國立臺灣大學管理學院資訊管理學研究所

碩士論文

Department of Information Management

College of Management

National Taiwan University

Master Thesis

以同態建造平行程式

Constructing Parallel Programs Using Homomorphism

郗昀彥

Yun-Yan Chi

指導教授：穆信成 博士

Advisor: Shin-Cheng Mu, Ph.D.

中華民國 104 年 8 月

August, 2015

以同態建造平行程式

Constructing Parallel Programs Using Homomorphism

本 論 文 係 提 交 國 立 台 灣 大 學

資 訊 管 理 研 究 所

作 為 完 成 碩 士 學 位 所 需 條 件 之 一 部 份

研 究 生：郗 昀 彥　撰

中 華 民 國 一 百 零 四 年 八 月

謝辭

時光匆匆，轉眼也走到這一步了。非常感謝這一路上給予我各種協助的人事物，讓我不致於因為現實的無奈與限制而放棄這條路。

首先最要感謝的是這一路上帶領我的指導教授穆信成老師。老師在很多地方都給予了我很多協助與自由，尤其是我因為身體因素時常不能如期完成規劃中的事情時。老師做研究的堅持與學術素養的深厚也給我有很多啟發與激勵。另外，也很感謝我的口試委員們，王柏堯、莊庭瑞和陳恭老師給予我很多討論，讓我得以完善我的論文內容以及接下來的研究方向。台大資管的各任課老師讓我學會了很多資工出身的學生不太容易注意到的議題和知識。此外，雖然說我們實驗室沒有很多成員，但是也很感謝向上和書泓在我研究的過程中給予了我一些建議與討論。此外，真的非常感謝在這些日子中總是可以在關鍵的時候給予我協助的健欣，雖然說他自己有工作要做，但是他還是安排時間幫助我，不管是研究或是私人領域他都惠我良多，甚至陪我演練過整份口試講稿。另外，我也很感謝少娟幫我處理了很多中研院相關的庶務。最後，當然還是要感謝家人們。這些時間身體好好壞壞的，也多虧了家人的陪伴和諒解使得我可以放心繼續完成學業。

其次我要特別感謝的是家琦。她不但這些年在研究相關的研究上協助我啟發我，甚至連學業上也給了我不少協助。但是最重要的是在我身體健康不穩定甚至是動手術的時候她都不時地給我很多安慰和鼓勵。沒有她我是絕對不能堅持到今天這一步的。

郗昀彥 謹識

于台灣大學資訊管理研究所

民國一百零四年八月

論文摘要

學生：郝昀彥　　　　　　　　　　　　　　　　　　　2015 年 8 月

指導教授：穆信成

以同態建造平行程式

平行程式在現今的世界中佔有非常重要的份量。但是，如何開發平行程式卻不是件簡單的事情。傳統上，為了開發平行程式，程式設計師習慣也必須將現有的序列化程式重新改寫成平行的版本。但是，這樣做卻需要花費額外的成本與時間。自然地，程式設計師會想要有一套自動化的方法可以讓他們不必為了平行化而重新實作同一套演算法。因此，大量的研究試圖去開發出有效的方法，以自動化地從現有的序列化程式開始建構或合成出平行程式。其中一種可行的方法就是採用程式推導(program derivation)來做為平行化程式的技術基礎。程式推導允許程式設計師從程式的規格，或是現有的程式定義與性質，來推導並建構出滿足目標需求的程式定義。

本研究中，我們使用同態來描述平行程式的性質並且專注在串列處理相關的程式。我們以兩種不同的特化實例來加以定義：串列同態(list-homomorphism)，用以描述處理平行串列歸納(list-reducing)的程式；串列反同態(list-unhomomorphism)，用以描述處理平行串列生成(list-generating)的程式。第三串列同態定理(third list-homomorphism theorem)告訴我們：一個程式如果具有雙摺疊性(bi-foldability)，則必然存在一個運算子使得該程式可以被寫成一個串列同態。經由利用第三串列同態定理，我們推廣並總結出兩種可以由序列化串列歸納程式推導出其平行定義的方法。第一種允許我們將雙摺疊性之證明一般化成串列同態之證明，而其中則包含了目標程式之串列同態定義。第二種方法允許我們經由語法操作而產生出目標串列同態之定義，一旦目標程式之右弱反函數(right weak inverse)可以被設計並決定。另一方面，對於串列反同態，我們推廣出一個第三串列同態定理之對偶定理(dual theorem)。基於此定理，如果某兩個串列反同態之基本性質之前提條件皆可以被滿足的話，我們可以根據序列化串列生成程式之定義建構出一個串列反同態之定義。

關鍵字：平行程式，函數式程式，串列同態，第三串列同態定理，程式推導

THESIS ABSTRACT

GRADUATE INSTITUTE OF INFORMATION MANAGEMENT
NATIONAL TAIWAN UNIVERSITY

Student: Chi, Yun-Yan
Advisor: Mu, Shin-Cheng

Month/Year: August, 2015

Constructing Parallel Programs Using Homomorphism

Parallel programming plays an very important role in nowadays world. To develop a parallel program, however, is not a simple task. Traditionally programmers are used to and have to rewrite a sequential program to its parallel version. This, however, takes lots of extra efforts and times. It natural that programmers want to have an automatic method for saving themselves from re-implementing the same algorithm twice. Therefore plenty of previous works have tried to develop methods for constructing and synthesising parallel programs from existing sequential programs. One of potential ways is using program derivation which allows us to construct a program definition from its specification, or, in this case, from existing definitions and properties.

In this thesis, we use *homomorphism* as a model of parallel programs and focus on programs that handle the well-known linear data structure, *list*. We realise homomorphism as two specialisations: the *list-homomorphism*, which works as properties and is used for modelling parallel list-reducing programs; and the *list-unhomomorphism*, for parallel list-generating programs. By taking advantage of the *third list-homomorphism theorem* we introduce two methods to derive parallel list-reducing programs from its sequential definitions. The first method allows us to transform a proof of bi-foldability to a proof of existing of list-homomorphism where, obviously, contains the definition of list-homomorphism itself. The second one, provides us a syntactical approach to construct a list-homomorphism once an right weak inverse of the program we want to parallelise can be designed and picked. For list-unhomomorphism, on the other hand, we develop a dual theorem of the third list-homomorphism theorem so that list-generating programs can be constructed from its sequential definitions once the sufficient conditions of two essential properties of list-unhomomorphism can be satisfied.

Keywords: Parallel Program, Functional Program, List-Homomorphism, Third List-Homomorphism Theorem, Program Derivation

Contents

List of Figures

Chapter 1

Introduction

Since the rising of multi-core processing, cloud computing and machine learning, program parallelisation has became an important and necessary technique. There are at least two scenarios where we need the benefits of parallelisation. Firstly programmers nowadays confront a world with a huge amount of data. A typical sequential algorithm is no longer satisfied our needs on computation since it is usually inefficient when input data is too large. Secondly multi-core computers and distributed systems become more and more common. Developing a program to execute on a multi-core or distributed system will be an essential skill for a programmer.

The well-known divide-and-conquer paradigm provides many benefits and one of the most important advantage is the potential for parallelisation. The main idea of divide-and-conquer is to split a problem into several sub-problems, and then recursively solve these sub-problems. For example, function h, which takes a list zs as input and returns b as result, could be written as a divide-and-conquer algorithm if we can somehow split zs into two sub-lists, xs and ys such that the result b could still be obtained by combining $h\ xs$ and $h\ ys$.

This thesis reveals a new parallelisation framework based on *homomorphism* [3] which matches the divide-and-conquer paradigm [9]. Moreover, our approach provides not only a way to construct parallel program from specification but also the proof of correctness of parallelization.

1.1 Correctness

Plenty of programmers are used to heuristically design and develop program specifications in their own mind before they actually implement an algorithm. Once a program is implemented, programmers will perform extensive testing to verify whether the program's behavior matches its specification. After a program passes all of its testing, programmers claim that their program would not go wrong. This approach *do* reveal many bugs if the specification was designed correctly. However, just as Edsger W. Dijkstra said in 1969: "Program testing can be used to show the presence of bugs, but never to show their absence!" There always exists some unexpected cases because testing can not be exhaustive.

An alternative approach is the *formal program construction* [4, 12], that guarantees correctness by modeling a specification as a mathematical structure and then constructing an implementation from it by performing mathematical manipulations. The whole construction process must follow mathematical principles and properties to ensure the consistency between a specification and its implementation. In other words, formal program construction provides not only the way to "calculate" a program specification to its efficient implementation but also the proof of correctness.

In this thesis we would like to discuss a methodology for constructing parallel program and providing its correctness proof as well. To do so, we will use (purely) functional programming languages [2, 11] to represent mathematical structures and therefore one can apply program derivation.

1.2 Parallelisation

To understand the main idea of the method we propose, considering the well-known algorithm paradigm – divide-and-conquer, a technique to solve problems by splitting a problem into several smaller sub-problems. For example, *mergesort* is a typical parallel algorithm in divide-and-conquer paradigm,

$$
\begin{aligned}
&\mathit{mergesort}\ [\,] &&= [\,]\\
&\mathit{mergesort}\ [a] &&= [a]\\
&\mathit{mergesort}\ \mathit{as} &&= \mathit{xs}\ \text{`}\mathit{merge}\text{`}\ \mathit{ys}\ \mathbf{where}\\
& && \qquad n = (\mathit{length}\ \mathit{as})\ \mathit{div}\ 2\\
& && \qquad \mathit{xs} = \mathit{mergesort}\ (\mathit{take}\ n\ \mathit{as})\\
& && \qquad \mathit{ys} = \mathit{mergesort}\ (\mathit{drop}\ n\ \mathit{as})\ ,
\end{aligned}
$$

where *merge* merges two sorted lists into one.

Although the divide-and-conquer paradigm is powerful framework for solving complicated problems in parallel, there is no explicit explanations of how to precisely develop or implement a parallel algorithm. In fact, development of parallel algorithms is often an ad-hoc process. One usually has to create parallel algorithms case by case. That developing process could be painful yet still leads us to a wrong end.

The way to go, in stead of directly writing a parallel program with human mind, we model the divide-and-conquer paradigm with mathematical structures, which provide several helpful mathematical theorems and properties. Then we develop syntactic construction methods that allow us to calculate (or derive) a definition of parallel program from its specification by using arithmetical and algebraic methods only. In other words, one may have an efficient parallel program by using purely mathematical manipulations rather than relying on problem oriented understanding.

In this thesis, we pick the well-known *homomorphism* as that mathematical structure. Taking lists as an example, function h is a *list-homomorphism* if for any lists xs and ys there exists an $(\oplus)$ such that

$$h\ (\mathit{xs} \mathbin{+\!\!+} \mathit{ys}) = (h\ \mathit{xs}) \oplus (h\ \mathit{ys}) \tag{1.1}$$

holds. In fact, in this thesis we focus only on lists for two reasons. Firstly, as we mentioned, one of the biggest challenge for nowadays programmers is gathering information from considerable data. Which means, it will be very common to develop a parallel program for generating or consuming lists. Secondly, from the viewpoint of recursive data structure, a list-based method is relatively easier to be developed. Besides, a list-based method also has potential to be generalised to another method which can handle some more complicated recursive data structures.

Back to the list-homomorphism, the problem is how could we know there exists an $(\oplus)$ such that h could be written in terms of a list homomorphism? If so, how could we construct that $(\oplus)$? One of the possible ways is to take the advantage of the *third list homomorphism theorem* [8], which says that there must exist an operator $(\oplus)$ such that the equation above holds if h can be evaluated *rightwards* and *leftwards*. The third list-homomorphism theorem provides the existence of $(\oplus)$, yet without its definition. Therefore, the challenge we are facing is, how does one develop or construct a definiton of $(\oplus)$? The answer is to be revealed in this thesis.

1.3 Background

In the late 90's, much effort has been made to find a systemic way of constructing a parallel program in terms of homomorphism. Gibbons [8] formalised and semantically proved the third list-homomorphism theorem followed by showing how to improve a sorting algorithm. This shows an opportunity of constructing a homomorphism with help of the third list–homomorphism theorem. Another attempt of constructing homomorphism was taking advantage of *almost homomorphisms*. An almost homomorphism is a non-homomorphic function which can be written either leftwards (i.e. as a *foldr*) or rightwards (i.e. as a *foldl*), after being tupled with several functions, each of which provides necessary information such that the almost homomorphism itself can be written as a homomorphism. Hu et al. [10] proposed a way to construct a list-homomorphism by fusion with *almost homomorphisms*. Gorlatch [9] proposed an extraction method, called *CS-method*, to extract a definition of homomorphism by generalising two sequential representations, namely *foldr* and *foldl*. Following the result of previous work, Geser and Gorlatch [7] applied term rewriting techniques to systematically extract homomorphism from a pair of sequential representations.

After then, the idea of constructing homomorphism with the third list-homomorphism theorem has received more attention. Morita et al. [18] proposed to automatically construct an $(\oplus)$ by picking some right weak inverse h^{-1} and refining the equation $(\oplus) = h\ (h^{-1}\ s +\!\!+ h^{-1}\ t)$. On the other hand, the third list-homomorphsim theorem, after minor modifications, was also shown to be workable on other polynomial structures. Taking binary tree as an example, those two sequential representations are no longer leftwards and rightwards. Instead, a function can be written as a tree-homomorphism if it can be computed downwards and upwards. Based on that concept, Morihata et al. [17] developed a method for constructing parallel programs on trees. To discover more details of the third list-homomorphism theorem, Mu and Morihata [19] applied relational formalism to formalise the third list-homomorphism theorem and, with the concept of right weak inverse, developed several necessary and sufficient conditions. With those results, they also developed a dual theorem discussing functions that generate lists, rather than consuming lists.

As mentioned above, the right-weak-inverse approach has been take by many previous works. However, coming up with a proper h^{-1} is not an easy task. Chi and Mu [5] observed that the steps of proving h can be written as *foldr* and *foldl*, and the steps of proving h is a list-homomorphism are very similar. From there, they developed a syntactical method to construct a $(\oplus)$ from the proof of h can be *foldr* and *foldl*.

Morihata [16] applied shortcut deforestation to prove a meta-theorem of the third list-homomorphism theorem. Additionally, following the formalization in [19], namely using relations to formalise and prove, the result can generalised to anamorphisms (unfolds) and hylomorhpisms.

Based on the previous works mentioned above, we have more and more understanding of list-homomorphisms. Some people may start to wonder the practicability of list-homomorphism. Liu, Hu and Matsuzaki [15] proposed and implemented a homomorphism-based framework for systematic parallel programming with the well-known MapReduce. That framework can derive an efficient parallel algorithm as a list-homomorphism if it is already known as satisfiying the requirements of the third list-homomorphism theorem. Since the requirements are still required to be satisfied, Emoto, Fischer and Hu [6] proposed a calculation-based framework, named generate-test-and-aggregate (GTA for short), by integrating the generate-and-test programing paradigm and semiring fusion theorem. That GTA framework allows programmers to systematically synthesize efficient MapReduce programs. To put GTA to practical use, Liu, Emoto and Hu [14] implemented this programming model to work with Scala, Spark or Hadoop.

