Homomorphism-based Structured Parallel Programming

(準同形に基づいた構造化並列プログラミングに関する研究)

江本 健斗

Committee

Professor	Masato Takeichi
Professor	Kazuo Murota
Professor	Masaaki Sugihara
Professor	Zhenjiang Hu
Associate Professor	Kenjiro Taura
Associate Professor	Shin-Cheng Mu

Acknowledgments

I would like to acknowledge all those people who kindly supported and assisted me during my research.

First of all, I would like to give my heartfelt thanks to Professor Masato Takeichi, who had been my supervisor at my doctoral course of the graduate school of the University of Tokyo, for his kind and helpful encouragement and guidance during my research.

I am also full of gratitude to Professor Zhenjiang Hu for giving me many valuable suggestions and great encouragement through a lot of helpful discussions.

Special thanks go Dr. Kiminori Matsuzaki, Dr. Kazuhiko Kakehi, and Professor Hideya Iwasaki for their insightful comments and kind encouragements to carry out this research.

I would like to thank Dr. Guy Lewis Steele Jr. and Dr. Jan-Willem Maessen for valuable suggestions through a lot of fruitful discussions.

I also thank all members of the laboratory. Especially, I thank my clever colleagues, Dr. Akimasa Morihata and Dr. Kazutaka Matsuda, for various kinds of suggestions during my laboratory life.

The thesis is composed of several research papers I have written and submitted to journals, conferences or workshops. I acknowledge several anonymous referees who gave me many helpful comments during the process of the research. The committee members gave many helpful comments to this thesis.

Abstract

In recent years, parallel programming has become an essential task for programmers, but it is still a very difficult task for most programmers. There are two difficult problems in parallel programming: difficulty of making (designing) efficient parallel algorithms, and difficulty of implementing the parallel algorithms. The former problem is caused by lack of useful characterization of parallel algorithms and systematic methods for deriving efficient parallel algorithms. The latter problem is caused by the need to implement the extra considerations, such as distribution of tasks among processors, communication of data among processors, and synchronization of progress of the tasks, which are not required in sequential programming.

Structured parallel programming, also known as *skeletal parallel programming*, has been proposed as one promising methodology to solve the problems of parallel programming. In this methodology, programmers build parallel programs (algorithms) by composing *skeletons*, i.e., ready-made parallel components that have efficient parallel implementation on various parallel architectures. This approach has the advantage that user programmers can compose parallel programs easily without considering difficult low-level parallelism, because low-level parallelism is completely concealed in skeletons. Once we compose parallel algorithms described with skeletons, we can straightforwardly implement them with skeleton implementation, which is usually provided as either libraries on existing programming languages or new programming languages including skeletons as their primitive constructs.

One difficult problem in skeletal parallel programming is to provide a methodology to develop efficient parallel algorithms. Especially, optimization of skeleton compositions to improve their efficiency is very important, because it provides us with a systematic way to compose efficient parallel algorithms from naively composed skeleton programs. In general, naively composed skeleton programs suffer from inefficiency caused by redundancy of compositions, such as redundant intermediate data structures communicated between composed skeletons.

The thesis tackles the problem by homomorphism-based design of parallel skeletons that have good properties to develop efficient parallel algorithms by fusion. Fusion optimization is one effective optimization to derive efficient parallel algorithms. It removes redundancy of compositions by fusing consecutive skeletons in the compositions. Therefore, we can develop efficient parallel algorithms from naively composed skeleton programs by applying fusions to the compositions repeatedly.

The following are three important contributions of this thesis.

The first contribution is the design of parallel skeletons based on homomorphisms, which provides skeletons with both good composability with each other and nice algebraic properties to optimize their compositions. Especially, design of skeletons for two-dimensional arrays remained as a challenging problem. Having designed the skeletons, we have be able to develop parallel programs to perform computation on various data structures, and to discuss structured parallel programming and optimization of structured parallel programs.

The second contribution is development of fusion optimizations for structured parallel programs. Parallel programs written with skeletons often suffers from the inefficiency problem caused by redundant generation of intermediate data structures among skeleton compositions. We have studied fusion optimizations of skeletons based on the fusibility of homomorphisms, which can remove intermediate data structures by fusing consecutive skeletons. Also, we have demonstrated the power of fusion optimization by a derivation of a non-trivial efficient algorithm for the maximum rectangle sum problem, giving a strategy to develop efficient skeletons programs using the fusion optimization by hand. Moreover, we have studied domain-specific fusion optimizations of skeleton programs for computation involving neighbor elements, which has been formalized as nested reductions.

The third contribution is implementation of the skeletons and the optimization mechanisms on C++ and Fortress. We have successfully implemented the designed skeletons efficiently on parallel machines, with systems for domain-independent and domain-specific optimizations. Moreover, we have shown that the optimizations can be implemented at library-level, owing to the homomorphism-based design of skeletons.

Contents

1 Introduction **1**
1.1 Background . . . 1
1.2 Homomorphism-based Structured Parallel Programming . . . 2
1.3 Contributions and Organizations of the Thesis . . . 8

2 Algebras and Homomorphisms for Data Structures **11**
2.1 Preliminaries . . . 11
2.2 Base Theory of Lists . . . 14
2.2.1 Algebra of Lists . . . 14
2.2.2 List Homomorphism . . . 15
2.3 Base Theory of Two-Dimensional Arrays . . . 16
2.3.1 Abide-tree Algebra for Two-Dimensional Arrays . . . 16
2.3.2 Abide-tree Homomorphism . . . 18
2.4 Base Theory of Trees . . . 19
2.4.1 Algebra of Trees . . . 19
2.4.2 Tree Homomorphisms . . . 20
2.5 Related Work . . . 22

3 Homomorphism-based Design of Parallel Skeletons **25**
3.1 Homomorphism-based Parallel Skeletons for Lists . . . 25
3.1.1 Definitions of Parallel Skeletons . . . 25
3.1.2 Example Programs and Variants of Skeletons . . . 27
3.2 Homomorphism-based Parallel Skeletons for Two-Dimensional Arrays 29
3.2.1 Definitions of Parallel Skeletons . . . 29
3.2.2 Example Complex Programs Described with Parallel Skeletons 39
3.3 Homomorphism-based Parallel Skeletons for Trees . . . 43

4 Fusion Optimizations of Parallel Skeletons **47**
4.1 Useful Fusion Laws for Skeletons on Lists . . . 48
4.1.1 Fusion of List Homomorphism . . . 48
4.1.2 Fusion Laws for Skeletons . . . 48
4.1.3 Accumulate-buildJ Fusion . . . 49
4.2 Useful Fusion Laws for Skeletons on Two-dimensional Arrays . . . 50

4.2.1 Fusion of Homomorphism for Two-dimensional Arrays 51
4.2.2 Fusion Laws of Skeleton Compositions 52
4.3 Developing Efficient Programs with Parallel Skeletons 53
4.3.1 Almost-homomorphism 53
4.3.2 Two Theorems for Deriving Efficient Parallel Programs 54
4.3.3 A Strategy for Deriving Efficient Parallel Programs 57
4.3.4 Deriving an Efficient Parallel Program for Maximum Rectangle Sum Problem 57
4.4 Related Work 71

5 Domain-Specific Optimization for Skeleton Programs 73
5.1 General Strategy for Domain-Specific Fusion Optimization 74
5.2 Computation Involving a Finite Number of Neighbor Elements 74
5.2.1 Target Skeleton Composition Patterns 75
5.2.2 Normal Form for the Domain 77
5.2.3 Fusion Rules for Transformation to a Normal Form 81
5.2.4 Parallel Implementation of Normal Form 85
5.2.5 Expansion of Target Programs 91
5.3 Nested Reductions: Involving an Infinite Number of Neighbor Elements 96
5.3.1 Generate-and-test Specifications: General Forms of Nested Reductions 96
5.3.2 Functions to Generate Nested Data Structures 97
5.3.3 Useful Properties on Predicates 110
5.3.4 Optimization Theorems for the Nested Reductions 112
5.4 Related Work 137

6 Implementation of Skeletons and Optimizations 139
6.1 Implementation of Parallel Skeletons on Lists 139
6.2 Implementation of Parallel Skeletons on Two-dimensional Arrays . . . 141
6.2.1 Implementation of Simple Parallel Skeletons 142
6.2.2 Implementation of Scan Skeleton 143
6.2.3 Refined Implementation for Nested Use of Skeletons 155
6.2.4 Experiment Results 157
6.3 Implementation of Fusion Optimizations 158
6.3.1 A Simple System for Domain-Independent Fusion Optimization 158
6.3.2 A Simple System for Domain-Specific Fusion Optimization . . 161
6.3.3 Experiment Results 165
6.4 Libraries with Optimization Capabilities 169
6.4.1 General Design of Growable Optimizing Libraries 170
6.4.2 Parallel Programming Language Fortress 172
6.4.3 Generators for Reductions in Fortress 176
6.4.4 Generators-of-generators: the Core of Nested Reductions . . . 180
6.4.5 Design and Implementation of the Optimizing GoG Library . 181

6.4.6 Programming with the Library . . . 188
6.4.7 Experiment Results . . . 191
6.4.8 Discussion . . . 193
6.5 Related Work . . . 197

7 Conclusion 199
7.1 Summary of the Thesis . . . 199
7.2 Future Work . . . 200

Bibliography 203

A Auxiliary Rules for Skeletons on Two-dimensional Arrays 213

B Complete Proof of Theorem 5.1 225

C Auxiliary Rules for the Proof of Theorem 5.44 231

D Dilation of Rects 235

Chapter 1

Introduction

1.1 Background

In recent years, parallel programming has become an essential task for programmers, but it is still a very difficult task for most programmers. Most of the latest computers have become parallel computers equipped with multi-core CPUs, and small scale PC clusters have become cheap to be used by individual programmers. At the same time, problems that we want to solve with computers have become bigger and heavier, and unsolvable in reasonable time and space without power of parallel computers. This situation strongly enforces parallel programming upon programmers for maximum use of these parallel computers. Parallel programming is, however, a very difficult task for most programmers, because programmers must take into account extra considerations such as distribution of tasks among processors, communication of data among processors, and synchronization of progress of the tasks.

There are two difficult problems in parallel programming: difficulty of making (designing) efficient parallel algorithms, and difficulty of implementing the parallel algorithms. The former problem is caused by lack of useful characterization of parallel algorithms and systematic methods for deriving efficient parallel algorithms. The latter problem is caused by the need to implement the extra considerations, such as distribution of tasks among processors, communication of data among processors, and synchronization of progress of the tasks, which are not required in sequential programming.

Structured parallel programming, also known as *skeletal parallel programming*, has been proposed as one promising methodology to solve the problems of parallel programming [RG02,Col02,Col89]. In this methodology, programmers build parallel programs (algorithms) by composing *skeletons*, i.e., ready-made parallel components that have efficient parallel implementation on various parallel architectures. This approach has two major advantages. The first advantage is that user programmers can compose parallel programs easily without considering difficult low-level parallelism, because low-level parallelism is completely concealed in skeletons. The other advantage is that the composed parallel programs are easy to understand and

maintain, since they are structured by skeletons. In other words, skeletal parallel programming enables us to make parallel programs in a sequential style.

Once we compose parallel algorithms described with skeletons, we can straightforwardly implement them with skeleton implementation, which is usually provided as either libraries on existing programming languages or new programming languages including skeletons as their primitive constructs. Much research has been done on skeletal parallel programming, and many skeleton libraries and systems have been proposed, which includes P3L [DPP97, Pel98], SCL [DFH+93, DGTY95], eSkel [Col04, BCHG05], MuesLi [Kuc02], QUAFF [FSCL06], and SkeTo [MIEH06].

One difficult problem in skeletal parallel programming is to provide a methodology to develop efficient parallel algorithms. Especially, optimization of skeleton compositions to improve their efficiency is very important, because it provides us with a systematic way to compose efficient parallel algorithms from naively composed skeleton programs. In general, naively composed skeleton programs suffer from inefficiency caused by redundancy of compositions, such as redundant intermediate data structures communicated between composed skeletons. This inefficiency problem is an essential problem in compositional style programming of skeletal parallel programming.

The thesis tackles the problem by homomorphism-based design of parallel skeletons that have good properties to develop efficient parallel algorithms by fusion. Fusion optimization is one effective optimization to derive efficient parallel algorithms. It removes redundancy of compositions by fusing consecutive skeletons in the compositions. Therefore, we can develop efficient parallel algorithms from naively composed skeleton programs by applying fusions to the compositions repeatedly.

