposium on Programming Languages, Implementations, Logics, and Programs*, LNCS. Aachen (Gernamy), September 1996. Springer-Verlag.

3. Z. Hu, H. Iwasaki, and M. Takeichi. Formal derivation of parallel program for 2-dimensional maximum segment sum problem. In *Annual European Conference in Parallel Processing* (Euro-Par'96), LNCS 1123, pages 553–562. LIP, ENS Lyon, France, August 1996. Springer-Verlag.

4. Z. Hu, H. Iwasaki, and M. Takeichi. Deriving structural hylomorphisms from recursive definitions. In *ACM SIGPLAN International Conference on Functional Programming* (ICFP'96), pages 73–82, Philadelphia, PA. May 1996. ACM Press.

5. Z. Hu and H. Iwasaki. Promotional transformation of monadic programs. In *Fuji International Workshop on Functional and Logic Programming*, pages 196–210, Susono, Japan, July 1995. World Scientific.

Submitted Papers

1. Z. Hu, H. Iwasaki, and M. Takeichi. Calculating Accumulations, submitted to *New Generation Computing*, July 1995.

Domestic Conference/Workshop Papers

1. Z. Hu, H. Iwasaki, and M. Takeichi. Formal derivation of parallel program for 2-dimensional maximum segment sum problem. In *IPSJ SIG Notes, 96-PRO-6*, pages 73–78, March 1996.

2. H. Iwasaki and Z. Hu. Relationship between two approaches to the promotional transformation of functional programs. In *IPSJ SIG Notes, 96-PRO-6*, pages 79–84, March 1996.

3. Z. Hu, H. Iwasaki, and M. Takeichi. Deriving efficient functional programs by constructing medio-types. In *Proc. Workshop on Functional Programming (JSSST-FP '94)*, pages 17–32, Kyoto, Japan, November 1994.

4. Z. Hu, H. Iwasaki, and M. Takeichi. Promotion strategies for parallelizing tree algorithms. In *11st Conf. Proc. Jpn Soc. for Software Sci. and Technical (JSSST '94)*, pages 421–424, Osaka, Japan, November 1994.

5. Z. Hu, H. Iwasaki, and M. Takeichi. Catamorphism-based transformation of functional programs. 94-PRG-16, IPSJ SIG Note, March 1994.

Technical Reports

1. Z. Hu, H. Iwasaki, and M. Takeichi. Calculating Accumulations, Technical Report METR 96–03, Faculty of Engineering, University of Tokyo, March 1996.

2. Z. Hu, H. Iwasaki, and M. Takeichi. Deriving structural hylomorphisms from recursive definitions. Technical Report METR 95–11, Faculty of Engineering, University of Tokyo, October 1995.

3. Z. Hu, H. Iwasaki, and M. Takeichi. Promotional transformation of monadic programs. Technical Report METR 95–05, Faculty of Engineering, University of Tokyo, June 1995.

4. Z. Hu, H. Iwasaki, and M. Takeichi. Making recursions manipulable by constructing medio-types. Technical Report METR 95–04, Faculty of Engineering, University of Tokyo, June 1995.

5. Z. Hu, H. Iwasaki, and M. Takeichi. Catamorphism-based transformation of functional programs. Technical Report METR 94–06, Faculty of Engineering, University of Tokyo, June 1994.