Excepting to apply list-homomorphism on parallel programming models such as MapReduce, some researchs also tried to design a notion of homomorphism to formalise structured models of parallelism. Legaux et al. [13] proposed and implemented the BSP homomorphsim dedicated to bulk synchronous parallelism.

1.4 Outline

The necessary theoretical concepts and technical programming knowledge will be introduced in Chapter 2 together with some necessary mathematical foundation. We then will show two possible ways for developing a parallel program from it specification if those corresponding requirements can be satisfied in Chapter 3. For each method, we will also demonstrate how to come up with a parallel program with examples. After introducing the way to build parallel program which folds a list into a value, we will also show the way to construct a parallel program for expending a list from a value in Chapter 4. In the end of this chapter, three examples will be given. Finally we will give a brief summary and discuss some interesting issues for future works in Chapter 5.

Chapter 2

Preliminaries

In this chapter, we will give mathematical background and a theoretical setting in Section 2.1. In Section 2.2 we will explain the way of formal program construction. Finally, we will also introduce those necessary programming technology in Section 2.3.

2.1 Mathematical Background

We assume a *set-theoretical* model for functional programming. All functions in this thesis are *total*, every element in domain is mapped to some element in range, and *simple*, every element in domain is mapped to only one element in range.

A function $(\circ)$ is called *functional composition* if, given functions $f :: A \to B$ and $g :: B \to C$, $g \circ f$ returns a function with type as $A \to C$ such that for all $x :: A$, we have

$$(g \circ f)\ x = g\ (f\ x)\ .$$

Given a function $f :: A \to B$, a function $f^{-1} :: B \to A$ is called *right weak inverse*, if for all y in the range of f, we have $f(f^{-1}\ y) = y$. In set-theoretical model, a right weak inverse must exist but may not be unique. In this thesis, we use an equivalent equation,

$$f = f \circ f^{-1} \circ f\ ,$$

instead of the original one.

In set-theoretical model, a relation $R :: A \to B$ is a set of pair, where, for all

$x \in A \wedge y \in B$, we have $(x, y) \in R$. In this thesis, however, we will deal with a situation where both of domain and range are a set of pair already. Therefore, we want an alternative representation to make it more friendly to human. Let relation R be defined as above, we introduce the operator $(\leftsquigarrow)$:

$$y \leftsquigarrow R\ x \equiv (x, y) \in R\,.$$

Additionally, because we will show some diagram in this thesis, based on $(\leftsquigarrow)$, we also define that the following snaky arrow,

$$\begin{tikzcd} x \arrow[r, squiggly, "R"] & y \end{tikzcd}\ ,$$

which will be used in Figure 3.1 and Figure 4.1, to represent $y \leftsquigarrow R\ x$ in diagram.

For relation, one can also define the *relational composition* by

$$z \leftsquigarrow (S \circ R)\ x\ \equiv\ \exists y\ .\ z \leftsquigarrow S\ y \wedge y \leftsquigarrow R\ x\,.$$

As one could notice, we use the very same notation to indicate both compositions since the functional composition is actually a special case of relational composition.

Given a relation $R :: A \to B$, its *converse* $R^{\circ} :: B \to A$ is defined by

$$x \leftsquigarrow R^{\circ}\ y \equiv y \leftsquigarrow R\ x\,.$$

Plus, we have $R \circ R^{\circ} \circ R = R$ since R and R° are total.

2.2 Program Derivation

As we mentioned in Section 1.1, instead of developing program then verifying it against its specification, we use *formal program construction* as our developing method in this thesis. In formal program construction, one models specifications by formal representations, usually mathematical structures like algebra [4] or logic [1], then a program (or a function definition) can be *constructed* from that specification by applying some mathematical properties somehow. In general, there are several ways to construct a program from its specification.

Among the many approaches to construct a program, we will use a framework named *program derivation* as our developing approach. As its name implies, constructing a program with program derivation means that, given a formal spesification, a program can be derived from it by applying mathematical operations and properties, which guarantee that each step in derivation will be mathematical correct. More precisely, in this thesis we will use algebraic program derivation – we model our specification with algebraic structures and derive a program from it by applying algebraic manipulations.

Usually, a specification is written by a sequence of composition of functions. But, here is the thing, functions we talk about here are all deterministic because, for all input, a function returns only one corresponding result. On the other hand, a relation can be used to present a non-deterministic program since a relation relates any element in its domain into more then one elements in its range. Therefore, let's introduce the *relational program derivation* that allows us to write a specification in terms of relations and then derive, or say, refine, it into some properties or requirements that one can construct a deterministic program, namely a function definition, from.

2.3 Functional Programming

Functional programming is a very powerful programming paradigm. The most important feature of functional programming is that it provides capability to capture or describe mathematical properties of programs. In functional world, it is easier for programmers to focus on the nature of the problem we want to solve by computer rather than some implement issues. That is, with functional languages and functional programming techniques, less efforts in issues irrelevant to the problem will be required; hence programmers can build or even realise complex programs well. Please notice that, since we assume a set-theoretical model and all functions are total and simple, when we say functional programming, we actually mean *total* functional programming.

2.3.1 Programming with Functions

Programming with a functional language means that programmer will use functions as first-class objects. To manipulate functions as atoms, some techniques are required.

Functional composition is the same thing as in mathematics and provides us a way to sequentially combine two functions. However, sometimes one may need to combine two functions parallelly. Therefore, given function $f :: A \to B$ and $g :: A \to C$, the function $(f, g) :: A \to (B, C)$ is defined by

$$(f, g)\ x \equiv (f\ x, g\ x)$$

for all $x :: A$. This composition requires that the domain of both functions must be the same. For the case two different domains, we can define another kind of parallel composition. Given $h :: A \to B$ and $k :: C \to D$, the function $(h \times k) :: (A, C) \to (B, D)$ is defined by

$$(h \times k)\ (x, y) \equiv (h\ x, k\ y)$$

for all $x :: A$ and $y :: C$. Interestingly, one can distribute $(\circ)$ over $(\times)$. That is, we have

$$(f \times g) \circ (h \times k) = (f \circ h \times g \circ k)\ ,$$

which is a very useful property in program derivation. Those compositions let us construct a complex program by combining several different functions, or, conversely we can divide the given program into several subproblems each of which can be solved by a simple function.

2.3.2 Linear Structures

One of the well-known essential data structure in programming is the linear data structure – *list*. We use [ ] and, for example, $[x_0, x_1, x_2, ...]$, for presenting such abstract list. Ideally a list should be nothing but a sequence of elements. However when one actually try to build a list, it will be a sequence of *connection* and suddenly the ordering of connections does matter. For example, list $[1, 2, 3]$ may be built in several ways: one can start by "connecting" 2 to the right-hand side of 1 followed by "connecting" 3 to their right end; or, start with 2 and "connect" 1 to its left-hand-side followed by "connecting" 3 to the right-hand-side of $[1,2]$. Even though there are several different ways to build a list, in this thesis we only require and hence introduce the most common two types of list, *cons-list* and *snoc-list*.

A list is called *cons-list* if it is constructed leftwards. The structure cons-list can be defined by:

$$\begin{aligned} \textbf{data } [a]_c &= [\,]_c \\ &\mid a \prec [a]_c. \end{aligned}$$

On the other hands, a *snoc-list* is constructed rightwards as:

$$\begin{aligned} \textbf{data } [a]_s &= [\,]_s \\ &\mid [a]_s \succ a. \end{aligned}$$

Sometimes it does not matter which kinds of list we are using since cons-list and snoc-list are isomorphic – there must exists two functions such that we can convert a cons-list from/to a snoc-list. In fact, for some list-related problems, it could be solved easily with both of cons-list or snoc-list. For example, it does not matter that the summation of a list is processed leftwards or rightwards. But most of times it would be easier to solve a problem only with either cons-list or snoc-list.

2.3.3 Folds and Unfolds

In this section we will introduce several fundamental concepts such as *folds*, *unfolds* and their *fusion theorems*.

Folds

A function is a *fold* if it "folds" a list into a value. Like the ways one can construct a list, there are planty of possibilities to fold a list. The simplest and most typical way is folding a list along its constructing direction. That is, since there are two implementations of list in practice, we may have two corresponding folding methods. In the following paragraphs, we will introduce the distinguished *foldr*, corresponding to cons-list, and the *foldl*, corresponding to snoc-list.

Given $e_c ::\ b$ and $(\triangleleft) :: a \rightarrow b \rightarrow b$, the following equations have a unique solution for $f :: [a] \rightarrow b$ which is denoted $\mathit{foldr}\ (\triangleleft)\ e_c$:

$$
\begin{aligned}
f\ [\,]_c &= e_c \\
f\ (x \prec xs) &= x \triangleleft (f\ \ xs).
\end{aligned}
$$

Such f takes a list as input and maintains the structure of elements but replaces constructor $[\,]_c$ and $(\prec)$ respectively by e_c and $(\triangleleft)$. For example, $x_0 \prec (x_1 \prec [\,]_c)$ becomes $x_0 \triangleleft (x_1 \triangleleft e_c)$. On the other hand, for snoc-list, the following equations have an unique solution for $f :: [a] \rightarrow b$ which is denoted $\mathit{foldl}\ (\triangleright)\ e_s$:

$$
\begin{aligned}
f\ [\,]_s &= e_s \\
f\ (ys \succ y) &= (f\ \ ys) \triangleright y.
\end{aligned}
$$

where $e_s ::\ b$ and $(\triangleleft) :: b \rightarrow a \rightarrow b$.

As an example of defining folds, to sum up a list of numbers, one can define a function *sum* for cons-list as $\mathit{foldr}\ (+)\ 0$; or, for snoc-list as $\mathit{foldl}\ (+)\ 0$. Both of them are workable, e.g. given a list $[3, 5, 7]$, we have its summation 15 by calculating either

$$
\begin{aligned}
&\mathit{sum}\ (3 \prec (5 \prec (7 \prec [\,]_c))) \\
= {}&\mathit{foldr}\ (+)\ 0\ (3 \prec (5 \prec (7 \prec [\,]_c))) \\
= {}&(3 + (5 + (7 + 0))) = 15,
\end{aligned}
$$

or

$$
\begin{aligned}
&\mathit{sum}\ ((([\,]_s \succ 3) \succ 5) \succ 7) \\
= {}&\mathit{foldl}\ (+)\ 0\ ((([\,]_s \succ 3) \succ 5) \succ 7) \\
= {}&(((0 + 3) + 5) + 7) = 15.
\end{aligned}
$$

Unfortunately, we do not always as lucky as above example, sometimes a function can be defined only in terms of either one of *foldr* and *foldl*. A function f is *bi-foldable* if it can be folded leftwards and rightwards. That is, given a bi-foldable function f, there must exists $(\triangleleft)$, $(\triangleright)$ and e such that

$$f = \mathit{foldr}\ (\triangleleft)\ e = \mathit{foldl}\ (\triangleright)\ e\ .$$

Unfolds

Another well-known process in functional programming is *unfold*. An unfold takes a value, called its *seed*, and expands a list from that seed. Since there are two ways to construct a list, we have two implementations for unfold, namely *unfoldr* and *unfoldl*, as follows,

$$\begin{array}{lcl}
\mathit{unfoldr} & :: & (a \to (b,a)) \to (a \to \mathit{Bool}) \to a \to [b] \\
\mathit{unfoldr}\ g_{\triangleleft}\ p\ s & = & \mathbf{if}\ p\,s\ \mathbf{then}\ [\,]_c\ \mathbf{else}\ x \prec \mathit{unfoldr}\,g_{\triangleleft}\,p\,t \\
 & & \quad \mathbf{where}\ (x,t) = g_{\triangleleft} s\ ; \\
\mathit{unfoldl} & :: & (a \to (a,b)) \to (a \to \mathit{Bool}) \to a \to [b] \\
\mathit{unfoldl}\ g_{\triangleleft}\ p\ s & = & \mathbf{if}\ p\,s\ \mathbf{then}\ [\,]_s\ \mathbf{else}\ \mathit{unfoldl}\ g_{\triangleright}\ p\ t \succ y \\
 & & \quad \mathbf{where}\ (t,y) = g_{\triangleright} s\ ,
\end{array}$$

where functions $g_{\triangleleft}$ and $g_{\triangleright}$ generate a value and a new seed from an old seed, and predicate p specifies the terminal condition of unfolding. An unfold is coinductive and hence could generate list with infinite length. In this thesis, we demand that our *unfoldr* and *unfoldl* generates inductive finite list instead. That it, all successive applications of seed-generating function eventually reaches some seed so that the terminal condition can be satisfied. There are two reasons for this restriction. The first reason will be explained in next paragraph and the other one will be mentioned later in Section 4.1.

Given a list generating function k, it is *bi-unfoldable* if there exists $g_{\triangleleft}$, $g_{\triangleright}$ and predicate p such that

$$k = \mathit{unfoldr}\ g_{\triangleleft}\ p = \mathit{unfoldl}\ g_{\triangleleft}\ p\ .$$

That a function is bi-unfoldable means that any list it returns must have both a left-end and a right-end. In chapter 4, the bi-unfoldability is an essential and widely-used property and therefore it becomes the first reason why we restrict our unfolds to return finite lists.

Fusion Theorems

There are plenty of properties and theorems regarding fold that can help us to construct or develop efficient programs. One of the most useful theorems is the well-known *fold-fusion theorem*, which gives conditions under which a function f can be fused into a *fold*. Technically, one can specify fold-fusion for different list implementations such as cons-list and snoc-list, and thus we have a *foldr*-fusion theorem and a *foldl*-fusion theorem. In this thesis, the reader only needs to understand the *foldr*-fusion theorem, so in this section we will introduce this one only.