1.2 Homomorphism-based Structured Parallel Programming

We will briefly review our homomorphism-based structured parallel programming.

Let's consider the following problem as an example: given a two-dimensional array, compute the maximum of sums of all rectangle areas in the array. This problem is called the maximum rectangle (or sub-array) sum problem, and important as a sort of data mining and pattern matching of two dimensional [Ben84a, Ben84b, Tak02, HIT97]. For example, for the following two-dimensional array

$$\begin{pmatrix} 3 & -1 & \mathbf{4} & \mathbf{-1} & -5 \\ 1 & -4 & \mathbf{-1} & \mathbf{5} & -3 \\ -4 & 1 & \mathbf{5} & \mathbf{3} & 1 \end{pmatrix}$$

the maximum rectangle sum is 15, which is the sum of bold numbers.

Let's start at designing the algorithm using skeletons. We are provided with the

following skeletons to manipulate two-dimensional arrays.

$$\mathsf{map}\ \ f \begin{pmatrix} x_{11} & x_{12} & \cdots & x_{1n} \\ x_{21} & x_{22} & \cdots & x_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{m1} & x_{m2} & \cdots & x_{mn} \end{pmatrix} = \begin{pmatrix} f\ x_{11} & f\ x_{12} & \cdots & f\ x_{1n} \\ f\ x_{21} & f\ x_{22} & \cdots & f^{x}_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ f\ x_{m1} & f\ x_{m2} & \cdots & f\ x_{mn} \end{pmatrix}$$

$$\mathsf{reduce}\ \ (\oplus, \otimes) \begin{pmatrix} x_{11} & x_{12} & \cdots & x_{1n} \\ x_{21} & x_{22} & \cdots & x_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{m1} & x_{m2} & \cdots & x_{mn} \end{pmatrix} = \begin{array}{c} (x_{11} \otimes x_{12} \otimes \cdots \otimes x_{1n}) \oplus \\ (x_{21} \otimes x_{22} \otimes \cdots \otimes x_{2n}) \oplus \\ \vdots \\ (x_{m1} \otimes x_{m2} \otimes \cdots \otimes x_{mn}) \end{array}$$

$$\mathsf{scan}\ \ (\oplus, \otimes) \begin{pmatrix} x_{11} & x_{12} & \cdots & x_{1n} \\ x_{21} & x_{22} & \cdots & x_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{m1} & x_{m2} & \cdots & x_{mn} \end{pmatrix} = \begin{pmatrix} y_{11} & y_{12} & \cdots & y_{1n} \\ y_{21} & y_{22} & \cdots & y_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ y_{m1} & y_{m2} & \cdots & y_{mn} \end{pmatrix}$$

$$\textbf{where}\ \ y_{ij} = \begin{array}{c} (x_{11} \otimes x_{12} \otimes \cdots \otimes x_{1j}) \oplus \\ (x_{21} \otimes x_{22} \otimes \cdots \otimes x_{2j}) \oplus \\ \vdots \\ (x_{i1} \otimes x_{i2} \otimes \cdots \otimes x_{ij}) \end{array}$$

$$\mathsf{scanr}\ \ (\oplus, \otimes) \begin{pmatrix} x_{11} & x_{12} & \cdots & x_{1n} \\ x_{21} & x_{22} & \cdots & x_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{m1} & x_{m2} & \cdots & x_{mn} \end{pmatrix} = \begin{pmatrix} z_{11} & z_{12} & \cdots & z_{1n} \\ z_{21} & z_{22} & \cdots & z_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ z_{m1} & z_{m2} & \cdots & z_{mn} \end{pmatrix}$$

$$\textbf{where}\ \ z_{ij} = \begin{array}{c} (x_{ij} \otimes \cdots \otimes x_{in}) \oplus \\ \vdots \\ (x_{mj} \otimes \cdots \otimes x_{mn}) \end{array}$$

The skeleton map applies the given function f to every element of the given two-dimensional array. The skeleton reduce collapses a two-dimensional array into a value using two associative binary operators $\oplus$ and $\otimes$. In other words, the skeleton reduce takes a *sum* of the given array with the given operators, in which the sum of vertical direction is computed with the first operator $\oplus$, and the sum of horizontal direction is computed with the second operator $\otimes$. The skeletons scan returns an array of which elements are partial results of applying reduce to the array, i.e., values of applying reduce to sub-arrays from the top-left to the bottom-right. The skeleton scanr works similarly in the reverse direction. Those skeletons are designed based on homomorphism as briefly mentioned later.

We can easily compose these skeletons to solve the problem as follows.

$$
\begin{array}{rcl}
mrs & = & max \circ \mathsf{map}\ \ sum \circ rects' \\
 & & \textbf{where} \\
 & & \quad \begin{array}{rcl}
 max & = & \mathsf{reduce}\ \ (\uparrow, \uparrow) \\
 sum & = & \mathsf{reduce}\ \ (+, +) \\
 rects' & = & \mathsf{reduce}\ \ (\ominus, \phi) \circ \mathsf{map}\ \ \mathsf{TLs} \circ \mathsf{BRs} \\
 \mathsf{TLs} & = & \mathsf{scan}\ \ (\ominus, \phi) \circ \mathsf{map}\ \ |\cdot| \\
 \mathsf{BRs} & = & \mathsf{scanr}\ \ (\ominus, \phi) \circ \mathsf{map}\ \ |\cdot|
 \end{array}
\end{array}
$$

Here, $\circ$ is a binary operator for function compositions, $\uparrow$ is a binary operator returning the bigger operand, and three operators $|\cdot|$, $\ominus$, and ϕ are used to construct two-dimensional arrays: Operator $|\cdot|$ makes a singleton array of the given element, $\ominus$ builds a bigger array from two arrays of the same width by stacking them, and ϕ combines two arrays of the same height horizontally. For example, $2 \uparrow 5 = 5$, $\begin{pmatrix}1 & 2\end{pmatrix} \ominus \begin{pmatrix}5 & 6\end{pmatrix} = \begin{pmatrix}1 & 2\\ 5 & 6\end{pmatrix}$, and $\begin{pmatrix}1 & 2\end{pmatrix} \phi \begin{pmatrix}5 & 6\end{pmatrix} = \begin{pmatrix}1 & 2 & 5 & 6\end{pmatrix}$. The function $rects'$ generates all possible rectangles of the given array, using TLs and BRs to generate top-left and bottom-right rectangles, respectively. For example, applying TLs, BRs, and $rects'$ to $\begin{pmatrix}1 & 2\\ 5 & 6\end{pmatrix}$, we get the following results.

$$
\begin{array}{rcl}
\mathsf{TLs}\ \begin{pmatrix}1 & 2\\ 5 & 6\end{pmatrix} & = & \begin{pmatrix} \begin{pmatrix}1\end{pmatrix} & \begin{pmatrix}1 & 2\end{pmatrix} \\ \begin{pmatrix}1\\5\end{pmatrix} & \begin{pmatrix}1 & 2\\ 5 & 6\end{pmatrix} \end{pmatrix} \\
\mathsf{BRs}\ \begin{pmatrix}1 & 2\\ 5 & 6\end{pmatrix} & = & \begin{pmatrix} \begin{pmatrix}1 & 2\\ 5 & 6\end{pmatrix} & \begin{pmatrix}2\\6\end{pmatrix} \\ \begin{pmatrix}5 & 6\end{pmatrix} & \begin{pmatrix}6\end{pmatrix} \end{pmatrix} \\
rects'\ \begin{pmatrix}1 & 2\\ 5 & 6\end{pmatrix} & = & \begin{pmatrix} \begin{pmatrix}1\end{pmatrix} & \begin{pmatrix}1 & 2\end{pmatrix} & \begin{pmatrix}2\end{pmatrix} \\ \begin{pmatrix}1\\5\end{pmatrix} & \begin{pmatrix}1 & 2\\ 5 & 6\end{pmatrix} & \begin{pmatrix}2\\6\end{pmatrix} \\ \begin{pmatrix}5\end{pmatrix} & \begin{pmatrix}5 & 6\end{pmatrix} & \begin{pmatrix}6\end{pmatrix} \end{pmatrix}
\end{array}
$$

The computation of $mrs = max \circ \mathsf{map}\ \ sum \circ rects'$ is very clear: enumerating all possible rectangles by $rects'$, then computing sums for all rectangles by $\mathsf{map}\ sum$, and finally returning the maximum value by max as the result of the whole computation.

Now, we have a clear naive algorithm to solve the problem. However, it seems inefficient in the sense that it needs to execute $O(n^6)$ addition operations for the input of $n \times n$ array.

The next step is to get an efficient algorithm from the naive one by fusions. To this end, let's see the basis of homomorphism-based skeletons for fusion optimization.

We will start at defining algebras of the target data structures. As an algebra to represent two-dimensional arrays, we will borrow *abide-tree* algebra from the theory of Constructive Algorithmics for sequential programming [Bir88]. The algebra consists of three constructors (operators): $|\cdot|$ (singleton), $\ominus$ (above), and ϕ (beside).

On this algebra, any two-dimensional array is constructed by the three constructors. For example, 2×2 two-dimensional array $\begin{pmatrix} 1 & 2 \\ 5 & 6 \end{pmatrix}$ is constructed by either $(|\cdot|\ 1 \,\phi\, |\cdot|\ 2) \ominus (|\cdot|\ 5 \,\phi\, |\cdot|\ 6)$ or $(|\cdot|\ 1 \ominus |\cdot|\ 5) \,\phi\, (|\cdot|\ 2 \ominus |\cdot|\ 6)$. It is worth noting that we think that these two constructions are equivalent (this equivalence is called abide-property), and use this "freedom" to capture the maximum parallelism, which will be used to get efficient parallel implementation. It is also worth noting that we use one algebra for one data structure. For example, we use another algebra called join-list for one-dimensional data arrays.

The next step to our fusion optimization is to define homomorphism of the algebra, by which our fusible skeletons are structured. Homomorphisms are recursive functions of specific forms; their computation structures and their processing data structures are closely related to each other. For two-dimensional arrays, we will have the following homomorphism h, which is parameterized with a function f and two associative binary operators $\oplus$ and $\otimes$ that have the abide-property.

$$\begin{array}{lcl} h\ (|\cdot|a) & = & f\ a \\ h\ (x \ominus y) & = & h\ x \oplus h\ y \\ h\ (x \,\phi\, y) & = & h\ x \otimes h\ y \end{array}$$

For notational convenience, we write $(\![f, \oplus, \otimes]\!)$ to denote h. Intuitively, a homomorphism $(\![f, \oplus, \otimes]\!)$ is a function to replace the constructors $|\cdot|$, $\ominus$, and ϕ in the input by f, $\oplus$, and $\otimes$, respectively. It is worth noting that homomorphism has efficient (balanced) parallel implementation by naive divide-and-conquer parallel computation, in which the freedom of the algebra plays an important role for balancing.

Actually, the skeletons shown above are homomorphisms. For example, $\mathsf{map}\ f = (\![|\cdot| \circ f, \ominus, \phi]\!)$, and $\mathsf{reduce}\ (\oplus, \otimes) = (\![id, \ominus, \phi]\!)$, where id is the identity function. Also, scan and scanr are special cases of homomorphisms, but we omit them here because they are too complicated to be shown here.

The reason why we structure our skeletons by homomorphism is that homomorphism has the following nice property for fusion optimization. Given a function g and homomorphism $(\![f, \oplus, \otimes]\!)$, the composition $g \circ (\![f, \oplus, \otimes]\!)$ is fused into another homomorphism $(\![g \circ f, \odot, \ominus]\!)$, when the following equations hold: $g\ (x \oplus y) = g\ x \odot g\ y$ and $g\ (x \otimes y) = g\ x \ominus g\ y$. This property is very useful, because we can repeatedly fuse functions into a homomorphism.