Theorem 2.3.1. [*Foldr*-Fusion] Given $f = \mathit{foldr}\ (\triangleleft)\ e$, for a function g there must exists a $(\prec)$ such that

$$g \circ f = \mathit{foldr}\ (\prec)\ (g\ e)$$

if we have

$$g\ (x \triangleleft z) = x \prec (g\ z)\ .$$

It can be easily proved by an induction on the input list. The base case trivially holds. For inductive case, we have,

$$\begin{aligned}
& g(\mathit{foldr}\ (\triangleleft)\ e\ (x < xs)) \\
= \ & \{ \text{ definition of } \mathit{foldr} \ \} \\
& g(x \triangleleft (\mathit{foldr}\ (\triangleleft)\ e\ xs)) \\
= \ & \{ \ g\ (x \triangleleft z) = x \prec (g\ z) \ \} \\
& x \prec (g(\mathit{foldr}\ (\triangleleft)\ e\ xs)) \\
= \ & \{ \text{ induction } \} \\
& x \prec (\mathit{foldr}\ (\prec)\ (g\ e)\ xs) \\
= \ & \{ \text{ definition of } \mathit{foldr}\text{, backwards } \} \\
& \mathit{foldr}\ (\prec)\ (g\ e)\ (x < xs)\ .
\end{aligned}$$

For *unfold* there are also some fusion theorems. In this thesis we will use only the most basic one - *map-unfold*-fusion theorem. Again, for *unfoldr* and *unfoldl* we will have *map-unfoldr*-fusion theorem and *map-unfoldl*-fusion theorem as follows,

Theorem 2.3.2. [*Map-Unfoldr*-Fusion] Given $k = unfoldr\ g_{\triangleleft}\ p$, for a function f there must exists a $h_{\triangleleft}$ such that

$$map\ f \circ k = unfoldr\ h_{\triangleleft}\ p$$

if we have

$$h_{\triangleleft} = (f \times id) \circ g_{\triangleleft}\ .$$

Theorem 2.3.3. [*Map-Unfoldl*-Fusion] Given $k = unfoldl\ g_{\triangleright}\ p$, for a function f there must exists a $h_{\triangleright}$ such that

$$map\ f \circ k = unfoldl\ h_{\triangleright}\ p$$

if we have

$$h_{\triangleright} = (id \times f) \circ g_{\triangleright}\ .$$

The two theorems above have very similar proofs therefore we show only the proof of *map-unfoldr*-fusion theorem, which can be proved by an induction on the applications of seed-generating. The base case, $p\ s$ is true for some s, is trivially holds. The inductive case, for any seed s we reason,

$$\begin{aligned}
& map\ f\ (unfoldr\ g_{\triangleleft}\ p\ s) \\
= & \quad \{ \text{ let } (x, t) = g_{\triangleleft}\ s \ \} \\
& map\ f\ (x \prec (unfoldr\ g_{\triangleleft}\ p\ t)) \\
= & \quad \{ \text{ definition of } map \ \} \\
& f\ x \prec (map\ f\ (unfoldr\ g_{\triangleleft}\ p\ t)) \\
= & \quad \{ \text{ induction } \} \\
& f\ x \prec (unfoldr\ h_{\triangleleft}\ p\ t) \\
= & \quad \{ (f\ x, t) = (f \times id) \circ g_{\triangleleft}\ s = h_{\triangleleft}\ s \ \} \\
& unfoldr\ h_{\triangleleft}\ p\ s\ .
\end{aligned}$$

2.3.4 List-Homomorphisms

A function h is a list-homomorphism if there exists a function $(\oplus)$ such that for any lists xs and ys

$$h\ (xs +\!\!+\ ys) = (h\ xs) \oplus (h\ ys)$$

can be satisfied. List-homomorphism provides great potential of parallelisation: to compute h, one may arbitrarily split the input list into two sub-lists, recursively compute them, and combine the result via $(\oplus)$.

But obviously, not every functions on list can be written as a list-homomorphism. It is important to find the requirements for a function to be a list-homomorphism. The well-known *third list-homomorphism theorem* [8] shows the very requirement we need.

Theorem 2.3.4. [Third List Homomorphism] Given a bi-foldable function h, there must exists an $(\oplus)$ such that h is a list-homomorphism. That is, for all xs and ys there must exists an $(\oplus)$ such that

$$h\ (xs +\!\!+\ ys) = (h\ xs) \oplus (h\ ys)$$

if for some some $(\triangleleft)$, $(\triangleright)$ and e we have

$$h = \mathit{foldr}\ (\triangleleft)\ e = \mathit{foldl}\ (\triangleright)\ e$$

Chapter 3

Constructing a List-Homomorphism

The third list-homomorphism theorem provides an opportunity to construct a parallel program on lists if it could be specified in both *foldr* and *foldl*. A traditional challenge is finding a mechanical method to take advantage of the third list-homomorphism theorem to construct operator $(\oplus)$ for a given function h such that h can be written as a parallel program.

In following sections, we start with developing sufficient conditions for existence of $(\oplus)$ by introducing the *second duality theorem*. Then we will apply some techniques, such as *inverse function* and *generalised fold*, to form essential properties of list-homomorphism, hoping these two different properties can lead us to clearer ways of constructing proper $(\oplus)$.

3.1 Sufficient Conditions for List-Homomorphism

As one may notice, given $h = \mathit{foldr}\ (\triangleleft)\ e = \mathit{foldl}\ (\triangleright)\ e$, the $(\oplus)$ is a generalised result of both of $(\triangleleft)$ and $(\triangleright)$. One effective method to construct a $(\oplus)$ is synthesising a proper $(\oplus)$ from the definitions of $(\triangleleft)$ and $(\triangleright)$ [7, 9]. This attempt, however, may take plenty of efforts, yet generate no useful result. In this section we try to develop sufficient conditions for bi-foldability and hope these conditions can show us some clues to find $(\oplus)$ with less efforts.

Given $(\triangleleft)$, $(\triangleright)$ and e, function $h = \mathit{foldr}\ (\triangleleft)\ e$ is bi-foldable if we can show that $h\ (ys \succ z) = \mathit{foldl}\ (\triangleright)\ e\ (ys \succ z)$, which can be written in point-free style as $h \circ (\succ z) = (\triangleright z) \circ h$. Therefore, to show h is truly bi-foldable, we reason that

$$\begin{aligned}
& h \circ (\succ z) \\
= \quad & \{\ \mathit{foldr}\text{-fusion, since } (\succ z) = \mathit{foldr}\ (\prec)\ [z]\ \} \\
& \mathit{foldr}\ (\triangleleft)\ (h\ [z]) \\
= \quad & \{\ \mathit{foldr}\text{-fusion, backwards}\ \} \\
& (\triangleright z) \circ \mathit{foldr}\ (\triangleleft)\ e \\
= \quad & \{\ h = \mathit{foldr}\ (\triangleleft)\ e\ \} \\
& (\triangleright z) \circ h\ ,
\end{aligned}$$

where two *foldr*-fusions are required. Fusion conditions in the first *foldr*-fusion trivially holds. The second fusion, on the other hand, requires

$$h\ [z] = (\triangleright z)\ e \ \wedge\ (\triangleright z)\ (x \triangleleft y) = x \triangleleft ((\triangleright z)\ y)$$

as its fusion-conditions and thus leads us to the *second duality theorem*[2, Pages 128].

Theorem 3.1.1. [Second Duality Theorem] Given $(\triangleleft)$, $(\triangleright)$ and e, $\mathit{foldr}\ (\triangleleft)\ e = \mathit{foldl}\ (\triangleright)\ e$ for all finite input lists if, for all x, y and z, we have

$$z \triangleleft e = e \triangleright z \ \wedge\ x \triangleleft (y \triangleright z) = (x \triangleleft y) \triangleright z\ . \tag{3.1}$$

Now considering that, given $h = \mathit{foldr}\ (\triangleleft)\ e = \mathit{foldl}\ (\triangleright)\ e$, our goal is to come up with an $(\oplus)$ such that h can be defined as a list-homomorphism. We now try to find out what properties $(\oplus)$ must satisfy and hope those properties can show us the way to construct $(\oplus)$. Starting with the definition of list-homomorphism, $h\ (xs +\!\!+ ys) = h\ xs \oplus h\ ys$, which can also be written point-free as $h \circ (+\!\!+ ys) = (\oplus ys) \circ h$, one could easily have the following derivation,

$$
\begin{aligned}
& h \circ (\mathbin{+\!\!+} ys) \\
= \;& \{ \; \mathit{foldr}\text{-fusion, since } (\mathbin{+\!\!+} ys) = \mathit{foldr}\ (<)\ ys \; \} \\
& \mathit{foldr}\ (\lhd)\ (h\ ys) \\
= \;& \{ \; \mathit{foldr}\text{-fusion, backwards; see below} \; \} \\
& (\oplus\ ys) \circ \mathit{foldr}\ (\lhd)\ e \\
= \;& \{ \; \text{assumption: } h = \mathit{foldr}\ (\lhd)\ e \; \} \\
& (\oplus\ ys) \circ h \,.
\end{aligned}
$$

Again, fusion conditions in the first fusion trivially holds. For the second fusion, the fusion-conditions

$$h\ ys = e \oplus (h\ ys) \;\wedge\; (x \lhd y) \oplus (h\ ys) = x \lhd (y \oplus (h\ ys)) \tag{3.2}$$

must be satisfied. That is, (3.2) is the essential property we desire.

Constructing an $(\oplus)$ So far we know that, given h is bi-foldable for $(\lhd)$, $(\rhd)$ and e, there are several useful facts. Firstly, there must exists a correctness proof for (3.1) which involves the definition of $(\rhd)$. Secondly, by Theorem 2.3.4, there also exists an $(\oplus)$ such that h can be defined as a list-homomorphism. That is, there also exists a correctness proof for (3.2), where we can find a definition of $(\oplus)$ from. Finally, since $(\rhd)$ is a special case of $(\oplus)$, it's not hard to see that the proof (3.2) can be generalised from the proof (3.1). Therefore, the way to go, one can construct a proof of (3.2) by transforming the proof of (3.1), which should be provided for showing bi-foldability anyway, with a candidate definition of $(\oplus)$. More precisely, to construct a correctness proof of (3.2), one can firstly replace each occurrences of the out-most element in the definition of $(\rhd)$ by some *meta-variables* and thus has a candidate definition of $(\oplus)$. Then, copy the steps in the proof of (3.1) and substitute each occurrences of $(\rhd)$ by the meta-variable-filled definition of $(\oplus)$. As result, one will obtain a meta-variable-filled correctness proof of (3.2). Those meta-variables could be refined and determined by verifying other related properties. In the end, a definition of $(\oplus)$ together with its correctness proof can thus be constructed.

3.2 The Steepness of a List

As an example of constructing list-homomorphism, we look at a specific problem. A list is *steep* if each number is larger than the sum of the numbers to its right. For example, [20,10,4,2,1] is steep but not [20,6,4,2,1]. One can easily define *steep* as:

$$\begin{aligned}
&steep && :: [Int] \rightarrow Bool \\
&steep\ [\,] && = True \\
&steep\ (x \prec xs) && = x > sum\ xs \wedge steep\ xs\ .
\end{aligned}$$

Sadly this *steep* is neither a *foldr* nor a *foldl* because it drops all list-related information and returns only a boolean value. But if one generalize *steep* a little bit by making it return more than just a boolean value, it could be written as folds. To be able to do so, we introduce *capacity* that is the maximum number which can be attached on its right-end such that the list is still steep. The capacity of $[15, 8, 4]$, for example, would be 3 which is the minimum of $15 - (8 + 4)$, $8 - 4$ and 4. Let $(\downarrow)$[1] returns the smaller number of its two arguments, one can easily construct a definition as follows

$$\begin{aligned}
&cap && :: [Int] \rightarrow Int \\
&cap\ [\,] && = \infty \\
&cap\ (ys \succ y) && = (cap\ ys - y) \downarrow y
\end{aligned}$$

to compute capacity for a list. It not hard to find out that a list is steep if its capacity is greater than zero. In other words, the following property, its proof is left to Appendix A.1,

$$steep\ xs = cap\ xs > 0 \tag{3.3}$$

must be satisfied for any list xs. So, instead of parallelising *steep*, our goal now is to construct a list-homomorphic definition for *cap*.

[1]In this thesis, we assume that $(\downarrow)$ has lower associativity than $(+)$, $(-)$, $(*)$, $(/)$ and function application.

To do so, we need to show that *cap* is bi-foldable. From the definition of *cap* above, one can immediately have $cap = foldl\ (\lambda\ c\ z \to (c - z) \downarrow z)\ \infty$. For the *foldr* part, however, one may write down

$$cap\ (x \prec xs) = x - sum\ xs \downarrow cap\ xs\ ,$$

which is almost a *foldr* but nevertheless not one. The reason of *cap* cannot be a *foldr* is that, the *sum xs* part is necessary, which, however, is not providable by *foldr*.

To solve this problem, one can simply use a technique named *tupling* that is a method for generalising the given function by returning more. In this case, we tuple *cap* together with *sum* and thus have $capsum\ xs = (cap\ xs, sum\ xs)$ for any xs. That is, from the attempting of writting *cap* as a *foldr* above, one can come up with

$$x\ \triangleleft\ (c_2, s_2) = ((x - s_2) \downarrow c_2, x + s_2)\ ,$$

and thus have $capsum = foldr\ (\triangleleft)\ (\infty, 0)$. Additionally, since both of *cap* and *sum* are *foldl*, *capsum* can also be written as $foldl\ (\triangleright)\ (\infty, 0)$ where

$$(c_1, s_1)\ \triangleright\ z = ((c_1 - z) \downarrow z, s_1 + z)\ .$$

Now, with $(\triangleleft)$ and $(\triangleright)$, the bi-foldability of *capsum* can be proved by showing that (3.1) can be satisfied. For base case, $x \triangleleft (\infty, 0) = (\infty, 0) \triangleright x$ trivially holds. To show that $(x \triangleleft (c, s)) \triangleright y = x \triangleleft ((c, s) \triangleright y)$ we reason:

Derivation 3.2.a.

$$\begin{aligned}
& (x \triangleleft (c, s)) \triangleright z \\
= \ & \{ \text{ definition of } (\triangleleft) \ \} \\
& ((x - s) \downarrow c, x + s) \triangleright z \\
= \ & \{ \text{ definition of } (\triangleright) \ \} \\
& ((((x - s) \downarrow c) - z) \downarrow z, x + s + z \\
= \ & \{ \ (-z) \text{ distributes over } (\downarrow) \ \} \\
& (((x - s - z) \downarrow (c - z)) \downarrow z, x + s + z) \\
= \ & \{ \text{ arithmetics } \} \\
& ((x - (s + z)) \downarrow ((c - z) \downarrow z), x + s + z) \\
= \ & \{ \text{ definition of } (\triangleleft) \ \} \\
& x \triangleleft ((c - z) \downarrow z, s + z) \\
= \ & \{ \text{ definition of } (\triangleright) \ \} \\
& x \triangleleft ((c, s) \triangleright z) \ .
\end{aligned}$$

In conclusion, although we failed on showing one of *steep* or *cap* could be bi-foldable, we do know that their generalised function, *capsum*, is bi-foldable. Therefore, our goal now is to construct a definition of $(\oplus)$, which must exist by Theorem 2.3.4, such that *capsum* can be written in terms of list-homomorphism.

Constructing a $(\oplus)$

As mentioned in Section 3.1, since $(\triangleright)$ is a special case of $(\oplus)$, a generalized candidate definition of $(\oplus)$ can be created by replacing all occurrences of z in the definition of $(\triangleright)$ by meta-variables as follow,

$$(c, s) \oplus (c2, s2) = ((c - X_1) \downarrow X_2, s + X_3) \ , \qquad (3.4)$$

which must satisfy (3.2) as well.