Our skeletons have good fusibility based on the fusibility of homomorphism. For example, we can fuse two maps into one map as follows.

$$\begin{array}{ll} & \mathsf{map}\ f \circ \mathsf{map}\ g \\ = & \{\ \mathsf{map}\ g \text{ is homomorphism } (\![|\cdot| \circ g, \ominus, \phi]\!)\ \} \\ & \mathsf{map}\ f \circ (\![|\cdot| \circ g, \ominus, \phi]\!) \\ = & \left\{ \text{homomorphism fusion with } \left\{ \begin{array}{l} \mathsf{map}\ f\ (x \ominus y) = \mathsf{map}\ f\ x \ominus \mathsf{map}\ f\ y \\ \mathsf{map}\ f\ (x \,\phi\, y) = \mathsf{map}\ f\ x \,\phi\, \mathsf{map}\ f\ y \end{array} \right. \right\} \\ & (\![\mathsf{map}\ f \circ |\cdot| \circ g, \ominus, \phi]\!) \end{array}$$

$$
\begin{array}{ll}
= & \{ \text{ simplification } (\mathsf{map}\ \ f \circ |\cdot| \circ g = |\cdot| \circ f \circ g) \ \} \\
 & (\![|\cdot| \circ f \circ g, \ominus, \phi]\!) \\
= & \{ \ \mathsf{map} \text{ is homomorphism } \} \\
 & \mathsf{map}\ \ (f \circ g)
\end{array}
$$

Here, each use of braces ({ }) shows a reason of an equation.

Now, let's return to our example. Using fusions of skeletons repeatedly, we can get the following optimized algorithm for *mrs*. Here, the complex composition of skeletons is fused into only one homomorphism, and it is split into two skeletons reduce and map. We will omit the details here (please see Section 4.3.4 for details).

$$
\begin{array}{ll}
 & mrs \\
= & \{ \text{ the naive composition of skeletons } \} \\
 & \mathsf{reduce}\ \ (\uparrow, \uparrow) \circ \mathsf{map}\ \ (\mathsf{reduce}\ \ (+, +)) \circ \mathsf{reduce}\ \ (\ominus, \phi) \circ rects' \\
 & \quad\quad \textbf{where}\ \ rects' = \mathsf{map}\ \ (\mathsf{scan}\ \ (\ominus, \phi) \circ \mathsf{map}\ \ |\cdot|) \circ (\mathsf{scanr}\ \ (\ominus, \phi) \circ \mathsf{map}\ \ |\cdot|) \\
= & \{ \text{ repeated application of fusions (details are omitted) } \} \\
 & \pi_1 \circ (\![f_{mrs}, \oplus_{mrs}, \otimes_{mrs}]\!) \\
= & \{ \text{ homomorphism is composition of } \mathsf{map} \text{ and } \mathsf{reduce} \ \} \\
 & \pi_1 \circ \mathsf{map}\ \ f_{mrs} \circ \mathsf{reduce}\ \ (\oplus_{mrs}, \otimes_{mrs})
\end{array}
$$

Here, π_1 is a function to extract the first component of a tuple. The cost of the derived algorithm (optimized program) is $O(n^3)$, while the naive one is $O(n^6)$; we have succeeded in deriving the efficient parallel algorithm by eliminating redundancy of the algorithm using fusions.

Now, we have become able to develop efficient parallel algorithms with skeletons. The rest of our concern is how to implement the algorithm as an actual parallel program.

Implementation of algorithms described with skeletons is straightforward, when we are provided with parallel implementation of the skeletons used, which is usually provided as either libraries on existing programming languages or new languages that include skeletons as their primitive constructs. For example, we have made a library on C++ to provide implementation of our designed skeletons. Using this library, we can implement the optimized program of *mrs* as follows.

```
1 int mrs(const matrix<int> &mat)
2 {
3     dist_matrix<int> dmat(&mat);
4     dist_matrix<tuple> *mat2 = map(F_MRS, &dmat);
5     tuple *res = reduce(OplusMRS, OtimesMRS, mat2);
6     return res->first;
7 }
```

Here, `dist_matrix` is a data structure to be manipulated by skeletons, and `F_MRS`, `OplusMRS`, and `OtimesMRS` are function objects that implement the function and the operators in the derived algorithm (these are sequential programs).

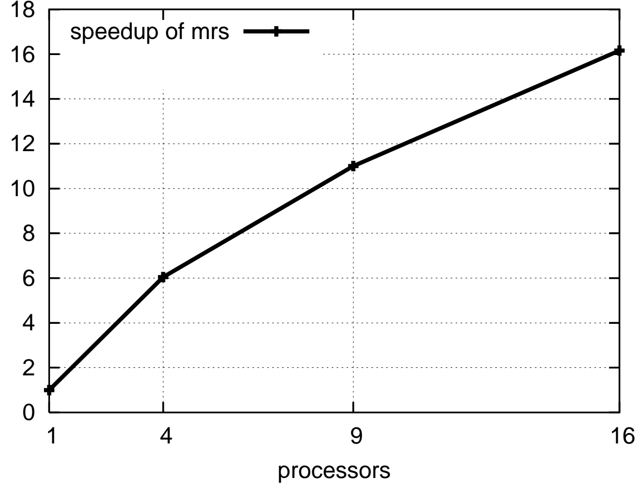


Figure 1.1. Speedup of the skeleton program for the maximum rectangle sum.

Since skeletons are implemented as components of parallel programs, programs written with the provided skeletons work as parallel programs. For example, the above program achieves good speed up when we use multiple processors, which is shown in Figure 1.1.

Finally, we mention about alternative notation for parallel programming based on our skeletons. We can provide alternative notation for parallel programming when we have mapping from the notation to our skeletons. For example, we can use comprehension notation [BS90, BHS+94, Ble96, BG96, CK00, CK01, CKLP01, LCK06, CLJ+07, FRR+07] as an alternative, in which we can describe computation by two parts: one is an expression (a function) to specify element-wise computation applied to each element in a data structure, and the other is an operator to specify reduction computation on the results of the element-wise computation. For example, we can describe computation of a sum of squares in comprehension notation as follows.

$$\sum \langle a^2 \mid a \in x \rangle$$

Here, x is the data structure such as arrays, the expression a^2 specifies how to compute new elements from elements a in x, and $\sum$ specifies how to *sum* them up (i.e., taking a sum with the usual plus operator). Intuitively, when x is a two-dimensional array, the relation between the notation and our skeleton is given as follows: $\bigoplus \langle f\ a \mid a \in x \rangle \Leftrightarrow \mathsf{reduce}\ (\oplus, \oplus)\ (\mathsf{map}\ f\ x)$. This notation is very useful when we need nested use of skeletons. For example, the program for the maximum rectangle sum problem is clearly written with comprehension notation as follows.

$$\uparrow \langle \sum \langle a \mid a \in r \rangle \mid r \in \mathit{rects}'\ x \rangle$$

Or, it can be abbreviated as follows.

$$\uparrow \langle \sum r \mid r \in \mathit{rects}'\ x \rangle$$

Here, we assume that we are provided with the function *rects*′, since this function is often used for programming on two-dimensional arrays.

Also, high-level optimization like that for *mrs* shown above can be implemented by an automatic optimization mechanism, because such optimization can be summarized as rewriting rules on skeleton compositions. Although such rewriting rules may have conditions on parameter functions (operators), such conditions are usually represented by whether the parameters have certain properties or not. Therefore, an optimization mechanism can check such conditions and apply correct optimization, once we annotate functions (operators) with possession of properties.

We have designed and implemented comprehension notation with a mechanism for the high-level optimization, in which we can enjoy concise notation and the benefit of high-level optimization for free.

1.3 Contributions and Organizations of the Thesis

The body of the thesis consists of three parts. The first part (Chapters 2 and 3) addresses principles of homomorphism-based parallel skeletons for various data structures. The second part (Chapters 4 and 5) addresses principles of optimizations on skeleton programs. The last part (Chapter 6) addresses implementations of the skeletons and the optimizations.

In Chapter 2, we will introduce algebras for various data structures and homomorphisms on the algebras, which are the basis of our homomorphism-based parallel skeletons. Since homomorphisms have no gap between the structure of their computation and the structure of their processing data, they have good composability with each other and good optimizability for their compositions. In the thesis, we will deal with the following data structures: lists, tow-dimensional arrays, and trees. Especially, we will propose a *novel use of the abide-tree representation of two-dimensional arrays* [Bir88], of which importance has not been fully recognized in parallel programming community.

In Chapter 3, we will design a set of parallel skeletons based on the homomorphisms for lists, two-dimensional arrays, and trees. Structured by homomorphisms, our designed skeletons have good composability with each other owing to that of homomorphisms, which will be shown by some example programs written with skeletons. Especially, we will tackle the following two challenging problem: the *homomorphism-based design of parallel skeletons on two-dimensional arrays*, and *development of parallel programs with these skeletons*, which includes some parallel matrix operations.

In Chapter 4, we will proceed to theories of optimizations of skeleton programs. Parallel programs written with skeletons often suffers from the inefficiency problem caused by redundant generation of intermediate data structures among skeleton compositions. This problem is essential in skeletal parallel programming, because it supports the compositional-style programming; the skeletons are designed

so that we can describe many programs by composing a small set of skeletons. Therefore, we will study fusion optimizations of skeletons based on the fusibility of homomorphisms, which can remove intermediate data structures by fusing consecutive skeletons. Also, we will demonstrate the power of fusion optimization by a derivation of a non-trivial efficient algorithm for the maximum rectangle sum problem [Ben84a, Ben84b, Tak02, HIT97], giving a strategy to develop efficient skeletons programs using the fusion optimization by hand. Especially, the development of *fusion optimization for skeletons on two-dimensional arrays* and the *derivation of efficient programs for two-dimensional arrays* are both big contributions of the thesis.

In Chapter 5, we will study domain-specific optimizations of skeleton programs, to solve some problems of domain-independent fusion optimizations caused by their generality. We will concentrate on *domain-specific fusion optimization of skeleton programs for computation involving neighbor elements*, which can be categorized into two types: that involving a finite number of neighbor elements, such as filtering of sequences and images, the finite difference method, and some matrix-vector operations; and that involving an infinite number of neighbor elements, such as queries of interesting segments on lists and rectangles (sub-arrays) on two-dimensional arrays. We will develop domain-specific fusion rules for the former type, with proposing a new strategy for developing domain-specific fusion optimization of skeleton programs. Then, for the latter type, we will formalize it as as nested reductions, and develop various theorems to provide efficient algorithms to nested reductions by fusion. Especially, *characterization of domain-specific fusion rules*, and *optimization theorems for nested reductions with filtering or two-dimensional arrays* are new results of the thesis.

In Chapter 6, we will report implementations of the designed skeletons and the optimizations over skeleton programs. Especially, we will propose *efficient implementation of skeletons for two-dimensional arrays*, *a small system for fusion optimizations*, and *design and implementation of libraries with optimization capabilities for structured parallel programs* in Fortress [ACH+08]. Experimental results with these implementation show that the developed theories are actually effective for parallel programming, and guarantee the success of the proposed skeletal programming framework.

Chapter 2

Algebras and Homomorphisms for Data Structures

In this chapter, we will introduce algebras and their homomorphisms, which will be the basis of homomorphism-based skeletal parallel programming. Homomorphisms are recursive functions of specific forms; their computation structures and their processing data structures are closely related to each other. This closeness brings good composability of homomorphisms, and good algebraic laws for optimizing the compositions, which has been shown in the theory of Constructive Algorithmics [Bir88, Ski94, BdM96]. Therefore, using the algebras and homomorphisms as the basis of the design, we will be able to design well-composable, well-optimizable skeletons, which will be seen in the next chapter.

First, we will introduce notation, operators, and their properties used throughout the thesis. Then, we will introduce algebras and their homomorphisms for lists [Bir88, Ski94, BdM96], two-dimensional arrays [Bir88, Mil94], and trees [Ski96, Mat07].

2.1 Preliminaries

Notation

We will use the following notation throughout the thesis, unless otherwise noted.

Notation in the thesis follows that of Haskell [Jon02, Bir98], a pure functional language that can describe both algorithms and algorithmic transformation concisely.

Function application is denoted by a space and the argument may be written without brackets. Thus, $f\ a$ means $f(a)$ in ordinary notation.

Functions are curried; a function takes one argument and returns a function or a value.

The function application associates to the left. Thus, $f\ a\ b$ means $(f\ a)\ b$.

The function application binds more strongly than any other operator. Thus, $f\ a \otimes b$ means $(f\ a) \otimes b$ but not $f\ (a \otimes b)$.

Function composition is denoted by $\circ$, so $(f \circ g)\, x = f\, (g\, x)$ from its definition.

Binary operators can be used as functions by sectioning as follows: $a \oplus b = (a\oplus)\, b = (\oplus b)\, a = (\oplus)\, a\, b$.

For any binary operator $\otimes$, operator $\tilde{\otimes}$ applies the operator after swapping the arguments. Thus, $a \,\tilde{\otimes}\, b = b \otimes a$.

For any binary operator $\oplus$, its identity is denoted by $\imath_{\oplus}$.

Pairs are Cartesian products of plural data, written like (x, y).