To refine a definition of $(\oplus)$ from (3.4), one may start with manipulating the inductive case. The derivation of $(x \triangleleft y) \oplus (c_2, s_2) = x \triangleleft (y \oplus (c_2, s_2))$ should contain the same derivation skeleton as Derivation 3.2.a. Therefore we copy steps in Derivation 3.2.a and replace $(\triangleright)$ by $(\oplus)$:

$$\begin{aligned}
& (x \triangleleft (c, s)) \oplus (c_2, s_2) \\
= \;& \{ \text{ definition of } (\triangleleft) \} \\
& ((x - s) \downarrow c, x + s) \oplus (c_2, s_2) \\
= \;& \{ \text{ definition of } (\oplus) \} \\
& ((((x - s) \downarrow c) - X_1) \downarrow X_2, x + s + X_3 \\
= \;& \{ (-X_1) \text{ distributes over } (\downarrow) \} \\
& (((x - s - X_1) \downarrow (c - X_1)) \downarrow X_2, x + s + X_3) \\
= \;& \{ \text{ arithmetics } \} \\
& ((x - (s + X_1)) \downarrow ((c - X_1) \downarrow X_2), x + s + X_3)
\end{aligned}$$

In the next step, we would like to fold back the definition of $(\triangleleft)$, which requires $s + X_1 = s + X_3$, which can be satisfied by unifying x_1 and x_3.

$$\begin{aligned}
& ((x - (s + X_1)) \downarrow ((c - X_1) \downarrow X_2), x + s + X_1) \\
= \;& \{ \text{ definition of } (\triangleleft) \} \\
& x \triangleleft ((c - X_1) \downarrow X_2, s + X_1) \\
= \;& \{ \text{ definition of } (\oplus) \} \\
& x \triangleleft ((c, s) \oplus (c_2, s_2))
\end{aligned}$$

From the derivation, it can be inferred that $(\oplus)$ must have the following form:

$$(c, s) \oplus (c_2, s_2) = ((c - X_1) \downarrow X_2, s + X_1)$$

for some X_1 and X_2. This $(\oplus)$ have to satisfy the base case of (3.2), namely $(c_2, s_2) =$

$(\infty, 0) \oplus (c_2, s_2)$, as well. We reason:

$$
\begin{aligned}
& (\infty, 0) \oplus (c_2, s_2) \\
= \;& \{ \text{ definition of } (\oplus) \; \} \\
& ((\infty - X_1) \downarrow X_2, 0 + X_1) \\
= \;& \{ \text{ an obvious choice would be } X_2 = c_2 \text{ and } X_1 = s_2 \; \} \\
& ((\infty - s_2) \downarrow c_2, 0 + s_2) \\
= \;& \{ \text{ arithmetics } \} \\
& (c_2, s_2)
\end{aligned}
$$

In the end we have our definition of $(\oplus)$ as $(c, s) \oplus (c_2, s_2) = ((c - s_2) \downarrow c_2, s + s_2)$. This $(\oplus)$ has got to be correct because we have the proof already.

3.3 Essential Properties of List-Homomorphism

Synthesizing an $(\oplus)$ with help of the second duality theorem is a very applicable solution, however, it is also an ad-hoc method. In other words, a proof of (3.1), which may not easy to come up with, must be provided case by case. In this section we will try another angle - developing an essential property for $(\oplus)$ by applying *resumable fold* and *inverse function*, hoping that such essential property can also show us some useful hint to construct $(\oplus)$.

The aim is to compute $h\ (xs + ys)$. If $h\ ys$ has been computed, we wish that there exists a function $f_{\triangleleft}$ such that $h\ (xs + ys) = f_{\triangleleft}\ (xs, h\ ys)$. There may exist several such functions, e.g. $(\oplus) \circ (h \times id)$, can serve the role as the $f_{\triangleleft}$. If $h = foldr\ (\triangleleft)\ e$, $f_{\triangleleft}$ can be defined as the following function

$$
\begin{aligned}
& foldrr && :: ((a \times b) \to b) \to ([a], b) \to b \\
& foldrr\ (\triangleleft)\ ([\,], e) && = e \\
& foldrr\ (\triangleleft)\ (x \prec xs, e) && = x \triangleleft (foldrr\ (\triangleleft)\ (xs, e))\,,
\end{aligned}
$$

we thus have $h\ (xs + ys) = foldrr\ (\triangleleft)\ (xs, h\ ys)$ that, with cat as the uncurried $(+)$, can be presented in point-free style as

$$h \circ cat = foldrr\ (\lhd) \circ (id \times h)\ . \tag{3.5}$$

Function *foldrr* is a variantion of *foldr*. It replaces each constructor in xs by $(\lhd)$ and takes the pre-computed part, $h\ ys$, as its base case. Operationally one can image that function *foldrr*, a *resumable foldr*, can *resume* the process of a paused-in-the-middle h.

On the other hand, if $h\ xs$ has been computed first, we wish to define $f_{\rhd}$ such that $h\ (xs +\!\!+\ ys) = f_{\rhd}\ (h\ xs, ys)$. Given that $h = foldl\ (\rhd)\ e$, mimicking the pausing-resuming idea above, one can define $f_{\rhd}$ as a resumalbe *foldl*:

$$\begin{aligned}
&foldlr && :: ((b \times a) \to b) \to (b, [a]) \to b \\
&foldlr\ (\rhd)\ (e, [\,]) && = e \\
&foldlr\ (\rhd)\ (e, ys \succ y) && = (foldlr\ (\rhd)\ (e, ys)) \rhd y\ ,
\end{aligned}$$

and thus have

$$h \circ cat = foldlr\ (\rhd) \circ (h \times id)\ . \tag{3.6}$$

Let $h = foldr\ (\lhd)\ e = foldl\ (\rhd)\ e$, we have (3.5), (3.6) and, by Theorem 2.3.4, there must exists an $(\oplus)$ such that h can be defined as a list-homomorphism. In other words, for that h we also know that

$$h \circ cat = (\oplus) \circ (h \times h)\ . \tag{3.7}$$

Moreover, given (zs, s) such that $zs = xs +\!\!+\ ys \wedge h\ zs = s$ for some xs and ys, all properties we discussed above can be summed up as the following diagram,

Calculating $(\oplus)$

To come up with a definition of that $(\oplus)$, another key concept required here is to use *right inverse* to expand h. Defining h^{-1} as a right inverse of h if for all z we have $h\ (h^{-1}\ z) = z$, which is equivalent to $h = h \circ h^{-1} \circ h$. In set-theoretical model, a right inverse must exist and may not be unique.

$$\begin{tikzcd}
 & & (xs, ys) \arrow[ddll, "(id \times h)"'] \arrow[dddl, "(h \times h)"'] \arrow[ddrr, "(h \times id)"] \arrow[dr, dotted, "cat"'] & & \\
 & & & zs \arrow[ul, dotted, bend right, "cat^{\circ}"'] \arrow[dddl, dotted, "h"] & \\
(xs, u) \arrow[dr, "(h \times id)"] \arrow[ddrr, "foldrr\ (\triangleleft)"'] & & & & (t, ys) \arrow[dlll, "(id \times h)"] \arrow[ddll, "folder\ (\triangleright)"] \\
 & (t, u) \arrow[dr, "(\oplus)"] & & & \\
 & & s & &
\end{tikzcd}$$

Figure 3.1: List-Homomorphism Diagram

Now let us try to develop a property for list-homomorphism. Starting from the left-hand siede of (3.7), we reason:

Derivation 3.3.a.

$$
\begin{aligned}
& h \circ cat \\
= \;& \{ \ (3.5) \ \} \\
& foldrr\ (\triangleleft) \circ (id \times h) \\
= \;& \{ \ h = h \circ h^{-1} \circ h;\ (a \circ b \times f \circ g) = (a \times f) \circ (b \times g) \ \} \\
& foldrr\ (\triangleleft) \circ (id \times h) \circ (id \times h^{-1}) \circ (id \times h) \\
= \;& \{ \ (3.5) \text{ backwards, and } (3.6) \ \} \\
& foldlr\ (\triangleright) \circ (h \times id) \circ (id \times h^{-1}) \circ (id \times h) \\
= \;& \{ \ h = h \circ h^{-1} \circ h;\ (a \circ b \times f \circ g) = (a \times f) \circ (b \times g) \ \} \\
& foldlr\ (\triangleright) \circ (h \times id) \circ (h^{-1} \times id) \circ (h \times id) \circ (id \times h^{-1}) \circ (id \times h) \\
= \;& \{ \ (a \circ b \times f \circ g) = (a \times f) \circ (b \times g) \ \} \\
& foldlr\ (\triangleright) \circ (h \times id) \circ (h^{-1} \times h^{-1}) \circ (h \times h) \\
= \;& \{ \ (3.6) \text{ backwards } \} \\
& h \circ cat \circ (h^{-1} \times h^{-1}) \circ (h \times h)\ .
\end{aligned}
$$

We thus have property $h \circ cat = h \circ cat \circ (h^{-1} \times h^{-1}) \circ (h \times h) = (\oplus) \circ (h \times h)$ and write down a lemma as follows

Lemma 3.3.1. For a bi-foldable function h, let $(\oplus) = h \circ cat \circ (h^{-1} \times h^{-1})$, we have

$$h \circ cat = (\oplus) \circ (h \times h)\ .$$

From now on, we know that, for a bi-foldable function h, there must exists an $(\oplus)$ such that h can be written as a list-homomorphism and a $(\oplus)$ can be defined as shown in Lemma 3.3.1 if a proper h^{-1} is picked.

3.4 Parallelization with Right Inverse

Chi and Mu [5] argued that, in practical, Lemma 3.3.1 may be not as that applicable as one might image since, in many examples, we have failed to find any simple inverse which can lead us to the definition of $(\oplus)$. But it turns out that it may not be that unuseful. In fact, if one can generalise the target function with enough information, the inverse function we need can often be defined by returning a smaller list which has the same properties of the original input list. In this section we will show how to find a definition of $(\oplus)$ with two examples. As the first example, we will show that the *steep* problem can also be solved by applying this inverse method. Secondly we will derive an $(\oplus)$ for computing the maximum prefix sum for a given list in parallel.

Steepness

As we described in Section 3.2, a list is steep if each number is larger than the sum of the numbers to its right. Clearly *steep* can not be written as a *fold* because of the lack of information. So we tuple function *cap*, which computes the bound of input list, and function *sum* as an alternative specification of steep, namely $\mathit{steep} \equiv ({>}\,0) \circ \mathit{fst} \circ \mathit{capsum}$. Function $\mathit{capsum} :: [\mathit{Int}] \to (\mathit{Int}, \mathit{Int})$ can be defined by $\mathit{foldr}\ (\triangleleft)\ (\infty, 0)$ and $\mathit{foldl}\ (\triangleright)\ (\infty, 0)$ where

$$
\begin{aligned}
x \ \triangleleft\ (c_2, s_2) &= ((x - s_2) \downarrow c_2, x + s_2)\ , \\
(c_1, s_1)\ \triangleright\ z &= ((c_1 - z) \downarrow z, s_1 + z)\ .
\end{aligned}
$$

To pick an inverse for *capsum*, one can assume the inverse function will always return a list with two elements and the cap is decided by the difference between these two elements. That is, one may pick the inverse function as $\mathit{capsum}^{-1}(c, s) = [a, b]$

where $a = \frac{s+c}{2}$ and $b = \frac{s-c}{2}$[2]. Now, from lemma 3.3.1 we reason:

$$
\begin{aligned}
& (c_1, s_1) \oplus (c_2, s_2) \\
= \; & \{ \text{ lemma 3.3.1 } \} \\
& \mathit{capsum}\ (\mathit{capsum}^{-1}\ (c_1, s_1) +\!\!+ \mathit{capsum}^{-1}\ (c_2, s_2)) \\
= \; & \{ \text{ let } (a_1, b_1) = (\tfrac{s_1+c_1}{2}, \tfrac{s_1-c_1}{2}) \text{ and } (a_2, b_2) = (\tfrac{s_2+c_2}{2}, \tfrac{s_2-c_2}{2}) \ \} \\
& \mathit{capsum}\ [a_1, b_1, a_2, b_2] \\
= \; & \{ \text{ let } z < \mathit{zs} = [a_1, b_1, a_2, b_2];\ \mathit{capsum} \text{ as a } \mathit{foldr} \ \} \\
& a_1 \ \triangleleft \ (\mathit{capsum}\ \mathit{zs}) \\
= \; & \{ \text{ let } (c, s) = \mathit{capsum}\ \mathit{zs};\ \text{definition of } (\triangleleft) \ \} \\
& ((a_1 - s) \downarrow c, a_1 + s)
\end{aligned}
$$

For deriving further, one may want to directly expend s and c by their definition. But that will lead us to a very tricky derivation which is not the point here. So, instead of just expending s and c, we use three properties below to help us to continue our derivation.

$$s = s_1 + s_2 - a_1 \ , \tag{3.8}$$

$$(a_1 - s) = (c_1 - s_2) \ , \tag{3.9}$$

$$(c_1 - s_2) \downarrow c = (c_1 - s_2) \downarrow c_2 \ . \tag{3.10}$$

Proving the properties above is relatively less important here so we leave it to

[2]Please notice that, to solve $\mathit{capsum}^{-1}(c, s) = [a, b]$ for some (a, b), taking $(a, b) = (\frac{s+c}{2}, \frac{s-c}{2})$ is not the only solution. In fact, this inverse function is designed under an assumption that $c = a - b$. If one takes another assumption, namely $c = b$, then we have another definition of inverse function by taking $(a, b) = (s - c, c)$. This second solution will lead us to the very same $(\oplus)$ as the first one.

Appendix A.2. Now, one can continue the derivation as,

$$
\begin{aligned}
& ((a_1 - s) \downarrow c, a_1 + s) \\
= \;& \{ \text{ by (3.8), (3.9) and (3.10) } \} \\
& ((c_1 - s_2) \downarrow c_2, a_1 + (s_1 + s_2 - a_1)) \\
= \;& \{ \text{ arithmetic } \} \\
& ((c_1 - s_2) \downarrow c_2, s_1 + s_2)
\end{aligned}
$$

This is the very same definition of $(\oplus)$ we discovered in Section 3.2!

Maximum Prefix Sum

As the second example, considering a list of natural numbers, the maximum prefix sum function, $mps :: [Int] \to Int$, returns the biggest summation of each prefix of it. Let $(\uparrow)$ returns the maximum of its two arguments and has the same associativity as $(\downarrow)$, an example of mps could be $mps\ [1,3,-5,1,-2,8] = (mps\ [\,]) \uparrow (mps\ [1]) \uparrow (mps\ [1,3]) \uparrow (mps\ [1,3,-5]) \uparrow (mps\ [1,3,-5,1]) \uparrow (mps\ [1,3,-5,1,-2]) \uparrow (mps\ [1,3,-5,1,-2,8]) = 0 \uparrow 1 \uparrow 4 \uparrow -1 \uparrow 0 \uparrow -2 \uparrow 6 = 6$. This function can easily be defined in terms of $foldr$, but not $foldl$. To overcome this problem, tupling sum with mps is required. Let $mpsum = (mps, sum)$, we have $mpsum = foldr\ (\triangleleft)\ (0,0) = foldl\ (\triangleright)\ (0,0)$ where

$$
\begin{aligned}
x \;\triangleleft\; (m,s) &= (0 \uparrow (x+m), x+s) \;, \\
(m,s) \;\triangleright\; z &= (m \uparrow (s+z), s+z) \;.
\end{aligned}
$$

Notice that mps returns at least 0 since every list has $[\,]$ as one of its prefixes.

Assuming $mpsum\ xs = (m,s)$ for list xs, we then know that $m \geq 0$ and $m \geq s$. The aim now is to define (a,b) in terms of m and s such that $mpsum\ [a,b] = (m,s)$. Immediately, one can take $b = s - a$. And then, since $mps\ [a,b] = 0 \uparrow a \uparrow s = m$ and $m \geq 0 \wedge m \geq s$, an obvious choice is taking $a = m$. As result, we can now define that

$mpsum^{-1}\ (m,s) = [m, s-m]$, and then with Lemma 3.3.1 we derive,

$$
\begin{aligned}
& (m_1,s_1) \oplus (m_2,s_2) \\
= \;& \{ \text{ lemma 3.3.1 } \} \\
& mpsum\ (mpsum^{-1}\ (m_1,s_1) +\!\!+ \ mpsum^{-1}\ (m_2,s_2)) \\
= \;& \{ \text{ definition of } capsum^{-1} \ \} \\
& mpsum\ [m_1, s_1 - m_1, m_2, s_2 - m_2] \\
= \;& \{ \text{ let } (m,s) = mpsum\ [s_1 - m_1, m_2, s_2 - m_2];\ mpsum \text{ as a } foldr \ \} \\
& (0 \uparrow (m_1 + m), m_1 + s) \\
= \;& \{ \ mpsum \text{ as a } foldr \text{ again } \} \\
& (0 \uparrow (m_1 + (0 \uparrow (s_1 - m_1 + m_2))), m_1 + s_1 - m_1 + s_2) \\
= \;& \{ \ (x+) \text{ distributes over } (\uparrow) \ \} \\
& (0 \uparrow m_1 \uparrow (m_1 + s_1 - m_1 + m_2), m_1 + s_1 - m_1 + s_2) \\
= \;& \{ \text{ arithmetic } \} \\
& (0 \uparrow m_1 \uparrow (s_1 + m_2), s_1 + s_2) \\
= \;& \{ \text{ since } m_1 \geq 0 \ \} \\
& (m_1 \uparrow (s_1 + m_2), s_1 + s_2)
\end{aligned}
$$

Finally, we have a list-homomorphic definition of *mpsum* with $(\oplus)$ defined by $(m_1, s_1) \oplus (m_2, s_2) = (m_1 \uparrow (s_1 + m_2), s_1 + s_2)$.

3.5 Limitation

So far in this chapter we have introduced two approaches to construct a list-homomorphism. On the one hand, with the *sufficient conditions* we developed in Section 3.1, one can easily construct a well definition of $(\oplus)$ for list-homomorphism if a proof of bi-foldability can be provided. On the other hand, with the *essential property* we discovered in Section 3.3, an $(\oplus)$ can be simplified if a proper right inverse is chosen. They both serve as good methods for constructing list-homomorphism with its correctness proof from a given function. However, both of them also have limitations.

First of all, both of methods in this chapter require the bi-foldability, namely, both ($\triangleleft$) and ($\triangleright$) are pre-defined and known. But, in most cases, only one of them can easily be produced. The other one would be rather hard to define. For example, the function *mps* we mentioned in last section can be easily defined in terms of *foldr* but not *foldl* due to the lack of necessary information. In fact, the required efforts in finding proper definition for both of ($\triangleleft$) and ($\triangleright$) are often very considerable.

For the second approach, even if ($\triangleleft$) and ($\triangleright$) are provided, one still need to pick a proper right inverse. The tricky part is, as one can see in Section 3.4, where $capsum^{-1}$ and $mpsum^{-1}$ are both designed, picking a right inverse may not be a simple task. In fact, it usually requires us to take several attempts on designing the very h^{-1} we need.

The bottomline is, although we are thirsty on a purely syntactical mechanism to construct a list-homomorphism by using only algebraic manipulations. In the end, it turns out those methods still require some semantical operation in different degree.

Chapter 4

The Duality of Third List-Homomorphism

As beautiful as the third list-homomorphism theorem is, considering the duality between fold and unfold, one may naturally wonder whether there exists a dual theorem? The answer is yes! In this chapter we will discuss a dual theorem and how to derive it from Theorem 2.3.4 by using algebraic manipulations only.

4.1 Dualising the Third List-Homomorphism Theorem

The third list-homomorphism theorem describes requirements of being a list-homomorphism. Similarly, the dual theorem of third list-homomorphism theorem, if it exists, should describe some requirements of being a *parallel unfold*. That is, before starting to develop that dual theorem, we need to find out what a parallel unfold is by revealing its most essential properties.

The very first difference between sequential unfold and parallel unfold is as follows. In sequential unfold, only one sub-list will be generated within each step, thus, the seed-generating function will always spawn merely one single seed. In parallel unfold, on the other hand, n sub-lists will be generated in each step, therefore the seeds-generating function must returns n seeds in each step. In this thesis, we discuss only the situation of $n = 2$ since properties for cases when $n > 3$ can be developed in ways similar to that when $n = 2$.

Considering that we are looking for a dual theorem regarding list-homomorphism, the parallel unfold must satisfy the "reversed" properties of list-homomorphism. More precisely, it must satisfy the properties corresponding to the patterns in list-homomorphism, namely *empty-list*, *singleton-list* and *concatenation-of-two-sub-lists*.

Let function $g_\diamond$ be a two-seeds-generating function and g_v is a value-generating function. A list-generating function k is a *list-unhomomorphism*, denoted by $k = \mathit{unhom}\ g_\diamond\ g_v\ p_1\ p_2$, if k satisfies:

$$\begin{aligned}
&k\ s \mid p_1\ s &&= [\,]\\
&k\ s \mid p_2\ s &&= [g_v\ s]\\
&k\ s \mid (t,u) \leftarrow g_\diamond\ s &&= k\ t \mathbin{+\!\!+} k\ u,
\end{aligned}$$

where the predicates p_1 and p_2 present terminal conditions corresponding to empty-list case and singleton-list case respectively.

In this thesis, we view unfolding as the relational converse of folding, which is the other reason why we restrict our *unfoldr* and *unfoldl* to return finite lists, and a list-unhomomorphism is therefore the relational converse of a list-homomorphism. In other words, for any unfolding function, we demand that successive applications of its seed-generating functions eventually produce seeds which satisfy its terminal condition. This restriction also makes list-unhomomorphism we discussed produce only finite list and hence successive applications of $g_\diamond$ will eventually produce seeds which satisfy either p_1 or p_2.

Given a bi-unfoldable function k, namely $k = \mathit{unfoldr}\ g_\triangleleft\ p = \mathit{unfoldl}\ g_\triangleright\ p$ for some $g_\triangleleft$, $g_\triangleright$ and p. To establish a dual theorem of the third list-homomorphism theorem, the aim is to show that k is a list-unhomomorphism as well. We need to find out some proper $g_\diamond$, p_1 and p_2 by using $g_\triangleleft$, $g_\triangleright$ and p such that $k = \mathit{unhom}\ g_\diamond\ p_1\ p_2$. For predicates p_1 and p_2, a simple choice is to pick $p_1 = p$, $p_2 = p \circ \mathit{snd} \circ g_\triangleleft$ and $g_v = \mathit{fst} \circ g_\triangleleft$. The real challenge here is to come up with a $g_\diamond$. The way to go, instead of directly proving the existence of $g_\diamond$, we apply the same method in Section 3.3 to develop essential properties of $g_\diamond$, which, if satisfied, guarantees the existence of a list-unhomomorphism.

4.2 Essential Properties of List-Unhomomorphism

Conceptually, in Section 3.3, we firstly fold a list by one kind of implementation of folding, say *foldr*, then we stop that folding process in the middle and "switch" to the other implementation, *foldl*, to finish the rest folding process. In that case, we need a resumable fold such that the whole folding process can be finished. Following this idea, we also want to pause an unfolding process in the middle and resume it by another unfolding. To make this possible, a *"pausable" unfolding* is required. Such pausable unfolding returns an intermediate list together with a seed. Taking *unfoldr* as an example, the intermediate list should be a prefix of the final list and the corresponding suffix should be able to be generated by the returned seed. Please notice that a pausable unfolding is non-deterministic because an unfolding process could be paused in any position. This is the place where the relational derivation can help us.

Before we just jump into the definition of pausable unfolding, let's get familiar with relations by taking an exercise. Considering the following situation, given a bi-unfoldable k and an initial seed s, there must exist a list $zs = k\ s$ and, since our goal is to parallelise k, we can arbitrarily cut zs into two lists, say xs and ys. We can easily come up with the following property

$$(xs, ys) \leftsquigarrow (cat^{\circ} \circ k)\ s \equiv \exists zs.k\ s = zs \land xs \mathbin{+\!\!+} ys = zs\ .$$

This property is quite easy to understand and, in fact, is very helpful for understanding some properties we will introduce later.

Now, let's define the pausable unfolding with relation! Given $k = unfoldr\ g_{\triangleleft}\ p$, a relation $r_{\triangleleft}$ is a pausable *unfoldr*, denoted by $unfoldrp\ g_{\triangleleft}\ p$, if given s as an initial seed we have

$$\begin{aligned}(xs, u) \leftsquigarrow r_{\triangleleft}\ s \equiv\ & (xs = [\,] \land s = u) \\ & \lor\ (\exists x, xs'\ .\ xs = x \prec xs' \land (\exists u'.\neg p\ u' \land g_{\triangleleft}\ u' = (x, u) \land (xs', u') \leftsquigarrow r_{\triangleleft}\ s))\ .\end{aligned}$$

That is, $unfoldrp\ g_{\triangleleft}\ p\ s$ relates s to a pair of xs, a prefix of $k\ s$, and a seed u, which the corresponding suffix can be generated from. Symmetrically, for $k = unfoldl\ g_{\triangleright}\ p$

and s as an initial seed, a relation $r_\triangleright$ is a pausable *unfoldl*, denoted by *unfoldlp* $g_\triangleright\ p$, if

$$(t, ys) \leftharpoondown r_\triangleright\ s \equiv\ (ys = [\,] \wedge s = t) \\ \vee\ (\exists ys', y\ .\ ys = ys' \succ y \wedge (\exists t'. \neg p\ t' \wedge g_\triangleright\ t' = (t, y) \wedge (t', ys') \leftharpoondown r_\triangleright\ s))\ .$$

That is, *unfoldlp* $g_\triangleright\ p\ s$ relates s to a pair of ys, a suffix of $k\ s$, and a seed t, which the corresponding prefix can be generated from.

Based on these two definitions above and given $k = \mathit{unfoldr}\ g_\triangleleft\ p = \mathit{unfoldl}\ g_\triangleright\ p$, one can immediately have the following equations,

$$\mathit{cat}^\circ \circ k = (\mathit{id} \times k) \circ \mathit{unfoldrp}\ g_\triangleleft\ p\,; \tag{4.1}$$

$$\mathit{cat}^\circ \circ k = (k \times \mathit{id}) \circ \mathit{unfoldlp}\ g_\triangleright\ p\,, \tag{4.2}$$

by applying k to seed t and u. The commutative diagram shown in Figure 4.1 can help us to get an intuition of properties above.

$$\begin{tikzcd}
 & & (xs, ys) \arrow[dr, dotted, bend right, "\mathit{cat}"'] & & \\
 & & & zs \arrow[ul, dotted, squiggly, bend right, "\mathit{cat}^\circ"'] & \\
(xs, u) \arrow[uurr, "(\mathit{id} \times k)"] & & & & (t, ys) \arrow[uull, "(k \times \mathit{id})"'] \\
 & (t, u) \arrow[ul, "(k \times \mathit{id})"] \arrow[uuur, "(k \times k)"] \arrow[urrr, "(\mathit{id} \times k)"'] & & & \\
 & & s \arrow[ul, "g_\diamond"] \arrow[uull, squiggly, "\mathit{unfoldrp}\ g_\triangleleft\ p"] \arrow[uurr, squiggly, "\mathit{unfoldlp}\ g_\triangleright\ p"'] \arrow[uuur, dotted, "k"'] & &
\end{tikzcd}$$

Figure 4.1: List-Unhomomorphism Diagram

Recall that our goal is to apply the same way as Section 3.3 for discovering essential properties of $g_\diamond$. In other words, we want to mimic Derivation 3.3.a to come up with a similar derivation for list-unhomomorphism, where an essential property of $g_\diamond$ can be found out. Interestingly, Derivation 3.3.a is started with (3.7), which says that two particular paths from (xs, ys) to s in Figure 3.1 are equal. So, it is natural that we want a property what shows two corresponding paths from s to (xs, ys) in Figure 4.1 are also equal. That is, given $k = \mathit{unfoldr}\ g_\triangleleft\ p = \mathit{unfoldl}\ \mathit{rwdg}\ p$, we have

$$\mathit{cat}^\circ \circ k = (k \times k) \circ g_\diamond . \tag{4.3}$$

Now, starting with $\mathit{cat}^\circ \circ\ k$ one can derive that

Derivation 4.2.a.

$$
\begin{aligned}
& \mathit{cat}^\circ \circ k \\
= \quad & \{ \text{ (4.1) } \} \\
& (\mathit{id} \times k) \circ \mathit{unfoldrp}\ g_\triangleleft\ p \\
= \quad & \{\ k = k \circ k^\circ \circ k, \text{ product functor } \} \\
& (\mathit{id} \times k) \circ (\mathit{id} \times k^\circ) \circ (\mathit{id} \times k) \circ \mathit{unfoldrp}\ g_\triangleleft\ p \\
= \quad & \{ \text{ (4.1) backwards, and (4.2) } \} \\
& (\mathit{id} \times k) \circ (\mathit{id} \times k^\circ) \circ (k \times \mathit{id}) \circ \mathit{unfoldlp}\ g_\triangleright\ p \\
= \quad & \{\ k = k \circ k^\circ \circ k, \text{ product functor } \} \\
& (\mathit{id} \times k) \circ (\mathit{id} \times k^\circ) \circ (k \times \mathit{id}) \circ (k^\circ \times \mathit{id}) \circ (k \times \mathit{id}) \circ \mathit{unfoldlp}\ g_\triangleright\ p \\
= \quad & \{ \text{ product functor } \} \\
& (k \times k) \circ (k^\circ \times k^\circ) \circ (k \times \mathit{id}) \circ \mathit{unfoldlp}\ g_\triangleright\ p \\
= \quad & \{ \text{ (4.2) backwards } \} \\
& (k \times k) \circ (k^\circ \times k^\circ) \circ \mathit{cat}^\circ \circ k\ .
\end{aligned}
$$

We thus have $g_\diamond = (k^\circ \times k^\circ) \circ \mathit{cat}^\circ \circ k$! But wait! The $g_\diamond$ we want should be a function rather than a relation like $(k^\circ \times k^\circ) \circ \mathit{cat}^\circ \circ k$. So the property we need here should be $g_\diamond \subseteq (k^\circ \times k^\circ) \circ \mathit{cat}^\circ \circ k$.