A projection function π_1 extracts the first component of a pair. Thus, $\pi_1\ (x, y) = x$. These can be extended to the case of arbitrary number of elements.

Useful Binary Operators and Properties of Operators

We will introduce some useful binary operators used in the thesis. After that, we will define some properties on operators.

The maximum and minimum operators are denoted by arrows $\uparrow$ and $\downarrow$, respectively, as follows.

$$\begin{array}{rcll} a \uparrow b & = & a & \text{if } a \geq b \\ & = & b & \text{otherwise} \\ a \downarrow b & = & a & \text{if } a \leq b \\ & = & b & \text{otherwise} \end{array}$$

Given a function f, the maximum and minimum operation with respect to the function are denoted by $\uparrow_f$ and $\downarrow_f$, respectively, as defined below.

$$\begin{array}{rcll} a \uparrow_f b & = & a & \text{if } f\ a \geq f\ b \\ & = & b & \text{otherwise} \\ a \downarrow_f b & = & a & \text{if } f\ a \leq f\ b \\ & = & b & \text{otherwise} \end{array}$$

Given two functions f and g, binary operator $\times$ makes function $f \times g$ that applies the given functions respectively to the components of the argument pair (x, y). Thus, $(f \times g)\, (x, y) = (f\, x, g\, y)$.

Given two functions f and g, binary operator $\vartriangle$ makes function $f \vartriangle g$ that applies the given functions separately to an element, and returns the pair of the results. Thus, $(f \vartriangle g)\, a = (f\, a, g\, a)$.

Two binary operators $\ll$ and $\gg$ are defined by $a \ll b = a$ and $a \gg b = b$, respectively.

Now, we will define some properties of operators, which have certain importance in developing parallel programs.

Associativity is the basis of balanced parallel computation since it guarantees correctness of re-balancing of computation tasks.

Definition 2.1 (Associativity)**.** Binary operator $\oplus$ is said to be associative if the following equation holds for all a, b and c.

$$a \oplus (b \oplus c) = (a \oplus b) \oplus c$$

$\square$

Distributivity plays an important role in deriving efficient implementations. Basically, distributivity guarantees efficient reuse of partial results.

Definition 2.2 (Left-distributivity)**.** Binary operator $\otimes$ is said to be left-distributive over $\oplus$ if the following equation holds for all a, b and c.

$$a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$$

$\square$

Definition 2.3 (Right-distributivity)**.** Binary operator $\otimes$ is said to be right-distributive over $\oplus$ if the following equation holds for all a, b and c.

$$(a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c)$$

$\square$

Definition 2.4 (Distributivity)**.** Binary operator $\otimes$ is said to be distributive over $\oplus$ if $\otimes$ is left- and right-distributive over $\oplus$, i.e. the following equations hold for all a, b and c.

$$a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$$
$$(a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c)$$

$\square$

Commutativity is also used to change the order of computation to achieve good efficiency.

Definition 2.5 (Commutativity)**.** Binary operator $\oplus$ is said to be commutative if the following equation holds for all a and b.

$$a \oplus b = b \oplus a$$

$\square$

For example, well-known binary operators $+$ (plus), $\times$ (times), $\uparrow$ (maximum), and $\downarrow$ (minimum) are all associative and commutative. Moreover, $\times$ distributes over $+$, and $+$ distributes over $\uparrow$ or $\downarrow$. As seen later, concatenation operator $+\!\!+$ of lists is associative, but not commutative. Also, function $\mathsf{map}\, f$ is left-distributive over $+\!\!+$, since $\mathsf{map}\, f\, (x +\!\!+ y) = (\mathsf{map}\, f\, x) +\!\!+ (\mathsf{map}\, f\, y)$. Distributivity of $\mathsf{map}\, f$ is often used in the following derivations of efficient implementations.

2.2 Base Theory of Lists

In this section, we will introduce the base theory of lists. We will start at defining algebra of lists. Then, we will introduce homomorphism on the algebra as basic parallel computation, in which no gap exists between the structure of computation and the structure of data. This close relationship results in good composability and good optimizability.

2.2.1 Algebra of Lists

To represent lists, i.e., sequences of elements, we will use the following algebra of lists [Bir88].

Definition 2.6 (List algebra[1])**.** The algebra of lists is defined with two constructors: $+\!\!+$ (concatenation) and $[\cdot]$ (singleton).

$$\begin{array}{lcl} \textbf{data}\ \mathit{List}\ \alpha & = & (\mathit{List}\ \alpha) +\!\!+ (\mathit{List}\ \alpha) \\ & | & [\cdot]\ \alpha \end{array}$$

□

Here, $[\cdot]\ a$, or abbreviated as $[a]$, represents a singleton list of element a; and for any lists x and y, $x +\!\!+ y$ represents the concatenated list consisting of elements of x followed by those of y.

For example, a list of three elements a_1, a_2, and a_3 is represented by $[a_1] +\!\!+ [a_2] +\!\!+ [a_3]$, and may be abbreviated as $[a_1, a_2, a_3]$. Similarly, for a list of arbitrary number of elements $a_1, a_2, \ldots, a_n$, we will use the intuitive notation $[a_1, a_2, \ldots, a_n]$. We also use a notation $a : x = [a] +\!\!+ x$ to show the head of a list.

It is worth noting that the concatenation $+\!\!+$ has the associativity; for any lists x, y, and z, two concatenations $x +\!\!+ (y +\!\!+ z)$ and $(x +\!\!+ y) +\!\!+ z$ result in the same list.

The associativity of the concatenation $+\!\!+$ is the most important property of the algebra, since it is the basis of good parallelism of parallel skeletons. This will be explained after introducing the homomorphism.

Here, we will introduce a special value $[\,]$ to represent an empty list, which is sometimes useful to be used as the default value or the initial value in auxiliary functions, although we consider computations on non-empty lists only. We assume that the application of a function to $[\,]$ results in $[\,]$ when the function returns a list, and also assume that it results in the identity of the binary operator when the function returns a value computed with the binary operator, which is the natural assumption since $[\,]$ is the identity of $+\!\!+$; for any list x, either of $x +\!\!+ [\,]$ and $[\,] +\!\!+ x$ equals x.

[1] Actually, there are several alternative algebras to represent lists [Bir88]. We have chosen the algebra because it has good properties to be used as the basis of parallel skeletons.

2.2.2 List Homomorphism

From the theory of Constructive Algorithmics [BdM96], it follows that each constructively built-up data structure (i.e., algebraic data structure) is equipped with a powerful computation pattern called homomorphism. The homomorphism of the list algebra is defined as follows.

Definition 2.7 (List homomorphism)**.** Given a function f and an associative binary operator $\oplus$, a list homomorphism h is a function defined by the following equations.

$$\begin{array}{lcl} h\ (x +\!\!+\, y) & = & h\ x \oplus h\ y \\ h\ [a] & = & f\ a \end{array}$$

For notational convenience, we write $([f, \oplus])$ to denote h. When it is clear from the context, we just call $([f, \oplus])$ homomorphism. □

Intuitively, a homomorphism $([f, \oplus])$ is a function to replace the constructors $[\cdot]$ and $+\!\!+$ of the input list with f and $\oplus$, respectively. For example, applying $([f, \oplus])$ to a list $[a_1, a_2, a_3, a_4]$, we get the following result.

$$\begin{array}{lcl} ([f, \oplus])\ [a_1, a_2, a_3, a_4] & = & ([f, \oplus])\ ([\cdot]\ a_1 +\!\!+ [\cdot]\ a_2 +\!\!+ [\cdot]\ a_3 +\!\!+ [\cdot]\ a_4) \\ & = & f\ a_1 \oplus f\ a_2 \oplus f\ a_3 \oplus f\ a_4 \end{array}$$

Many programs are written in terms of homomorphisms. For example, a summation of elements of a list,

$$sum\ [1, 2, 3, 4] = 1 + 2 + 3 + 4,$$

is clearly written with a homomorphism as follows:

$$sum = ([id, +])\ .$$

It applies the identity function id to each of the elements, and then sums up them by the operator $+$:

$$\begin{aligned} ([id, +])\ [1, 2, 3, 4] &= id\ 1 + id\ 2 + id\ 3 + id\ 4 \\ &= 1 + 2 + 3 + 4 \\ &= sum\ [1, 2, 3, 4]\ . \end{aligned}$$

Functions to calculate the length of a list and to double elements of a list are also written as homomorphism:

$$\begin{aligned} length &= ([one, +]) \quad \textbf{where}\ \ one\ a = 1\ , \\ double &= ([[\cdot] \circ dbl, +\!\!+]) \quad \textbf{where}\ \ dbl\ a = 2 \times a\ . \end{aligned}$$

Note that $\oplus$ of homomorphism $([f, \oplus])$ must be associative, inheriting the associativity of $+\!\!+$. Otherwise the result is not well-defined, because the recursion structure of the computation of the homomorphism can vary according to the freedom of division (construction) of the argument list.

Balanced Divide-and-conquer Parallel Computation of Homomorphism

We will briefly see the good parallelism of the list algebra and the list homomorphism.

The associativity of the algebra brings good divide-and-conquer parallelism in computation of homomorphism. Let's consider to apply homomorphism $([f, \oplus])$ to list $[a_1, a_2, \ldots, a_n]$. To compute $([f, \oplus])\ [a_1, a_2, \ldots, a_n]$ we can divide the list at the middle point like $[a_1, a_2, \ldots, a_n] = [a_1, a_2, \ldots, a_{\lceil n/2 \rceil}] +\!\!+ [a_{\lceil n/2 \rceil+1}, \ldots, a_n]$, since the concatenation is associative. This division results in the following computation.

$$
\begin{array}{ll}
 & ([f, \oplus])\ [a_1, a_2, \ldots, a_n] \\
= & \{ \text{ Diving the list at the middle point } \} \\
 & ([f, \oplus])\ ([a_1, a_2, \ldots, a_{\lceil n/2 \rceil}] +\!\!+ [a_{\lceil n/2 \rceil+1}, \ldots, a_n]) \\
= & \{ \text{ Definition of homomorphism } \} \\
 & (([f, \oplus])\ [a_1, a_2, \ldots, a_{\lceil n/2 \rceil}]) \oplus (([f, \oplus])\ [a_{\lceil n/2 \rceil+1}, \ldots, a_n])
\end{array}
$$

The above equation means that we can compute $([f, \oplus])\ [a_1, a_2, \ldots, a_n]$ in two steps: (1) computing two sub-results $([f, \oplus])\ [a_1, a_2, \ldots, a_{\lceil n/2 \rceil}]$ and $([f, \oplus])\ [a_{\lceil n/2 \rceil+1}, \ldots, a_n]$, and then (2) combining those results with the operator $\oplus$. Also, we can compute the sub-results in parallel, since their computations are independent. Thus, repeatedly applying the division of computation to get the sub-results, we can compute the result of homomorphism by a naive divide-and-conquer parallel computation. Therefore, we can use homomorphism as the basic parallel computation pattern.

2.3 Base Theory of Two-Dimensional Arrays

We will build the base theory of two-dimensional arrays based on Constructive Algorithmics [Bir88, Ski94, BdM96]. First, we will introduce an algebra of two-dimensional arrays. Then, we will define its homomorphism, which is the basis of parallel computations on two-dimensional arrays.

2.3.1 Abide-tree Algebra for Two-Dimensional Arrays

To represent two-dimensional arrays, we use the following *abide-tree* algebra, which are built up by three constructors $|\cdot|$ (singleton), $⦵$ (above) and $⦶$ (beside) [Bir88]. The abide-tree representation of two-dimensional arrays not only inherits the advantages of tree representations of matrices where recursive blocked algorithms can be defined to achieve better performance [EGJK04, FW03, Wis99], but also supports systematic development of parallel programs and architecture independent implementation.

Definition 2.8 (Abide-tree algebra)**.** The abide-tree algebra for two-dimensional

arrays is defined with three constructors: $|\cdot|$ (singleton), $⦵$ (above), and $⦶$ (beside).

$$\begin{array}{lcl} \textbf{data}\ \mathit{AbideTree}\,\alpha & = & |\cdot|\ \alpha \\ & | & (\mathit{AbideTree}\,\alpha) ⦵ (\mathit{AbideTree}\,\alpha) \\ & | & (\mathit{AbideTree}\,\alpha) ⦶ (\mathit{AbideTree}\,\alpha) \end{array}$$

□

Here, $|\cdot|\ a$, or abbreviated as $|a|$, means a singleton array of a, i.e. a two-dimensional array with a single element a. We define the function the to extract the element from a singleton array, i.e. $\mathsf{the}\ |a| = a$. For two-dimensional arrays x and y of the same width, $x ⦵ y$ represents an array made by putting x on y. Similarly, for two-dimensional arrays x and y of the same height, $x ⦶ y$ represents an array made by putting x on the left of y. Moreover, $⦵$ and $⦶$ are associative binary operators and have the following *abide* (a coined term from above and beside) property.

Definition 2.9 (Abide Property)**.** Two binary operators $\oplus$ and $\otimes$ are said to have the abide property or to be abiding, if the following equation holds for any x, y, u, and y in the interested domain:

$$(x \otimes u) \oplus (y \otimes v) = (x \oplus y) \otimes (u \oplus v)\ .$$

□

In the rest of the thesis, we will assume no inconsistency in height or width when $⦶$ and $⦵$ are used.

Note that one two-dimensional array may be represented by more than one abide-trees, but these abide-trees are equivalent because of the abide property of $⦵$ and $⦶$. For example, we can express the 2×2 two-dimensional array

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

by the following two equivalent abide-trees.

$$\begin{array}{c} (|1| ⦶ |2|) ⦵ (|3| ⦶ |4|) \\ (|1| ⦵ |3|) ⦶ (|2| ⦵ |4|) \end{array}$$

This is in sharp contrast to other representations of matrices, such as quadtree representation [FW03], which do not allow such freedom. This freedom is important in our framework. First, this freedom allows easy re-balancing of the tree in computations such as divide-and-conquer computations on abide-trees, as associativity of $+\!\!+$ does for computation on lists. Therefore, the freedom is important for efficient balanced parallel computation. Next, the freedom is useful to describe and derive efficient programs for various architectures. In our framework, a program construction may have two phases. First, we make a general program that is architecture

independent. Then, we derive a good program that may be architecture dependent or may have restricted order of access. In this approach, restrictive representation prevents us form describing and deriving efficient programs. For example, a list of lists representation, which restricts the access order to outer dimension to inner dimension, does not allow us to describe and to derive efficient blocked algorithms. Thus, we start with a program using the abide-tree representation that does not impose restrictions on the access order, then we transform it to a good program that may have the restricted order of accesses.

2.3.2 Abide-tree Homomorphism

Next, we will introduce homomorphism on the abide-trees, which will be the basis of parallel computations on two-dimensional data structures. From the theory of Constructive Algorithmics [BdM96], it follows that each constructively built-up data structure (i.e., algebraic data structure) is equipped with a powerful computation pattern called homomorphism.

Definition 2.10 (Abide-tree homomorphism)**.** Given a function f and two associative binary operators $\oplus$ and $\otimes$ that have the abide property, an abide-tree homomorphism h is a function defined by the following equations.

$$
\begin{array}{lcl}
h\ |a| & = & f\ a \\
h\ (x \ominus y) & = & h\ x \oplus h\ y \\
h\ (x \phi y) & = & h\ x \otimes h\ y
\end{array}
$$

For notational convenience, we write $(\![f, \oplus, \otimes]\!)$ to denote h. When it is clear from the context, we just call $(\![f, \oplus, \otimes]\!)$ homomorphism. □

Intuitively, a homomorphism $(\![f, \oplus, \otimes]\!)$ is a function to replace the constructors $|\cdot|$, $\ominus$, and ϕ in an input abide-tree by f, $\oplus$, and $\otimes$, respectively. We will see in Section 3.2 that many algorithms on two-dimensional arrays can be concisely specified by homomorphisms.

Note that $\oplus$ and $\otimes$ of homomorphism $(\![f, \oplus, \otimes]\!)$ must be associative and have the abide property, inheriting the properties of $\ominus$ and ϕ.

We can describe many functions with homomorphism. For example, a summation of elements of an array,

$$
sum \begin{pmatrix} 1 & 5 \\ 7 & 3 \end{pmatrix} = 1 + 5 + 7 + 3,
$$

is clearly written with a homomorphism as follows:

$$
sum = (\![id, +, +]\!)\ .
$$

It applies the identity function id to each of the elements, and then sums up them by the operator $+$:

$$
\begin{aligned}
(\!| id, +, + |\!) \begin{pmatrix} 1 & 5 \\ 7 & 3 \end{pmatrix} &= (\!| id, +, + |\!) \begin{pmatrix} 1 & 5 \end{pmatrix} + (\!| id, +, + |\!) \begin{pmatrix} 7 & 3 \end{pmatrix} \\
&= (id\ 1 + id\ 5) + (id\ 7 + id\ 3) \\
&= 1 + 5 + 7 + 3 \\
&= sum \begin{pmatrix} 1 & 5 \\ 7 & 3 \end{pmatrix} .
\end{aligned}
$$

In the above computation, we assumed that the input array is constructed as $(|1| \phi |5|) \ominus (|7| \phi |3|)$. It is worth noting that we can get the same result when the array is constructed in the alternative way as $(|1| \ominus |7|) \phi (|5| \ominus |3|)$, because the plus operator $+$ has the abide property for itself. Generally, a commutative, associative binary operator satisfies the condition of the abide property for itself. Functions to calculate the width of an array and to double elements of an array are also written as homomorphism:

$$
\begin{aligned}
width &= (\!| one, \ll, + |\!) \quad \textbf{where}\ \ one\ a = 1 \ , \\
double &= (\!| [\cdot] \circ dbl, \ominus, \phi |\!) \quad \textbf{where}\ \ dbl\ a = 2 \times a \ .
\end{aligned}
$$

Because of the flexibility of the abide-tree representation, a homomorphism $(\!| f, \oplus, \otimes |\!)$ can be implemented *efficiently* in parallel, which will be shown in Chapter 6. Therefore, we will design our parallel skeletons based on the homomorphisms.

2.4 Base Theory of Trees

This section briefly reviews the base theory of trees. Here, we will deal with only binary trees, in which internal nodes have exactly two children. The theory of another type of trees, i.e., rose trees (a term coined by Meertens [Mee88]), of which internal nodes have an arbitrary number of children, can be found in Matsuzaki's thesis [Mat07].

First, we will introduce an algebra of binary trees. Then, we will proceed to tree homomorphism.

2.4.1 Algebra of Trees

The algebra of binary trees is given as follows.

Definition 2.11 (Algebra of binary trees)**.** The algebra of binary trees is defined with two constructors BLeaf for leaves and BNode for internal nodes as follows.

$$
\begin{aligned}
\textbf{data}\ \ \mathsf{BTree}\ \alpha\ \beta \ &= \ \mathsf{BLeaf}\ \alpha \\
& \mathsf{BNode}\ (\mathsf{BTree}\ \alpha\ \beta)\ \beta\ (\mathsf{BTree}\ \alpha\ \beta)
\end{aligned}
$$

□

The BLeaf is the constructor for leaves, and holds an element of type α. The BNode is the constructor for internal nodes and takes three parameters, the left subtree, the value of the node, and the right subtree, in this order. The value of the node has type β.

Although the above algebra has no freedom for parallelism, we can construct another equivalent algebra with certain freedom for parallelism, which has been formalized as *tree associativity* [Mat07]. However, the algebra and tree associativity are too complicated to show in this thesis. Thus, we will use the above simple algebra to define tree skeletons, although we will show condition for efficient parallel computation translated from the tree associativity.

We introduce two functions for manipulating binary trees. Function $root_{\mathrm{b}}$ returns the value of the root node, and function $setroot_{\mathrm{b}}$ takes a binary tree and a value, and replaces the value of the root node with the input value. Note that we use _ to denote a *don't-care* value.

$$
\begin{array}{lcl}
root_{\mathrm{b}}\ (\mathsf{BLeaf}\ a) & = & a \\
root_{\mathrm{b}}\ (\mathsf{BNode}\ _\ b\ _) & = & b
\end{array}
$$

$$
\begin{array}{l}
setroot_{\mathrm{b}}\ (\mathsf{BLeaf}\ _)\ a' = \mathsf{BLeaf}\ a' \\
setroot_{\mathrm{b}}\ (\mathsf{BNode}\ l\ _\ r)\ b' = \mathsf{BNode}\ l\ b'\ r
\end{array}
$$

It is worth noting that there has been research on parallel skeletons for rose trees (a term coined by Meertens [Mee88]) of which internal nodes have an arbitrary number of children. However, we will omit the results here, because it has been developed based on the theory of binary trees. Please refer to Matsuzaki's thesis [Mat07] for the details.

2.4.2 Tree Homomorphisms

For binary trees, we can define the following *binary-tree homomorphism* (or *tree homomorphism* for short) [Ski94, Ski96].

Definition 2.12 (Tree Homomorphism)**.** Let k_l and k_n be given functions. A function h is called *tree homomorphism* (or simply *homomorphism*), if it is defined in the following recursive form.

$$
\begin{array}{lcl}
h\ (\mathsf{BLeaf}\ a) & = & k_l\ a \\
h\ (\mathsf{BNode}\ l\ b\ r) & = & k_n\ (h\ l)\ b\ (h\ r)
\end{array}
$$

We may denote the tree homomorphism above as $h = (\![k_l, k_n]\!)_{\mathrm{b}}$. □

We can specify many tree manipulations in the form of tree homomorphism. An example of tree homomorphism is function $height_{\mathrm{b}}$ that computes the height of a binary tree.

$$
\begin{array}{lcl}
height_{\mathrm{b}}\ (\mathsf{BLeaf}\ a) & = & 1 \\
height_{\mathrm{b}}\ (\mathsf{BNode}\ l\ b\ r) & = & 1 + (height_{\mathrm{b}}\ l \uparrow height_{\mathrm{b}}\ r)
\end{array}
$$

This function is indeed a tree homomorphism $height_{\mathrm{b}} = (\!|height_l, height_n|\!)_{\mathrm{b}}$ with the two parameter functions defined as follows:

$$\begin{array}{lcl} height_l\ a & = & 1 \\ height_n\ l\ b\ r & = & 1 + (l \uparrow r)\ . \end{array}$$

Tree homomorphisms that return a basic value, like the $height_{\mathrm{b}}$ function, are often called *tree reductions*.

The following two computational patterns called *tree accumulations* return trees instead of basic values. These two tree accumulations are in fact tree homomorphisms [Mat07] as stated later.

Definition 2.13 (Upwards accumulation)**.** Let k_l and k_n be given functions. A function h^u is called *upwards accumulation*, if it is defined in the following recursive form.

$$\begin{array}{lcl} h_u\ (\mathsf{BLeaf}\ a) & = & \mathsf{BLeaf}\ (k_l\ a) \\ h_u\ (\mathsf{BNode}\ b\ l\ r) & = & \mathbf{let}\ l' = h_u\ l \\ & & \qquad r' = h_u\ r \\ & & \mathbf{in}\ \mathsf{BNode}\ (k_n\ b\ (root_{\mathrm{b}} l')\ (root_{\mathrm{b}}\ r'))\ l'\ r' \end{array}$$

□

Definition 2.14 (Downwards accumulation)**.** Let g_l and g_r be given functions. A function h^d is called *downwards accumulation*, if it is defined in the following recursive form with additional parameter c.

$$\begin{array}{lcl} h^d\ c\ (\mathsf{BLeaf}\ a) & = & \mathsf{BLeaf}\ c \\ h^d\ c\ (\mathsf{BNode}\ l\ b\ r) & = & \mathsf{BNode}\ (h^d\ (g_l\ c\ b)\ l)\ c\ (h^d\ (g_r\ c\ b)\ r) \end{array}$$

□

These tree accumulations are in fact tree homomorphisms. The upwards accumulation h^u defined with two parameter functions k_l and k_n is a tree homomorphism, $h^u = (\!|k'_l, k'_n|\!)_{\mathrm{b}}$, in which the two parameter functions are defined as follows.

$$\begin{array}{lcl} k'_l\ a & = & \mathsf{BLeaf}\ (k_l\ a) \\ k'_n\ l\ b\ r & = & \mathsf{BNode}\ l\ (k_n\ b\ (root_{\mathrm{b}}\ l)\ (root_{\mathrm{b}}\ r))\ r \end{array}$$

The downwards accumulation h^d defined with two functions g_l and g_r is a higher-order tree homomorphism, $h^d\ c\ t = (\!|k_l, k_n|\!)_{\mathrm{b}}\ t\ c$, in which the two parameter functions are defined as follows.

$$\begin{array}{lcl} k_l\ a & = & \lambda c.\mathsf{BLeaf}\ c \\ k_n\ b\ l\ r & = & \lambda c.\mathsf{BNode}\ c\ (l\ (g_l\ c\ b))\ (r\ (g_r\ c\ b)) \end{array}$$

2.5 Related Work

We will briefly mention related work on representations of data structures.