In the end, summing up everything we discussed so far, we will obtain the dual theorem of the third list-homomorphism theorem as follows,

Theorem 4.2.1. Given a bi-unfoldable function k for some $g_{\triangleleft}$, $g_{\triangleright}$ and p, there must exists a function

$$g_{\diamond} \subseteq (k^{\circ} \times k^{\circ}) \circ \mathit{cat}^{\circ} \circ k$$

such that $k = \mathit{unhom}\ g_{\diamond}\ (\mathit{fst} \circ g_{\triangleleft})\ p\ (p \circ \mathit{snd} \circ g_{\triangleleft})$.

Calculating $g_{\diamond}$

Although $(k^{\circ} \times k^{\circ}) \circ \mathit{cat}^{\circ} \circ k$ looks very lovely, unfortunately, it is not easy to simplify. However, starting from it, one can derive that

$$\begin{aligned}
& (k^{\circ} \times k^{\circ}) \circ \mathit{cat}^{\circ} \circ k \\
= \quad & \{\ \forall S. S = S \cap S\ \} \\
& ((k^{\circ} \times k^{\circ}) \circ \mathit{cat}^{\circ} \circ k) \cap ((k^{\circ} \times k^{\circ}) \circ \mathit{cat}^{\circ} \circ k) \\
= \quad & \{\ (4.1) \text{ and } (4.2)\ \} \\
& ((k^{\circ} \times k^{\circ}) \circ (\mathit{id} \times k) \circ \mathit{unfoldrp}\ g_{\triangleleft}\ p) \cap \\
& ((k^{\circ} \times k^{\circ}) \circ (k \times \mathit{id}) \circ \mathit{unfoldlp}\ g_{\triangleright}\ p) \\
= \quad & \{\ (f \times g) \circ (j \times k) = (f \circ j \times g \circ k) \text{ and } k^{\circ} \circ k = \mathit{id}\ \} \\
& ((k^{\circ} \times \mathit{id}) \circ \mathit{unfoldrp}\ g_{\triangleleft}\ p) \cap ((\mathit{id} \times k^{\circ}) \circ \mathit{unfoldlp}\ g_{\triangleright}\ p)\ .
\end{aligned}$$

That is, we now have both of the following properties,

$$\begin{aligned}
g_{\diamond} &\subseteq ((k^{\circ} \times \mathit{id}) \circ \mathit{unfoldrp}\ g_{\triangleleft}\ p)\ ; \\
g_{\diamond} &\subseteq ((\mathit{id} \times k^{\circ}) \circ \mathit{unfoldlp}\ g_{\triangleright}\ p)\ .
\end{aligned}$$

Let $g_{\diamond}\ s = (t, u)$ for seeds s, t and u, the first property says that, (t, u) must be related by s by relation $(k^{\circ} \times \mathit{id}) \circ \mathit{unfoldrp}\ g_{\triangleleft}\ p$. In other words, we have

$$(t, u) \leftsquigarrow ((k^{\circ} \times \mathit{id}) \circ \mathit{unfoldrp}\ g_{\triangleleft}\ p)\ s\ .$$

Similarly, for the second property above, we have that

$$(t, u) \leftsquigarrow ((id \times k^{\circ}) \circ \mathit{unfoldlp}\ g_{\triangleright}\ p)\ s\ .$$

Putting everything so far we have together, the following lemma will show itself!

Lemma 4.2.2. Let k be a bi-unfolable with given $g_{\triangleleft}$, $g_{\triangleright}$ and p, the function $g_{\diamond}$ is a subset of $(k^{\circ} \times k^{\circ}) \circ \mathit{cat}^{\circ} \circ k$ if, for all s, we have $g_{\diamond}\ s = (t, u)$ such that

$$(k\ t, u) \leftsquigarrow \mathit{unfoldrp}\ g_{\triangleleft}\ p\ s \wedge (t, k\ u) \leftsquigarrow \mathit{unfoldlp}\ g_{\triangleright}\ p\ s.$$

From now on, we will use Lemma 4.2.2 to help us to calculate $g_{\diamond}$.

4.3 Basic List Generating Functions

Lemma 4.2.2 shows a hint for constructing a list-unhomomorphism from a bi-unfoldable function. However, writing $g_{\diamond}$ as a real function still requires some efforts and is not entirely syntactical. To help readers to get familiar with the way of building a proper $g_{\diamond}$, we show two of the most elementary examples in this section.

Example: map

The well-known function *map*, which applies the given function to each element in the input list, is widely used in plenty of applications. It is also known on being able to be written as *unfoldr* and *unfoldl* with $g_{\triangleleft}\ (x \triangleleft xs) = (f\ x, xs)$, $g_{\triangleright}\ (ys \triangleright y) = (ys, f\ y)$ and $p = \mathit{null}$. Given function f, the function $\mathit{map}\ f$ can obviously be written as a parallel program.

By lemma 4.2.2, to construct a $g_{\diamond}$ such that $\mathit{map}\ f$ is indeed a list-unhomomorphism, we need to find out sufficient conditions of both of following memberships:

$$(k\ ts, us) \leftsquigarrow \mathit{unfoldrp}\ g_{\triangleleft}\ p\ ss$$
$$(ts, k\ us) \leftsquigarrow \mathit{unfoldlp}\ g_{\triangleright}\ p\ ss,$$

where $(ts, us) = g_{\diamond} ss$. Starting with the first membership, for $ss = [\,]$, no requirement will be produced. For non-empty ss, we can reason

$$\begin{aligned}
& (k\ ts, us) \leftsquigarrow \mathit{unfoldrp}\ g_{\lhd}\ p\ ss \\
\equiv\quad & \{\ \text{let } s' \prec ss' = ss\ \} \\
& (k\ ts, us) = ([\,], ss) \lor \\
& (k\ ts, us) \leftsquigarrow ((((f\ s')\ \prec) \times \mathit{id}) \circ \mathit{unfoldrp}\ g_{\lhd}\ p)\ ss' \\
\Leftarrow\quad & \{\ \text{for non-empty } ts\text{, let } t' \prec ts' = ts\ \} \\
& (ts = [\,] \land us = ss) \lor \\
& ((f\ t') \prec k\ ts', us) \leftsquigarrow ((((f\ s')\ \prec) \times \mathit{id}) \circ \mathit{unfoldrp}\ g_{\lhd}\ p)\ ss' \\
\Leftarrow\quad & \{\ \text{induction}\ \} \\
& (ts = [\,] \land us = ss) \lor (f\ t' = f\ s' \land ts' \mathbin{+\!\!+} us = ss') \\
\equiv\ & ts \mathbin{+\!\!+} us = ss
\end{aligned}$$

For the second membership, one will obtain the very same condition as $ts \mathbin{+\!\!+} us = ss$ from another derivation.

That is, to parallelly $map\ f$ to a list, we splits the input list ss into two and applies $map\ f$ to them recursively. In other words, let $ss = ts \mathbin{+\!\!+} us$, we have

$$map\ f\ (ts \mathbin{+\!\!+} us) = (map\ f\ ts) \mathbin{+\!\!+} (map\ f\ us)\ .$$

This equation is valid but not form a definition. As a program $map\ f$ might not terminate since, for exmaple, ts could be empty and the size of us equals that of ss. To avoid this situation, one may construct a terminating definition by enforcing that neither ts nor us could be empty. We thus have $map\ f = \mathit{unhom}\ g_{\diamond}\ g_v\ p_1\ p_2$ where $g_{\diamond}\ ss = (ts, us)$ for some non-empty ts and us such that $ts \mathbin{+\!\!+} us = ss$, $g_v = \mathit{fst} \circ g_{\lhd} = f$, $p_1 = p = \mathit{null}$ and $p_2 = p \circ \mathit{snd} \circ g_{\lhd}$ is the predicate that generates a singleton list. Recall the definition of list-unhomomorphism, one can come up with a easy-to-understand definition

$$\begin{aligned}
map\ f\ [\,] &= [\,]\\
map\ f\ [s] &= [f s]\\
map\ f\ (ts \mathbin{+\!\!+} us) \mid ts \neq [\,] \wedge us \neq [\,] &= map\ f\ ts \mathbin{+\!\!+} map\ f\ us\ .
\end{aligned}$$

Example: fromTo

In the last example, the sufficient condition of two memberships are identical. But not every function has that kind of luck. For example, the function *fromTo* takes a pair of numbers (m, n) and returns a list of numbers from m to n. The *fromTo* is obviously a bi-unfoldable function with

$$\begin{aligned}
p\ (m,n) &= m > n\\
g_{\lhd}\ (m,n) \mid\ m \leq n &= (m, (m+1, n))\\
g_{\rhd}\ (m,n) \mid\ m \leq n &= ((m, n-1), n)\ .
\end{aligned}$$

Assume that $((i,j),(a,b)) = g_{\diamond}\ (m,n)$. To construct $g_{\diamond}$, the aim is to develop sufficient conditions of

$$\begin{aligned}
&(fromTo\ (i,j), (a,b)) \leftsquigarrow unfoldrp\ g_{\lhd}\ p\ (m,n)\\
&((i,j), fromTo\ (a,b)) \leftsquigarrow unfoldlp\ g_{\rhd}\ p\ (m,n)\ .
\end{aligned}$$

Taking the first one as an example, the derivation itself would be an induction on the difference between m and n. For $m > n$, there produce no requirement. For

$m \leq n$ we reason,

$$
\begin{aligned}
& (\mathit{fromTo}\ (i,j),(a,b)) \leftsquigarrow \mathit{unfoldrp}\ g_{\triangleleft}\ p\ (m,n) \\
\equiv\ & \{\ \text{for } m \leq n,\ g_{\triangleleft}\ (m,n) = (m,(m+1,n))\ \} \\
& (\mathit{fromTo}\ (i,j),(a,b)) = ([\,],(m,n)) \vee \\
& (\mathit{fromTo}\ (i,j),(a,b)) \leftsquigarrow (((m \prec) \times \mathit{id}) \circ \mathit{unfoldrp}\ g_{\triangleleft}\ p)\ (m+1,n) \\
\Leftarrow\ & (i > j \wedge a = m \wedge b = n) \vee \\
& (i \prec \mathit{fromTo}\ (i+1,j),(a,b)) \leftsquigarrow (((m\prec) \times \mathit{id}) \circ \mathit{unfoldrp}\ g_{\triangleleft}\ p)\ (m+1,n) \\
\Leftarrow\ & \{\ \text{induction}\ \} \\
& (i > j \wedge a = m \wedge b = n)\ \vee\ (i \leq j \wedge i = m \wedge b = n \wedge j+1 = a) \\
\equiv\ & i \leq j \wedge i = m \wedge b = n \wedge j+1 = a\ .
\end{aligned}
$$

We thus have $i \leq j \wedge i = m \wedge b = n \wedge j+1 = a$ as the sufficient condition for the first membership. On the other hand, for the second membership, namely $((i,j), \mathit{fromTo}\ (a,b)) \in \mathit{unfoldlp}\ g_{\triangleright}\ p\ (m,n)$, one can easily calculate another set of sufficient condition – $a \leq b \wedge i = m \wedge b = n \wedge j+1 = a$. In the end we have a requirement of $g_{\diamond}$ as

$$
\begin{aligned}
& (i \leq j \wedge i = m \wedge b = n \wedge j+1 = a)\ \wedge\ (a \leq b \wedge i = m \wedge b = n \wedge j+1 = a) \\
\equiv\ & m \leq j \wedge j+1 \leq n\ .
\end{aligned}
$$

That is, given (m,n), to parallelly construct a list from m to n, we split the range of m to n into two parts and recursively generate sub-lists from those two sub-ranges. Namely, picking a j such that $m \leq j \wedge (j+1) \leq n$ holds, we have

$$\mathit{fromTo}\ (m,n) = \mathit{fromTo}\ (m,j) \mathbin{+\!\!\!+} \mathit{fromTo}\ (j+1,n)$$

This equation can seve as the definition of *fromTo* if $m > n$. We thus have $\mathit{map}\ f = \mathit{unhom}\ g_{\diamond}\ g_v\ p_1\ p_2$ where $g_{\diamond}\ (m,n) = ((m,j),(j+1,n))$ for some j such that $m \leq j \wedge (j+1) \leq n$, $g_v = \mathit{fst} \circ g_{\triangleleft} = (+1)$, $p_1 = p = (>)$ and $p_2 = p \circ \mathit{snd} \circ g_{\triangleleft} = (==)$. Again, *fromTo* can also be written as

$$
\begin{aligned}
&\mathit{fromTo}\ (m,n) \mid\ m > n = [\,] \\
&\mathit{fromTo}\ (m,n) \mid\ m == n = [m+1] \\
&\mathit{fromTo}\ (m,n) \mid\ m < n = \mathit{fromTo}\ (m,j) \mathbin{+\!\!+} \mathit{fromTo}\ (j+1,n)\,.
\end{aligned}
$$

4.4 Prefix Sum via List-Unhomomorphism

Given a list of numbers *xs*, the *prefix sum* of *xs*, denoted by *ps xs*, is a list of summation of each prefix of *xs*. In this section, we will take prefix sum as another example of constructing a list-unhomomorphism.

Let's start with calculating prefix. Assume that $\mathit{inits} :: [a] \rightarrow [[a]]$ returns a list of all prefixes of the input list. Please notice that, for the reason that we want to compute prefixes with unfolding, the prefixes we talk about here includes only non-empty prefixes[1]. That is, for example, *inits* $[1,2,3]$ returns $[[1],[1,2][1,2,3]]$ rather than $[[\,],[1],[1,2][1,2,3]]$.