List Representations

Besides the join-list representation used in this thesis, cons-list and snoc-list presentations [Bir88, BdM96] are often used in theoretical studies in the field of functional programming. The cons-list representation has two constructors: the empty list, and the *cons* operator to put an element on the head of the given list. The snoc-list has similar constructors: the empty list, and the *snoc* operator to put an element on the last of the given list. Since those two representations have no freedom for balancing the computation of their homomorphisms, they are suitable for sequential programs but not for parallel programs.

Fortunately, there has been research to connect the sequential representations (cons-list) with the parallel representation (join-list). The diffusion theorem [HTI99] gives a powerful method to obtain suitable program on the join-list representation from a program recursively defined on the cons-list representation; the theorem gives us a method to parallelize sequential programs. Chin et al. [HTC98, CTH98] have studied a systematic method to derive an associative operator that plays an important role in parallelization, based on which Xu et al. [XKH04] build an automatic derivation system for parallelizing recursive linear functions with normalization rules.

Matrix Representations

Wise et al. [Wis84] proposed representation of two-dimensional arrays by quadtrees, in which two-dimensional arrays are recursively constructed by four small sub-arrays of the same size. This representation is suitable for describing recursive blocked algorithms [EGJK04], which can provide better performance than existing algorithms for some matrix computations such as LU and QR factorizations [FW03, Wis99]. However, the quadtree representation requires the size of two-dimensional arrays to be the power of two. Moreover, once a two-dimensional array is represented by a quadtree, we cannot reblock the array by restructuring the quadtree, which would prevent us from developing more parallelism in the recursive blocked algorithms on them.

Bikshandi et al. [BGH+06] proposed representation of a two-dimensional array by a hierarchically tiled array (HTA). An HTA is an array partitioned into tiles, and these tiles can be either conventional arrays or lower level HTAs. The outermost tiles are distributed across processors for parallelism and the inner tiles are utilized for locality. In the HTA programming, users are allowed to use recursive index accessing according to the structure of HTAs, so that they can easily transform conventional programs onto HTA programs. Communication and synchronization

are explicitly expressed by index accessing to remote tiles. Thus HTA programs can control relatively low-level parallelism and can be efficient implementation. However, it is not presented how to derive efficient HTA programs. We think it is good that we derive an efficient algorithm on the abide-tree then implement it on HTAs.

A more natural representation of a two-dimensional array is to use nested one-dimensional arrays (lists) [Bir88, Ski94, Jeu93]. The advantage is that many results developed for lists can be reused. However, this representation imposes much restriction on the access order of elements.

The abide-tree representation, as used in this chapter, was first proposed by Bird [Bir88], as an extension of one-dimensional join lists. However, the focus there is on derivation of sequential programs for manipulating two-dimensional arrays, and there is little study on the framework for developing efficient parallel programs. Our work provides a good complement.

Tree Representations

There has been research on tree representations with good parallelism for load-balancing. Matsuzaki [Mat07] has proposed the ternary-tree representation for binary trees, in which three special nodes are introduced for the flexibility of structures. The flexibility has been formalized as tree-associativity, which is a natural extension of the usual associativity of binary operators.

Chapter 3

Homomorphism-based Design of Parallel Skeletons

In this chapter, we will introduce homomorphism-based design of parallel skeletons for various data structures. Homomorphism-based skeletons have good composability inherited from homomorphisms. Thus, we can describe many problems by composing a small set of homomorphism-based skeletons. Also, we will be able to develop optimizations of skeleton compositions, owing to the nice algebraic laws of homomorphisms, which will be seen in the next chapter.

First, we will introduce parallel skeletons for lists. Then, we will design parallel skeletons for tow-dimensional arrays, which has remained as a challenging problem. We will also show some non-trivial programs written with the skeletons. Finally, we will show parallel skeletons for trees briefly.

3.1 Homomorphism-based Parallel Skeletons for Lists

First, we will define five parallel skeletons based on lists. Then, we will show some programs written with the skeletons, and some variants of the skeletons.

3.1.1 Definitions of Parallel Skeletons

We will define five parallel skeletons on lists: map and reduce for basic computations, zipwith for extended element-wise computations, and scan and scanr on lists for accumulation computations. Those functions will be defined based on the homomorphism. In the theory of Constructive Algorithmics [Bir88, Ski94, BdM96], these functions are known to be the most fundamental computation components for manipulating algebraic data structures and for being glued together to express complicated computations. Intuitive definitions of the skeletons are shown in Figure 3.1.

$$
\begin{array}{lcl}
\mathsf{map}\ f\ [a_1, a_2, \ldots, a_n] & = & [f\ a_1, f\ a_2, \ldots, f\ a_n] \\
\mathsf{reduce}\ (\oplus)\ [a_1, a_2, \ldots, a_n] & = & a_1 \oplus a_2 \oplus \cdots \oplus a_n \\
\mathsf{zipwith}\ f\ [a_1, a_2, \ldots, a_n]\ [b_1, b_2, \ldots, b_n] & = & [f\ a_1\ b_1, f\ a_2\ b_2, \ldots, f\ a_n\ b_n] \\
\mathsf{scan}\ (\oplus)\ [a_1, a_2, \ldots, a_n] & = & [y_1, y_2, \ldots, y_n] \\
 & & \textbf{where}\ \ y_i = a_1 \oplus a_2 \oplus \cdots \oplus a_i \\
\mathsf{scanr}\ (\oplus)\ [a_1, a_2, \ldots, a_n] & = & [z_1, z_2, \ldots, z_n] \\
 & & \textbf{where}\ \ z_i = a_i \oplus a_{i+1} \oplus \cdots \oplus a_n
\end{array}
$$

Figure 3.1. Intuitive definitions of parallel skeletons on lists.

Basic Skeletons: Map and Reduce

The skeletons map and reduce are two special cases of homomorphism.

The skeleton map applies the given function to each element of the given list. It is defined as follows.

$$
\begin{array}{lcl}
\mathsf{map}\ f\ (x +\!\!+ y) & = & (\mathsf{map}\ f\ x) +\!\!+ (\mathsf{map}\ f\ y) \\
\mathsf{map}\ f\ [a] & = & [f\ a]
\end{array}
$$

This definition means $\mathsf{map}\ f = (\![[\cdot] \circ f, +\!\!+]\!)$.

The skeleton reduce takes a sum of the given list with the given operator. It is defined as follows.

$$
\begin{array}{lcl}
\mathsf{reduce}\ (\oplus)\ (x +\!\!+ y) & = & (\mathsf{reduce}\ (\oplus)\ x) \oplus (\mathsf{reduce}\ (\oplus)\ y) \\
\mathsf{reduce}\ (\oplus)\ [a] & = & a
\end{array}
$$

This definition means $\mathsf{reduce}\ (\oplus) = (\![id, \oplus]\!)$.

Here are example uses of the skeletons. We can use map to increment all elements of a list as follows.

$$\mathsf{map}\ (1+)\ [6, 2, 1, 4, 3, 5] \ = \ [7, 3, 2, 5, 4, 6]$$

We can use reduce to take a sum of a list as follows.

$$\mathsf{reduce}\ (+)\ [6, 2, 1, 4, 3, 5] \ = \ 21$$

Extended Skeletons: Zipwith, Scan, and Scanr

The two skeletons defined above are primitive skeletons. We will define other skeletons that are extensions of these primitive skeletons.

The skeleton zipwith, an extension of map, takes two lists of the same length, and applies the given function f to every pair of corresponding elements of the lists.

$$
\begin{array}{lcl}
\mathsf{zipwith}\ f\ (x +\!\!+ y)\ (u +\!\!+ v) & = & (\mathsf{zipwith}\ f\ x\ u) +\!\!+ (\mathsf{zipwith}\ f\ y\ v) \\
\mathsf{zipwith}\ f\ [a]\ [b] & = & [f\ a\ b]
\end{array}
$$

Note that in the above definition the two argument lists should be divided in the way that the corresponding sub-lists have the same length.

Here, we define a specialization of zipwith to make a list of pairs.

$$\mathsf{zip} = \mathsf{zipwith}\ (\lambda x\ y.(x, y))$$

We may define similar zip and zipwith for the case when the number of input lists is three or more, and those that take k lists are denoted by zip_k and $\mathsf{zipwith}_k$.

The skeleton scan, an extension of reduce, holds all values generated in reducing a list by reduce.

$$\begin{aligned}
\mathsf{scan}\ (\oplus)\ (x +\!\!+ y) &= (\mathsf{scan}\ (\oplus)\ x) \oplus' (\mathsf{scan}\ (\oplus)\ y) \\
&\quad \textbf{where}\ \ sx \oplus' sy = sx +\!\!+ \mathsf{map}\ ((\mathsf{reduce}\ (\gg)\ sx) \oplus\ )\ sy \\
\mathsf{scan}\ (\oplus)\ [a] &= [a]
\end{aligned}$$

This definition means $\mathsf{scan}\ (\oplus) = (\![[\cdot], \oplus']\!)$.

Also, the skeleton scanr is defined as a reverse of scan, which holds all values generated during a reduction in the reverse order.

$$\begin{aligned}
\mathsf{scanr}\ (\oplus)\ (x +\!\!+ y) &= (\mathsf{scanr}\ (\oplus)\ x) \oplus'' (\mathsf{scanr}\ (\oplus)\ y) \\
&\quad \textbf{where}\ \ sx \oplus'' sy = \mathsf{map}\ (\ \oplus (\mathsf{reduce}\ (\ll)\ sy))\ sx +\!\!+ sy \\
\mathsf{scanr}\ (\oplus)\ [a] &= [a]
\end{aligned}$$

This definition means $\mathsf{scanr}\ (\oplus) = (\![[\cdot], \oplus'']\!)$.

Here are example uses of the skeletons. We can use zipwith to add two lists element-wisely.

$$\mathsf{zipwith}\ (+)\ [6, 2, 1, 4, 3, 5]\ [2, 7, 6, 1, 0, 4] \ = \ [8, 9, 7, 5, 3, 9]$$

We can use scan and scanr to compute prefix sums and suffix sums, respectively.

$$\begin{aligned}
\mathsf{scan}\ (+)\ [6, 2, 1, 4, 3, 5] &= [6, 8, 9, 13, 16, 21] \\
\mathsf{scanr}\ (+)\ [6, 2, 1, 4, 3, 5] &= [21, 15, 13, 12, 8, 5]
\end{aligned}$$

3.1.2 Example Programs and Variants of Skeletons

Composing these skeletons defined above, we can describe many useful functions as follows.

$$\begin{aligned}
id &= \mathsf{reduce}\ (+\!\!+) \circ \mathsf{map}\ [\cdot] \\
reverse &= \mathsf{reduce}\ (\tilde{+\!\!+}) \circ \mathsf{map}\ [\cdot] \\
flatten &= \mathsf{reduce}\ (+\!\!+) \\
length &= \mathsf{reduce}\ (+) \circ \mathsf{map}\ (\lambda x.\ 1) \\
last &= \mathsf{reduce}\ (\gg) \\
head &= \mathsf{reduce}\ (\ll)
\end{aligned}$$

The function *id* is the identity function of lists. The function *reverse* reverses the given list. The function *flatten* flattens a list of lists into a list. The function *length* returns the length of the given list. The functions *last* and *head* returns the last element and the first element of the given list, respectively.

Finally, we will defined some variants of skeletons and useful functions defined with them.

$$
\begin{array}{lcl}
\mathsf{scan}'\ (\oplus)\ e & = & \mathsf{map}\ (e\oplus) \circ \mathsf{scan}\ (\oplus) \\
\mathsf{scanr}'\ (\oplus)\ e & = & \mathsf{map}\ (\oplus e) \circ \mathsf{scanr}\ (\oplus) \\
\mathsf{shift}_{\gg}\ e & = & \mathsf{map}\ \pi_1 \circ \mathsf{scan}'\ (\oplus)\ (_, e, _) \circ \mathsf{map}\ f \\
 & & \textbf{where}\ \ f\ a = (a, a, \mathsf{True}) \\
 & & \qquad\qquad (_, p, _) \oplus (_, c, \mathsf{True}) = (p, c, \mathsf{False}) \\
 & & \qquad\qquad (_, _, _) \oplus (p, c, \mathsf{False}) = (p, c, \mathsf{False}) \\
\mathsf{shift}_{\ll}\ e & = & \mathsf{map}\ \pi_1 \circ \mathsf{scanr}'\ (\oplus)\ (_, e, _) \circ \mathsf{map}\ f \\
 & & \textbf{where}\ \ f\ a = (a, a, \mathsf{True}) \\
 & & \qquad\qquad (_, c, \mathsf{True}) \oplus (_, p, _) = (p, c, \mathsf{False}) \\
 & & \qquad\qquad (p, c, \mathsf{False}) \oplus (_, _, _) = (p, c, \mathsf{False}) \\
\mathsf{filter}\, p & = & (\![\textbf{if}\ p\ a\ \textbf{then}\ [a]\ \textbf{else}\ [\,], +\!\!+]\!) \\
\mathsf{init} & = & \mathit{flatten} \circ \mathsf{shift}_{\gg}\ [\,] \circ \mathsf{map}\ [\cdot] \\
\mathsf{tail} & = & \mathit{flatten} \circ \mathsf{shift}_{\ll}\ [\,] \circ \mathsf{map}\ [\cdot] \\
\mathit{drop}\ n\ x & = & \mathsf{tail}\ (\mathit{drop}\ (n-1)\ x) \\
\mathit{drop}\ 0\ x & = & x \\
\mathit{take}\ n\ x & = & [\mathit{head}\ x] +\!\!+ \mathit{take}\ (n-1)\ (\mathsf{tail}\ x) \\
\mathit{take}\ 0\ x & = & [\,] \\
\mathit{taker}\ n & = & \mathit{reverse} \circ \mathit{take}\ n \circ \mathit{reverse} \\
\mathit{dropr}\ n & = & \mathit{reverse} \circ \mathit{drop}\ n \circ \mathit{reverse}
\end{array}
$$

Their intuitive definition is shown below.

$$
\begin{array}{lcl}
\mathsf{scan}'\ (\oplus)\ e\ [a_1, \ldots, a_n] & = & [b_1, \ldots, b_n] \\
 & & \textbf{where}\ \ b_i = e \oplus a_1 \oplus \cdots \oplus a_i \\
\mathsf{scanr}'\ (\oplus)\ e\ [a_1, \ldots, a_n] & = & [c_1, \ldots, c_n] \\
 & & \textbf{where}\ \ c_i = a_i \oplus \cdots \oplus a_n \oplus e \\
\mathsf{shift}_{\gg}\ e\ [a_1, \ldots, a_n] & = & [e, a_1, \ldots, a_{n-1}] \\
\mathsf{shift}_{\ll}\ e\ [a_1, \ldots, a_n] & = & [a_2, \ldots, a_n, e] \\
\mathsf{filter}\, p\, [a_1, a_2, \ldots, a_n] & = & [a_{i_1}, a_{i_2}, \ldots, a_{i_m}]\ \ \textbf{where}\ \ p\ (a_{i_j}) = \mathsf{True} \\
\mathsf{init}\ [a_1, \ldots, a_n] & = & [a_1, \ldots, a_{n-1}] \\
\mathsf{tail}\ [a_1, \ldots, a_n] & = & [a_2, \ldots, a_n] \\
\mathit{take}\ k\ [a_1, \ldots, a_n] & = & [a_1, \ldots, a_k] \\
\mathit{drop}\ k\ [a_1, \ldots, a_n] & = & [a_{k+1}, \ldots, a_n] \\
\mathit{taker}\ k\ [a_1, \ldots, a_n] & = & [a_{n-k+1}, \ldots, a_n] \\
\mathit{dropr}\ k\ [a_1, \ldots, a_n] & = & [a_1, \ldots, a_{n-k}]
\end{array}
$$

Skeletons scan' and scanr' take the initial values of reductions. Skeleton $\mathsf{shift}_{\gg}$ shifts the elements by putting the given element to the head and discarding the last element. Skeleton $\mathsf{shift}_{\ll}$ does the converse. Two functions init and tail take initial

segment and the tail segment of the list, respectively. The function *take* k picks up the first k elements of the given list, and the function *drop* removes the first k elements from the given list. Conversely, the function *taker* k picks up the last k elements of the given list, and the faction *dropr* removes the last k elements from the given list.

3.2 Homomorphism-based Parallel Skeletons for Two-Dimensional Arrays

In this section, we will design a set of parallel skeletons for manipulating two-dimensional arrays. Also, we will show that compositions of the skeletons are powerful enough to describe useful parallel algorithms. The power of skeleton compositions will be shown by describing nontrivial problems such as matrix multiplication and QR decomposition.

3.2.1 Definitions of Parallel Skeletons

We will define five parallel skeletons on two-dimensional arrays: map and reduce for basic computations, zipwith for extended element-wise computations, and scan and scanrfor accumulation computations. In the theory of Constructive Algorithmics [Bir88, Ski94, BdM96], list versions of these functions are known to be the most fundamental computation components for manipulating algebraic data structures and for being glued together to express complicated computations.

Intuitive definitions of the skeletons are shown in Figure 3.2.

Basic Skeletons: Map and Reduce

The skeletons map and reduce are two special cases of homomorphism.

The skeleton map applies the given function f to every element of the given two-dimensional array. Its definition is as follows.

$$
\begin{aligned}
\mathsf{map}\, f\, |a| &= |f\, a| \\
\mathsf{map}\, f\, (x \ominus y) &= (\mathsf{map}\, f\, x) \ominus (\mathsf{map}\, f\, y) \\
\mathsf{map}\, f\, (x \obar y) &= (\mathsf{map}\, f\, x) \obar (\mathsf{map}\, f\, y)
\end{aligned}
$$

This definition means $\mathsf{map}\, f = \llparenthesis |\cdot| \circ f, \ominus, \obar \rrparenthesis$.

The skeleton reduce collapses a two-dimensional array into a value using two associative binary operators $\oplus$ and $\otimes$ that have the abide property. In other words, the skeleton reduce takes a *sum* of the given array with the given operators, in which the sum of vertical direction is computed with the first operator $\oplus$, and the sum of horizontal direction is computed with the second operator $\otimes$. Its definition is given

$$\mathsf{map}\ f \begin{pmatrix} x_{11} & x_{12} & \cdots & x_{1n} \\ x_{21} & x_{22} & \cdots & x_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{m1} & x_{m2} & \cdots & x_{mn} \end{pmatrix} = \begin{pmatrix} f\ x_{11} & f\ x_{12} & \cdots & f\ x_{1n} \\ f\ x_{21} & f\ x_{22} & \cdots & f^{x}_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ f\ x_{m1} & f\ x_{m2} & \cdots & f\ x_{mn} \end{pmatrix}$$

$$\mathsf{reduce}\ (\oplus, \otimes) \begin{pmatrix} x_{11} & x_{12} & \cdots & x_{1n} \\ x_{21} & x_{22} & \cdots & x_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{m1} & x_{m2} & \cdots & x_{mn} \end{pmatrix} = \begin{array}{c} (x_{11} \otimes x_{12} \otimes \cdots \otimes x_{1n}) \oplus \\ (x_{21} \otimes x_{22} \otimes \cdots \otimes x_{2n}) \oplus \\ \vdots \\ (x_{m1} \otimes x_{m2} \otimes \cdots \otimes x_{mn}) \end{array}$$

$$\mathsf{zipwith}\ f \begin{pmatrix} x_{11} & x_{12} & \cdots & x_{1n} \\ x_{21} & x_{22} & \cdots & x_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{m1} & x_{m2} & \cdots & x_{mn} \end{pmatrix} \begin{pmatrix} y_{11} & y_{12} & \cdots & y_{1n} \\ y_{21} & y_{22} & \cdots & y_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ y_{m1} & y_{m2} & \cdots & y_{mn} \end{pmatrix}$$

$$= \begin{pmatrix} f\ x_{11}\ y_{11} & f\ x_{12}\ y_{12} & \cdots & f\ x_{1n}\ y_{1n} \\ f\ x_{21}\ y_{21} & f\ x_{22}\ y_{22} & \cdots & f\ x_{2n}\ y_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ f\ x_{m1}\ y_{m1} & f\ x_{m2}\ y_{m2} & \cdots & f\ x_{mn}\ y_{mn} \end{pmatrix}$$

$$\mathsf{scan}\ (\oplus, \otimes) \begin{pmatrix} x_{11} & x_{12} & \cdots & x_{1n} \\ x_{21} & x_{22} & \cdots & x_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{m1} & x_{m2} & \cdots & x_{mn} \end{pmatrix} = \begin{pmatrix} y_{11} & y_{12} & \cdots & y_{1n} \\ y_{21} & y_{22} & \cdots & y_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ y_{m1} & y_{m2} & \cdots & y_{mn} \end{pmatrix}$$

$$\textbf{where}\ y_{ij} = \begin{array}{c} (x_{11} \otimes x_{12} \otimes \cdots \otimes x_{1j}) \oplus \\ (x_{21} \otimes x_{22} \otimes \cdots \otimes x_{2j}) \oplus \\ \vdots \\ (x_{i1} \otimes x_{i2} \otimes \cdots \otimes x_{ij}) \end{array}$$

$$\mathsf{scanr}\ (\oplus, \otimes) \begin{pmatrix} x_{11} & x_{12} & \cdots & x_{1n} \\ x_{21} & x_{22} & \cdots & x_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{m1} & x_{m2} & \cdots & x_{mn} \end{pmatrix} = \begin{pmatrix} z_{11} & z_{12} & \cdots & z_{1n} \\ z_{21} & z_{22} & \cdots & z_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ z_{m1} & z_{m2} & \cdots & z_{mn} \end{pmatrix}$$

$$\textbf{where}\ z_{ij} = \begin{array}{c} (x_{ij} \otimes \cdots \otimes x_{in}) \oplus \\ \vdots \\ (x_{mj} \otimes \cdots \otimes x_{mn}) \end{array}$$

Figure 3.2. Intuitive definition of five skeletons on two-dimensional arrays.

as follows.

$$\begin{array}{lcl}
\mathsf{reduce}\ (\oplus,\otimes)\,|a| & = & a \\
\mathsf{reduce}\ (\oplus,\otimes)\,(x \ominus y) & = & (\mathsf{reduce}\ (\oplus,\otimes)\,x) \oplus (\mathsf{reduce}\ (\oplus,\otimes)\,y) \\
\mathsf{reduce}\ (\oplus,\otimes)\,(x \phi y) & = & (\mathsf{reduce}\ (\oplus,\otimes)\,x) \otimes (\mathsf{reduce}\ (\oplus,\otimes)\,y)\ ,
\end{array}$$

This definition means $\mathsf{reduce}(\oplus,\otimes) = (\!|id,\oplus,\otimes|\!)$.

Here are some example uses of the skeletons. We can use map to take a scalar product of an array as follows.

$$\mathsf{map}\,(2\times)\begin{pmatrix} 6 & 2 & 1 \\ 4 & 3 & 5 \end{pmatrix} = \begin{pmatrix} 12 & 4 & 2 \\ 8 & 6 & 10 \end{pmatrix}$$

We can use reduce to take a sum of an array as follows.

$$\mathsf{reduce}\ (+,+)\begin{pmatrix} 6 & 2 & 1 \\ 4 & 3 & 5 \end{pmatrix} = 21$$

Extended Skeleton for Element-wise Computation: Zipwith

The two skeletons defined above are primitive skeletons. We define other skeletons that are extensions of these primitive skeletons.

The skeleton zipwith, an extension of map, takes two two-dimensional arrays of the same shape, and applies a function f to every pair of corresponding elements of the arrays.

$$\begin{array}{lcl}
\mathsf{zipwith}\,f\,|a|\,|b| & = & |f\,a\,b| \\
\mathsf{zipwith}\,f\,(x \ominus y)\,(u \ominus v) & = & (\mathsf{zipwith}\,f\,x\,u) \ominus (\mathsf{zipwith}\,f\,y\,v) \\
\mathsf{zipwith}\,f\,(x \phi y)\,(u \phi v) & = & (\mathsf{zipwith}\,f\,x\,u) \phi (\mathsf{zipwith}\,f\,y\,v)
\end{array}$$

Note that in the above definition the argument two-dimensional arrays of the function should be divided in the way that the sizes of x and u are the same.

We also define useful function zip, which is a specialization of zipwith to make a two-dimensional array of pairs of corresponding elements.

$$\mathsf{zip}\,(u,v) = \mathsf{zipwith}\,(\lambda xy.\,(x,y))\,u\,v$$

We may define similar zip and zipwith for the case when the number of input arrays is three or more, and those that take k arrays are denoted by zip_k and $\mathsf{zipwith}_k$. Also, we define unzip to be the inverse of zip.