It is not hard to find out that *inits* can easily be defined as a *unfoldl* as follow

$$\mathit{inits}\ \mathit{ys} = \mathit{unfoldl}\ (\lambda\ (\mathit{ys} \triangleright y) \rightarrow (\mathit{ys}, \mathit{ys} \triangleright y))\ \mathit{null}\ \mathit{ys}\,.$$

For another direction, however, things become a little more complicated. The seed used here is a list, which means in each step of *unfoldr* a seed will be destructed into a head and a tail. Then the tail will be picked as a new seed. That is, *unfoldr* drop the information of the left-most element in each step. This makes it impossible to compute *inits* by using merely an *unfoldr*, at least not directly. To overcome this problem, one could simply generalise *inits* with an *accumulating parameter*, which is another technique for function generalisation with extra information. Comparing with tupling, which is to generalise with an additional computation, accumulating parameter is generalising with an additional growable data structure. It is a very useful technique especially when one need to record information about what has already been seen.

[1] In the maximum prefix sum example in Section 3.4, we compute prefixes with the empty list since we need *mps* returns at least 0.

The generalised *inits* can be written as an *unfoldr* as follows

$$\mathit{inits}\ \mathit{xs} = \mathit{unfoldr}\ (\lambda\ (a, x \prec \mathit{xs}) \to (a \succ x, (a \succ x, \mathit{xs})))\ (\mathit{null} \circ \mathit{snd})\ ([\,], \mathit{xs})\ .$$

Therefore, we define a function $\mathit{perfix} :: ([a],[a]) \to [[a]]$ by

$$\mathit{prefix}\ (\mathit{zs}, \mathit{xs}) = \mathit{map}\ (\mathit{zs} \mathbin{+\!\!+})\ (\mathit{inits}\ \mathit{xs})$$

which satisfies $\mathit{inits}\ \mathit{xs} = \mathit{perfix}\ ([\,], \mathit{xs})$. This *prefix* can be written as a $\mathit{unfoldr}\ g_{\triangleleft}\ p$ and a $\mathit{unfoldl}\ g_{\triangleright}\ p$ where

$$\begin{aligned} p &= \mathit{null} \circ \mathit{snd} \\ g_{\triangleleft}\ (\mathit{ws}, x \prec \mathit{xs}) &= (\mathit{ws} \succ x, (\mathit{ws} \succ x, \mathit{xs})) \\ g_{\triangleright}\ (\mathit{ws}, \mathit{ys} \succ y) &= ((\mathit{ws}, \mathit{ys}), \mathit{ws} \mathbin{+\!\!+} (\mathit{ys} \succ y)) \end{aligned}$$

and thus a list-unhomomorphism. The domain of $g_{\triangleleft}$ and $g_{\triangleright}$ are pair whose second components are non-empty list.

Back to the prefix sum problem, taking

$$\mathit{map}\ \mathit{sum} \circ \mathit{prefix}\ ,$$

as the specification of prefix sum, the real challenge is, can we come up with a definition of list-unhomomorphism for $\mathit{map}\ \mathit{sum} \circ \mathit{prefix}$?

To solve this problem, one may have two ways to go: the first and straightforward way is, since *prefix* are already bi-unfoldable, one can apply *map-unfold fusion* to obtain another bi-unfoldable function then construct a proper $g_{\diamond}$ for it. The alternative approach will be slightly tricky. Considering the characteristics of *map* and list-unhomomorphism, it is no surprising that there is a *map-list-unhomomorphism fusion theorem*. Therefore, as an alternative approach, one can constrcut a $g_{\diamond}$ for *prefix* then try to fuse *map sum* to it. In the rest of this section, we will show both approaches.

fusion-first

Given bi-unfoldable *prefix* with $g_{\triangleleft}$, $g_{\triangleright}$ and p defined as above. The main idea of the first approach is to apply *map*-unfold fusion to construct $\mathit{map}\ \mathit{sum} \circ \mathit{prefix}$ bi-unfoldable such that one can define it as a list-unhomomorhpsim.

In fact, since the *map*-unfold fusion theorem requires nothing of the mapped function, one can derive with a general specification. Let $\mathit{prefixf} = \mathit{map}\ f \circ \mathit{prefix}$ for any function f. By *map*-unfold fusion theorem we have $\mathit{prefixf} = \mathit{unfoldr}\ g'_{\triangleleft}\ p' = \mathit{unfoldl}\ g'_{\triangleright}\ p'$ where $p' = p$, $g'_{\triangleleft} = (f \times \mathit{id}) \circ g_{\triangleleft}$ and $g'_{\triangleright} = (\mathit{id} \times f) \circ g_{\triangleright}$. The aim now is to construct a definition of $g'_{\diamond}$ such that $\mathit{prefixf} = \mathit{unhom}\ g'_{\diamond}\ g'_{v}\ p'_1\ p'_2$. where $g'_v = \mathit{fst} \circ g'_{\triangleleft}$, $p'_1 = p'$ and $p'_2 = p \circ \mathit{snd} \circ g'_{\triangleleft}$.

Assume $((ws_l, ts), (ws_r, us)) = g'_{\diamond}\ (ws, ss)$ for any non-empty ss, we have

$$\begin{gathered}(\mathit{prefixf}\ (ws_l, ts), (ws_r, us)) \leftsquigarrow \mathit{unfoldrp}\ g'_{\triangleleft}\ p\ (ws, ss)\\ ((ws_l, ts), \mathit{prefixf}\ (ws_r, us)) \leftsquigarrow \mathit{unfoldlp}\ g'_{\triangleright}\ p\ (ws, ss)\ .\end{gathered}$$

The base case, ss is a singlton list, trivally holds. For inductive case, since both of above memberships will produce the same requirement, we therefore show only the derivation of first membership as follows,

$$\begin{aligned}
&\quad (\mathit{prefixf}\ (ws_l, ts), (ws_r, us)) \leftsquigarrow \mathit{unfoldrp}\ g'_{\triangleleft}\ p\ (ws, ss)\\
&\equiv \quad \{\ \text{let } s' \prec ss' = ss\ \}\\
&\quad (\mathit{prefixf}\ (ws_l, ts), (ws_r, us)) = (ws, s' \prec ss') \vee\\
&\quad (\mathit{prefixf}\ (ws_l, ts), (ws_r, us)) \leftsquigarrow ((((f\ (ws \succ s'))\ \prec) \times \mathit{id}) \circ \mathit{unfoldrp}\ g'_{\triangleleft}\ p)\ (ws \succ s', ss')\\
&\Leftarrow \quad \{\ \text{for non-empty } ts\text{, let } t' \prec ts' = ts\ \}\\
&\quad (ts = [\,] \wedge us = ss \wedge ws_r = ws) \vee\\
&\quad (f\ (ws_l \succ t')) \prec \mathit{prefixf}\ (ws_l \succ t', ts'), (ws_r, us))\\
&\qquad \leftsquigarrow ((((f\ (ws \succ s'))\ \prec) \times \mathit{id}) \circ \mathit{unfoldrp}\ g'_{\triangleleft}\ p)\ (ws \succ s', ss')\\
&\Leftarrow \quad \{\ \text{induction}\ \}\\
&\quad (ts = [\,] \wedge us = ss \wedge ws_r = ws) \vee\\
&\quad (f\ (ws_l \succ t') = f\ (ws \succ s') \wedge ws_l \succ t' = ws \succ s' \wedge ts' +\!\!+\ us = ss' \wedge ws_r = (ws_l \succ t') +\!\!+\ ts')\\
&\equiv ws_l = ws \wedge ts +\!\!+\ us = ss \wedge ws_r = ws_l +\!\!+\ ts
\end{aligned}$$

Taking $f = sum$, one can easily write down a parallel function, denoted by ps, which computes a prefix sum for its input list, as follows

$$\begin{aligned} ps\ (ws, [\,]) &= [\,] \\ ps\ (ws, [s]) &= [sum\ (ws \succ s)] \\ ps\ (ws, ts \mathbin{+\!\!\!+} us) &= ps\ (ws, ts) \mathbin{+\!\!\!+} ps\ (ws \mathbin{+\!\!\!+} ts, us)\ . \end{aligned}$$

constructing-$g_\diamond$-first

Given bi-unfoldable *prefix* with $g_\triangleleft$, $g_\triangleright$ and p defined as above, the other approach is constrcuting a $g_\diamond$ for *prefix* first followed by fusing *map sum* to that paralle *prefix*.

Assume $((ws_l, ts), (ws_r, us)) = g_\diamond\ (ws, ss)$ for any non-empty ss, from Lemma 4.2.2, we know that

$$\begin{aligned} &(prefix\ (ws_l, ts), (ws_r, us)) \leftsquigarrow unfoldrp\ g_\triangleleft\ p\ (ws, ss) \\ &((ws_l, ts), prefix\ (ws_r, us)) \leftsquigarrow unfoldlp\ g_\triangleright\ p\ (ws, ss)\ . \end{aligned}$$

They both produce the same requirements, we therefore show only the derivation of the first membership.

$$\begin{aligned} &\quad (prefix\ (ws_l, ts), (ws_r, us)) \leftsquigarrow unfoldrp\ g_\triangleleft\ p\ (ws, ss) \\ \equiv &\quad \{\ \text{let } s' \prec ss' = ss\ \} \\ &\quad (prefix\ (ws_l, ts), (ws_r, us)) = (ws, s' \prec ss') \vee \\ &\quad (prefix\ (ws_l, ts), (ws_r, us)) \leftsquigarrow ((((ws \succ s')\ \prec) \times id) \circ unfoldrp\ g_\triangleleft\ p)\ (ws \succ s', ss') \\ \Leftarrow &\quad \{\ \text{for non-empty } ts, \text{ let } t' \prec ts' = ts\ \} \\ &\quad (ts = [\,] \wedge us = ss \wedge ws_r = ws) \vee \\ &\quad ((ws_l \succ t') \prec prefix\ (ws_l \succ t', ts'), (ws_r, us)) \\ &\qquad \leftsquigarrow ((((ws \succ s')\ \prec) \times id) \circ unfoldrp\ g_\triangleleft\ p)\ (ws \succ s', ss') \\ \Leftarrow &\quad \{\ \text{induction}\ \} \\ &\quad (ts = [\,] \wedge us = ss \wedge ws_r = ws) \vee (ws_l \succ t' = ws \succ s' \wedge ts' \mathbin{+\!\!\!+} us = ss' \wedge ws_r = (ws_l \succ t') \mathbin{+\!\!\!+} ts') \\ \equiv &\ ws_l = ws \wedge ts \mathbin{+\!\!\!+} us = ss \wedge ws_r = ws_l \mathbin{+\!\!\!+} ts \end{aligned}$$

Now one can define that $g_\diamond\ (ws, ts + us) = ((ws, ts), (ws + ts, us))$ and thus have

$$prefix = unhom\ g_\diamond\ g_v\ p_1\ p_2$$

where $g_v = fst \circ g_\triangleleft$, $p_1 = p$ and $p_2 = p \circ snd \circ g_\triangleleft$. With this definition, now the goal is trying to fuse *map sum* to *prefix*. To do so, let's introduce the map-list-unhomomorphism fusion theorem proved in Appendix A.3.

Theorem 4.4.1. [Map-List-Unhomomorphism Fusion] Given a list-unhomomorphism $k = unhom\ g_\diamond\ g_v\ p_1\ p_2$, we have

$$map\ f \circ k = unhom\ g^\dagger_\diamond\ g^\dagger_v\ p^\dagger_1\ p^\dagger_2$$

if $g^\dagger_v = f \circ g_v$ and $g^\dagger_\diamond = g_\diamond \wedge p^\dagger_1 = p_1 \wedge p^\dagger_2 = p_2$.

Taking $f = sum$, by Theorem 4.4.1 we finally have parallel prefix sum, denoted by ps, as follows

$$ps = map\ sum \circ unhom\ g_\diamond\ g_v\ p_1\ p_2 = unhom\ g_\diamond\ (sum\ g_v)\ p_1\ p_2\ .$$

This equation can easily be expanded to the definition we revealed in the fusion-first approach.

4.5 Discussion

In this chapter we have shown how to construct a list-unhomomorphism for a bi-unfoldable function k with the help of Lemma 4.2.2. The method we introduced in this chapter provides a very helpful derivation framework such that one can construct a parallel list generating function together with its correctness proof. Still, there are some shortcomings that are worthwhile to bring up and discuss.

First of all, even we have Lemma 4.2.2, there is still some human effort required to come up with a real definition of $g_\diamond$. That is, this method is not as syntactical as we were looking forward. Then, as one may noticed, this method basically provides only a way to construct a correct parallel program but no guarantee on its efficiency.

For example, the function *ps* we shown in Section 4.4 is inefficient because we have to compute *sum* for each prefix.

Although this method might not be a realistic way to directly translate a list generating function into its parallel version, from what we can see in Section 4.4, it can still helpful in a way of program derivation. In Section 4.4 we constructed a parallel program for *prefix sum* by deriving a composition of two list-unhomomorphisms, $map\ f$ and *inits*. Therefore, even we can not have a syntactical way to parallelise a list generating function, we can still derive and build a parallel program by applying list-unhomomorphism related theorems and properties.

Chapter 5

Conclusion

In this chapter, we give a summary of this thesis, and point out some possible future work.

5.1 Contributions

The capability of parallel programming is important for programmers nowadays. But it takes lots of experience and efforts to write a parallel program. In this thesis, we try to summarise several methods, as syntactical as possible, to make developing parallel programs easier and faster. First of all, we use formal program construction, more precisely, program derivation, as our developing method. This gives us the capability of manipulating program's definition as mathematical structure and hence provides correctness as the same way in algebraic calculation. Then, to be able to derive and calculate parallel programs, a formal model is required. Therefore we use a mathematical structure named *homomorphism* to model parallel programs such that we can calculate and derive a parallel definition from its specification. We have also restricted ourselves to focus on folding or unfolding lists.

Based on the third list-homomorphism theorem, we have summarised two methods for constructing parallel folding programs.

- After studying some proofs, we noticed the similarity between the proof of bi-foldability and the proof of list-homomorphism. As result, we have developed a mechanism to syntactically derive a definition of list-homomorphism for f from the proof of bi-foldability. This is the first method we presented in Section

3.1.

- The second method presented in Section 3.3 says that, for a bi-foldable function f, one can also construct a list-homomorphic definition if a proper right inverse of f can be chosen. We used to think that an inverse is not easy to find, or, even if we have $(\oplus)$ constructed, we cannot find any simple inverse to explain its discovery. But it turns out that this inverse-based method may not be that hard to apply. In fact, it seems there is an approach to construct an inverse by algebraic calculation.

Dually, we have also shown a method to construct parallel list-generating programs. For any bi-unfoldable function, to construct its list-unhomomorphic definition means that there must exist a seeds-generating function $g_{\diamond}$ such that these two memberships in Lemma 4.2.2 can be satisfied. Therefore, the way to go is to refine the conditions derived from those two memberships and thus one can construct a list-homomorphic definition. Our contributions are,

- We use another notation for represneting a relation and make it looks much like a function. This makes our derivations and properties in Section 4.2 and 4.4 much friendly for reader who is not familiar with relation.

- Moreover, based on that notation, we also developed a diagram, Figure 4.1, to show the relationships among input data, output data and immediate data. Besides, this very same diagram can also show all properties we used and hence helps reader to get an intuition. Additionally, we also found out that if we "revert" the arrows in Figure 4.1, we get a diagram that captures all mathematical properties for list-homomorphism.

5.2 Future Work

Besides limitations and shortcomings we mentioned in this thesis, there are still improvements that can be done.

- In this thesis we model only the linear structure, however, there are tons of other data structures used in real world. So the very first improvement is to generalise the methods to other data structures.

- Moreover, as one can see, Figure 3.1 and Figure 4.1 are nothing but the same diagram with opposite arrows. It is natural one would wonder: does there exist a general diagram where some joint properties might be able to be found? and, are there other diagrams for other data structures? Both of these questions opens the door to another researching topic.

- In Section 4.4, we introduced and appied the map-list-unhomomorphism fusion theorem. This enlighten us that there may be other properties related to list-homomorphism or list-unhomomorphism that can be useful in the way of deriving a parallel program from its specification.

- Homomorphism is a general mathematical structure and we used it for modeling the ideal parallel programs. Interestingly, there is an assumption for using it – those parallel programs will be executed on a system where processes are independent of each other. It is a wonderful assumption if we only discuss in a purely functional world. However, if we change our point of view from functional programming to distributed system, it not hord to find out that, the distributed system a homomorphism described is quite limitative because that assumption leads us the lack of communication among instances or processes. This brings up another opportunity of improvement. That is, to increase the suitability of homomorphism-based methods, different homomorphism-liked mathematical structures, and its mathematical properties, are required for different distributed systems. As an example, the BSP-homomorphism [13] is a mathematical structure designed for the *bulk synchronous parallel model*.

Bibliography

[1] Backhouse, R. *Program Construction: Calculating Implementations from Specifications*. Wiley, 2003.

[2] Bird, R. *Introduction to Functional Programming using Haskell*, second ed. Prentice Hall, 1998.

[3] Bird, R. S. An introduction to the theory of lists. In *Proceedings of the NATO Advanced Study Institute on Logic of programming and calculi of discrete design* (New York, NY, USA, 1987), Springer-Verlag New York, Inc., pp. 5–42.

[4] Bird, R. S., and de Moor, O. *Algebra of Programming*. Prentice Hall International series in computer science. Prentice Hall, 1997.

[5] Chi, Y.-Y., and Mu, S.-C. Constructing list homomorphisms from proofs. In *Proceedings of the 9th Asian conference on Programming Languages and Systems* (Berlin, Heidelberg, 2011), APLAS'11, Springer-Verlag, pp. 74–88.

[6] Emoto, K., Fischer, S., and Hu, Z. Generate, test, and aggregate - a calculation-based framework for systematic parallel programming with mapreduce. In *ESOP* (2012), pp. 254–273.

[7] Geser, A., and Gorlatch, S. Parallelizing functional programs by generalization. *J. Funct. Program. 9*, 6 (Nov. 1999), 649–673.

[8] Gibbons, J. The third homomorphism theorem. *Journal of Functional Programming 6*, 4 (1996), 657–665.

[9] Gorlatch, S. Systematic extraction and implementation of divide-and-conquer parallelism. In PLILP (1996), H. Kuchen and S. D. Swierstra, Eds., vol. 1140 of Lecture Notes in Computer Science, Springer, pp. 274–288.

[10] Hu, Z., Iwasaki, H., and Takeichi, M. Construction of list homomorphisms by tupling and fusion. In MFCS (1996), W. Penczek and A. Szalas, Eds., vol. 1113 of Lecture Notes in Computer Science, Springer, pp. 407–418.

[11] Hughes, J. Why Functional Programming Matters. Computer Journal 32, 2 (1989), 98–107.

[12] Kaldewaij, A. Programming: The Derivation of Algorithms. Prentice Hall International Series in Computer Science. Prentice Hall International, 1990.

[13] Legaux, J., Hu, Z., Loulergue, F., Matsuzaki, K., and Tesson, J. Programming with bsp homomorphisms. In Euro-Par (2013), F. Wolf, B. Mohr, and D. an Mey, Eds., vol. 8097 of Lecture Notes in Computer Science, Springer, pp. 446–457.

[14] Liu, Y., Emoto, K., and Hu, Z. A generate-test-aggregate parallel programming library: systematic parallel programming for mapreduce. In PMAM (2013), pp. 71–81.

[15] Liu, Y., Hu, Z., and Matsuzaki, K. Towards systematic parallel programming over mapreduce. In Euro-Par (2) (2011), E. Jeannot, R. Namyst, and J. Roman, Eds., vol. 6853 of Lecture Notes in Computer Science, Springer, pp. 39–50.

[16] Morihata, A. A short cut to parallelization theorems. In ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013 (2013), pp. 245–256.

[17] Morihata, A., Matsuzaki, K., Hu, Z., and Takeichi, M. The third homomorphism theorem on trees: downward & upward lead to divide-and-conquer. In POPL (2009), Z. Shao and B. C. Pierce, Eds., ACM, pp. 177–185.

[18] Morita, K., Morihata, A., Matsuzaki, K., Hu, Z., and Takeichi, M. Automatic inversion generates divide-and-conquer parallel programs. In PLDI (2007), pp. 146–155.

[19] Mu, S.-C., and Morihata, A. Generalising and dualising the third list-homomorphism theorem: functional pearl. In Proceedings of the 16th ACM SIGPLAN international conference on Functional programming (New York, NY, USA, 2011), ICFP '11, ACM, pp. 385–391.

Appendix A

Missing Proofs

A.1 Property (3.3)

The correctness of $\mathit{steep} = (>0) \circ \mathit{cap}$ can be shown by an induction on the input list. For the base case, we show that,

$$
\begin{aligned}
& \mathit{steep}\ [\,] = \mathit{True} = \infty > 0 \\
= \ & \mathit{cap}\ \infty > 0\,.
\end{aligned}
$$

For inductive case, we reason,

$$
\begin{aligned}
& \mathit{steep}\ (x \prec \mathit{xs}) \\
= & \quad \{\ \text{definition of } \mathit{steep}\ \} \\
& x > \mathit{sum}\ \mathit{xs} \wedge \mathit{steep}\ \mathit{xs} \\
= & \quad \{\ \text{induction}\ \} \\
& x > \mathit{sum}\ \mathit{xs} \wedge \mathit{cap}\ \mathit{xs} > 0 \\
= & \quad \{\ \text{arithmetic}\ \} \\
& x - \mathit{sum}\ \mathit{xs} > 0 \wedge \mathit{cap}\ \mathit{xs} > 0 \\
= & \quad \{\ m > i \wedge n > i = (m \downarrow n) > i\ \} \\
& (x - \mathit{sum}\ \mathit{xs} \downarrow \mathit{cap}\ \mathit{xs}) > 0 \\
= & \quad \{\ \text{definition of } \mathit{cap}\ \} \\
& \mathit{cap}\ (x \prec \mathit{xs}) > 0\,.
\end{aligned}
$$

A.2 Properties (3.8), (3.9) and (3.10)

Given $(c_1, s_1) = \mathit{capsum}\ xs$ and $(c_2, s_2) = \mathit{capsum}\ ys$ for some xs and ys. By the definition of capsum^{-1} we given in Section 3.4, let $(a_1, b_1) = (\frac{s_1+c_1}{2}, \frac{s_1-c_1}{2})$ and $(a_2, b_2) = (\frac{s_2+c_2}{2}, \frac{s_2-c_2}{2})$ then we have the following equations.

$$\mathit{sum}\ [a_1, b_1] = a_1 + b_1 = \frac{s_1 + c_1}{2} + \frac{s_1 - c_1}{2} = s_1 \tag{A.1}$$

$$\mathit{sum}\ [a_2, b_2] = a_2 + b_2 = \frac{s_2 + c_2}{2} + \frac{s_2 - c_2}{2} = s_2 \tag{A.2}$$

$$\mathit{cap}\ [a_1, b_1] = (a_1 - b_1) \downarrow b_1 = \frac{s_1 + c_1}{2} - \frac{s_1 - c_1}{2} \downarrow b_1 = c_1 \downarrow b_1 = c_1 \tag{A.3}$$

$$\mathit{cap}\ [a_2, b_2] = (a_2 - b_2) \downarrow b_2 = \frac{s_2 + c_2}{2} - \frac{s_2 - c_2}{2} \downarrow b_2 = c_2 \downarrow b_2 = c_2 \tag{A.4}$$

That is, xs and $[a_1, b_1]$ have the identical capability and summation, as well as ys and $[a_2, b_2]$.

Now, let $(c, s) = \mathit{capsum}\ [b_1, a_2, b_2]$, (3.8) and (3.9) can be easily proved as the following derivations:

$$\begin{aligned}
& s \\
= \quad & \{ \text{ definition of } \mathit{sum} \ \} \\
& b_1 + a_2 + b_2 \\
= \quad & \{ \text{ arithmetic } \} \\
& (a_1 + b_1 + a_2 + b_2) - a_1 \\
= \quad & \{ \text{ (A.1) and (A.2) } \} \\
& s_1 + s_2 - a_1
\end{aligned}$$

$$
\begin{array}{ll}
& (a_1 - s) \\
= & \{ \text{ by } (3.8) \text{ } \} \\
& (a_1 - (s_1 + s_2 - a_1)) \\
= & \{ \text{ arithmetic } \} \\
& (2 * a_1 - (s_1 + s_2)) \\
= & \{ \text{ definition of } a_1 \text{ } \} \\
& (2 * (\frac{s_1 + c_1}{2}) - s_1 - s_2) \\
= & \{ \text{ arithmetic } \} \\
& (c_1 - s_2)
\end{array}
$$

To prove (3.10), however, another equation is necessary. Starting with $c_1 - s_2$, one could show that

$$
\begin{array}{ll}
& c_1 - s_2 \\
= & \{ \text{ definition of } c_1 \text{ } \} \\
& (a_1 - b_1 \downarrow b_1) - s_2 \\
= & \{ \text{ } (-s_2) \text{ distributes over } (\downarrow) \text{ } \} \\
& a_1 - b_1 - s_2 \downarrow b_1 - s_2 \\
\equiv & \{ \text{ } (A.2) \text{ } \} \\
& a_1 - b_1 - s_2 \downarrow b_1 - a_2 - b_2 .
\end{array}
$$

Now, to prove (3.10), we reason

$$\begin{aligned}
& c_1 - s_2 \downarrow c \\
= \quad & \{ \ c = cap\ [b_1, a_2, b_2] \ \} \\
& c_1 - s_2 \downarrow b_1 - a_2 \downarrow a_2 - b_2 \downarrow b_2 \\
= \quad & \{ \ \text{the last derivation} \ \} \\
& a_1 - b_1 - s_2 \downarrow b_1 - a_2 - b_2 \downarrow b_1 - a_2 \downarrow a_2 - b_2 \downarrow b_2 \\
= \quad & \{ \ x - n \downarrow x = x - n \Leftarrow n \geq 0 \ \} \\
& a_1 - b_1 - s_2 \downarrow b_1 - a_2 - b_2 \downarrow a_2 - b_2 \downarrow b_2 \\
= \quad & \{ \ \text{the last derivation, } \mathit{reverse!} \ \} \\
& c_1 - s_2 \downarrow a_2 - b_2 \downarrow b_2 \\
= \quad & \{ \ (\text{A.4}) \ \} \\
& c_1 - s_2 \downarrow c_2 .
\end{aligned}$$

A.3 Theorem 4.4.1

The map-list-unhomomorphism fusion theorem says that, given a list-unhomomorphism $k = \mathit{unhom}\ g_{\diamond}\ g_v\ p_1\ p_2$, we have

$$\mathit{map}\ f \circ k = \mathit{unhom}\ g_{\diamond}^{\dagger}\ g_v^{\dagger}\ p_1^{\dagger}\ p_2^{\dagger}$$

if $g_v^{\dagger} = f \circ g_v$ and $g_{\diamond}^{\dagger} = g_{\diamond} \wedge p_1^{\dagger} = p_1 \wedge p_2^{\dagger} = p_2$. This theorem can be proved by an induction on seed generating sequence since we restrict that all successive applications of $g_{\triangleleft}$ and $g_{\triangleright}$ eventually reaches some s such that either $p_1\ s$ or $p_2\ s$ is ture. For $p_1\ s$ is true, 4.4.1 trivally holds if $p_1^\dagger = p_1$. For $p_2\ s$ is true, one can reason

$$
\begin{aligned}
& \mathit{map}\ f\ (\mathit{unhom}\ g_\diamond\ g_v\ p_1\ p_2\ s) \\
=\ & \{\ \text{assumption: } p_2\ s \text{ is true}\ \} \\
& \mathit{map}\ f\ [g_v\ s] \\
=\ & \{\ \text{definition of } \mathit{map}\ \} \\
& [f\ (g_v\ s)] \\
=\ & \{\ g_v^\dagger = f \circ g_v \wedge p_2^\dagger = p_2\ \} \\
& \mathit{unhom}\ g_\diamond^\dagger\ g_v^\dagger\ p_1^\dagger\ p_2^\dagger\ s.
\end{aligned}
$$

Finally, for the inductive case, we have,

$$
\begin{aligned}
& \mathit{map}\ f\ (\mathit{unhom}\ g_\diamond\ g_v\ p_1\ p_2\ s) \\
=\ & \{\ \text{let } (t, u) = g_\diamond\ s\ \} \\
& \mathit{map}\ f\ (\mathit{unhom}\ g_\diamond\ g_v\ p_1\ p_2\ t +\!\!+\ \mathit{unhom}\ g_\diamond\ g_v\ p_1\ p_2\ u) \\
=\ & \{\ \mathit{map}\ f\ (\mathit{xs} +\!\!+\ \mathit{ys}) = (\mathit{map}\ f\ \mathit{xs}) +\!\!+\ (\mathit{map}\ f\ \mathit{ys})\ \} \\
& (\mathit{map}\ f\ \mathit{unhom}\ g_\diamond\ g_v\ p_1\ p_2\ t) +\!\!+\ (\mathit{map}\ f\ \mathit{unhom}\ g_\diamond\ g_v\ p_1\ p_2\ u) \\
=\ & \{\ \text{induction}\ \} \\
& \mathit{unhom}\ g_\diamond^\dagger\ g_v^\dagger\ p_1^\dagger\ p_2^\dagger\ t +\!\!+\ \mathit{unhom}\ g_\diamond^\dagger\ g_v^\dagger\ p_1^\dagger\ p_2^\dagger\ u \\
=\ & \{\ g_\diamond^\dagger = g_\diamond\ \} \\
& \mathit{unhom}\ g_\diamond^\dagger\ g_v^\dagger\ p_1^\dagger\ p_2^\dagger\ s.
\end{aligned}
$$