For example, we can use zipwith to compute an addition of two arrays as follows.

$$\mathsf{zipwith}\,(+)\begin{pmatrix} 6 & 2 & 1 \\ 4 & 3 & 5 \end{pmatrix}\begin{pmatrix} 0 & 8 & 2 \\ 9 & 1 & 7 \end{pmatrix} = \begin{pmatrix} 6 & 10 & 3 \\ 13 & 4 & 12 \end{pmatrix}$$

Simple Functions Described with Map, Reduce, and Zipwith

Composing these skeletons defined above, we can describe many useful functions as follows. We will use these functions to define complex skeletons scan and scanr.

$$
\begin{array}{lcl}
id & = & \mathsf{reduce}(\ominus, \phi) \circ \mathsf{map}\,|\cdot| \\
tr & = & \mathsf{reduce}(\phi, \ominus) \circ \mathsf{map}\,|\cdot| \\
rev & = & \mathsf{reduce}(\tilde{\ominus}, \tilde{\phi}) \circ \mathsf{map}\,|\cdot| \\
flatten & = & \mathsf{reduce}(\ominus, \phi) \\
height & = & \mathsf{reduce}(+, \ll) \circ \mathsf{map}\,(\lambda x.\,1) \\
width & = & \mathsf{reduce}(\ll, +) \circ \mathsf{map}\,(\lambda x.\,1) \\
cols & = & \mathsf{reduce}(\mathsf{zipwith}(\ominus), \phi) \circ \mathsf{map}\,||\cdot|| \\
rows & = & \mathsf{reduce}(\ominus, \mathsf{zipwith}(\phi)) \circ \mathsf{map}\,||\cdot|| \\
\mathsf{reduce}_c(\oplus) & = & \mathsf{map}(\mathsf{reduce}(\oplus, \ll)) \circ cols \\
\mathsf{reduce}_r(\otimes) & = & \mathsf{map}(\mathsf{reduce}(\ll, \otimes)) \circ rows \\
\mathsf{map}_c\,f & = & \mathsf{reduce}(\ll, \phi) \circ \mathsf{map}\,f \circ cols \\
\mathsf{map}_r\,f & = & \mathsf{reduce}(\ominus, \ll) \circ \mathsf{map}\,f \circ rows \\
madd & = & \mathsf{zipwith}(+) \\
msub & = & \mathsf{zipwith}(-) \\
top & = & \mathsf{reduce}(\gg, \phi) \circ \mathsf{map}\,|\cdot| \\
bottom & = & \mathsf{reduce}(\gg, \phi) \circ \mathsf{map}\,|\cdot| \\
left & = & \mathsf{reduce}(\ominus, \gg) \circ \mathsf{map}\,|\cdot| \\
right & = & \mathsf{reduce}(\ominus, \gg) \circ \mathsf{map}\,|\cdot|
\end{array}
$$

Here, $||\cdot||$ is abbreviation of $|\cdot| \circ |\cdot|$. The function id is the identity function of $AbideTree$, and tr is the matrix-transposing function. The function rev takes a two-dimensional array and returns the array reversed in the vertical and the horizontal direction. The function $flatten$ flattens nested arrays. The functions $height$ and $width$ return the number of rows and columns, respectively. The functions $cols$ and $rows$ return arrays of which elements are columns and rows of the array of the argument, respectively. The functions reduce_c and reduce_r are specializations of reduce to collapse two-dimensional arrays in column direction and row direction, respectively. They return a row-vector (an array of which height is one) and a column-vector (an array of which width is one), respectively. The functions map_c and map_r are specializations of map to apply functions to each column and row, respectively (i.e. the function of the argument takes column-vector or row-vector). The functions $madd$ and $msub$ denote matrix addition and subtraction, respectively. The functions top, $bottom$, $left$, and $right$ return the bottom row, the top row, the leftmost column, and the rightmost column, respectively.

Extended Skeleton for Accumulation Computation: Scan and Scanr

We will define the rest of the extended skeletons.

The skeleton scan, an extension of reduce, holds all values generated in reducing a two-dimensional array by reduce. Its definition is as follows.

$$\begin{array}{lcl} \mathsf{scan}\ (\oplus, \otimes)\ |a| & = & |a| \\ \mathsf{scan}\ (\oplus, \otimes)\ (x \ominus y) & = & (\mathsf{scan}\ (\oplus, \otimes)\ x) \oplus' (\mathsf{scan}\ (\oplus, \otimes)\ y) \\ \mathsf{scan}\ (\oplus, \otimes)\ (x \phi\ y) & = & (\mathsf{scan}\ (\oplus, \otimes)\ x) \otimes' (\mathsf{scan}\ (\oplus, \otimes)\ y) \end{array}$$

Here operators $\oplus'$ and $\otimes'$ are defined as follows.

$$\begin{array}{lcl} sx \oplus' sy & = & sx \ominus sy' \\ & & \quad \mathbf{where}\ sy' = \mathsf{map}_r\ (\mathsf{zipwith}\ (\oplus)\ (bottom\ sx))\ sy \\ sx \otimes' sy & = & sx \phi\ sy' \\ & & \quad \mathbf{where}\ sy' = \mathsf{map}_c\ (\mathsf{zipwith}\ (\otimes)\ (right\ sx))\ sy \end{array}$$

The definition means $\mathsf{scan}\ (\oplus, \otimes) = (\!|\, |\cdot|, \oplus', \otimes' |\!)$. Note that $\oplus$ and $\otimes$ must be associative and have the abide property.

The skeleton scanr is the reverse of scan, i.e., holds all values generated during the reduction in the reverse order. Its definition is as follows.

$$\begin{array}{lcl} \mathsf{scanr}\ (\oplus, \otimes)\ |a| & = & |a| \\ \mathsf{scanr}\ (\oplus, \otimes)\ (x \ominus y) & = & (\mathsf{scanr}\ (\oplus, \otimes)\ x) \oplus' (\mathsf{scanr}\ (\oplus, \otimes)\ y) \\ \mathsf{scanr}\ (\oplus, \otimes)\ (x \phi\ y) & = & (\mathsf{scanr}\ (\oplus, \otimes)\ x) \otimes' (\mathsf{scanr}\ (\oplus, \otimes)\ y) \end{array}$$

Here operators $\oplus'$ and $\otimes'$ are defined as follows.

$$\begin{array}{lcl} sx \oplus' sy & = & sx' \ominus sy \\ & & \quad \mathbf{where}\ sx' = \mathsf{map}_r\ (\lambda x.\,\mathsf{zipwith}\ (\oplus)\ x\ (top\ sy))\ sx \\ sx \otimes' sy & = & sx' \phi\ sy \\ & & \quad \mathbf{where}\ sx' = \mathsf{map}_c\ (\lambda x.\,\mathsf{zipwith}\ (\otimes)\ x\ (left\ sy))\ sx \end{array}$$

The definition means $\mathsf{scanr}\ (\oplus, \otimes) = (\!|\, |\cdot|, \oplus', \otimes' |\!)$. Note that $\oplus$ and $\otimes$ must be associative and have the abide property.

Here are example uses of the skeletons. We can compute a prefix rectangle sum (an upper-left prefix sum) and a suffix rectangle sum (a lower-right suffix sum) using scan and scanr, respectively.

$$\mathsf{scan}\ (+,+) \begin{pmatrix} 6 & 2 & 1 \\ 4 & 3 & 5 \end{pmatrix} = \begin{pmatrix} 6 & 8 & 9 \\ 10 & 15 & 21 \end{pmatrix}$$
$$\mathsf{scanr}\ (+,+) \begin{pmatrix} 6 & 2 & 1 \\ 4 & 3 & 5 \end{pmatrix} = \begin{pmatrix} 21 & 11 & 6 \\ 12 & 8 & 5 \end{pmatrix}$$

Using the skeletons scan and scanr, we can define $allred_r$ and $allred_c$ to the results of reduce_r and reduce_c in rows and columns, respectively. These functions are defined as follows. We will use them in later sections.

$$\begin{array}{lcl} allred_c(\oplus) & = & \mathsf{scanr}(\gg, \ll) \circ \mathsf{scan}(\oplus, \gg) \\ allred_r(\otimes) & = & \mathsf{scanr}(\ll, \gg) \circ \mathsf{scan}(\gg, \otimes) \end{array}$$

Composition of the Skeletons: Accumulate

Finally, we will define a complex general skeleton accumulate that can deal with accumulation parameters. This skeleton is sometimes useful to develop general theory for skeletons. Also, we will show that the skeleton is a composition of the previous skeletons.

The idea of this skeleton is that we can calculate the internal value of a subarray if we know the intermediate result on the edges of upper and left subarrays. This idea is quite natural, because the cross section of a two-dimensional array is an edge, and the minimum information that should be transmitted between them is the edge. As seen later, this skeleton is a general skeleton in the sense that it contains all skeletons defined above.

$$\begin{array}{l}\mathsf{accumulate}\ (x_1 \ominus x_2)\ (e, u, v_1 \ominus v_2) = y_1 \oplus' y_2 \\ \quad \textbf{where} \\ \qquad \begin{array}{lcl} y_1 &=& \mathsf{accumulate}\ x_1\ (e, u, v_1) \\ y_2 &=& \mathsf{accumulate}\ x_2\ (e_2, u_2, v_2) \\ e_2 &=& e \oplus \mathsf{reduce}\ (\oplus, \otimes)\ v_1 \\ u_2 &=& \mathsf{zipwith}\ (\oplus)\ u\ (\mathsf{reduce}_c\ (\oplus)\ (\mathsf{map}\ q\ x_1)) \end{array}\end{array}$$

$$\begin{array}{l}\mathsf{accumulate}\ (x_1 \phi x_2)\ (e, u_1 \phi u_2, v) = y_1 \otimes' y_2 \\ \quad \textbf{where} \\ \qquad \begin{array}{lcl} y_1 &=& \mathsf{accumulate}\ x_1\ (e, u_1, v) \\ y_2 &=& \mathsf{accumulate}\ x_2\ (e_2, u_2, v_2) \\ e_2 &=& e \otimes \mathsf{reduce}\ (\oplus, \otimes)\ u_1 \\ u_2 &=& \mathsf{zipwith}\ (\otimes)\ v\ (\mathsf{reduce}_r\ (\otimes)\ (\mathsf{map}\ q\ x_1)) \end{array}\end{array}$$

$$\mathsf{accumulate}\ |a|\ (e, |u|, |v|) = p\ a\ ((e \otimes u) \oplus (v \otimes q\ a))$$

To create the accumulation parameters, the skeleton accumulate collapses the argument two-dimensional array with a function q and two abiding operators $\oplus$ and $\otimes$, and puts them on each element. Then, to get the result value it consumes the accumulation parameters and the element by replacing constructors with a function p and operators $\oplus'$ and $\otimes'$. The meaning of the accumulation parameters is as follows: e is the resulting value of collapsing the upper-left subarray of the argument array with the operators $\oplus$ and $\otimes$, u is the resulting row-vector of collapsing each column of the upper subarray with the operator $\oplus$, and v is the resulting column-vector of collapsing each row of the left subarray with the operator $\otimes$. We write $[\![(\oplus', \otimes', p), (\oplus, \otimes, q)]\!]$ to denote this accumulate. Here, $(\oplus, \otimes, q)$ corresponds to the computation of accumulation parameters and $(\oplus', \otimes', p)$ corresponds to the computation of resulting value.

The skeleton accumulate has somewhat complicated definition, however, it can be described as a composition of the other skeletons defined above. We give here the theorem showing the relation between accumulate and other skeletons. The functions

$drop_c$ and $drop_r$ used in the theorem drop left-columns and upper-rows respectively:

$$
\begin{array}{rcll}
drop_c\ n\ (x \mathbin{⦶} y) & = & y & \text{s.t. } width\ x = n\ , \\
drop_r\ n\ (x \ominus y) & = & y & \text{s.t. } height\ x = n\ .
\end{array}
$$

This theorem is an extension of the diffusion theorem on lists [HTI99].

Theorem 3.1 (Diffusion)**.** The skeleton $\mathsf{accumulate}$ can be diffused to the composition of other skeletons as follows.

$$
\begin{array}{rl}
[\![(\oplus', \otimes', p), (\oplus, \otimes, q)]\!]\ x\ (e, u, v) = & \mathbf{let}\ \ x' = \mathsf{map}\ \ q\ x \\
& \phantom{\mathbf{let}}\ \ x'' = \mathsf{scan}(\oplus, \otimes)\ ((|e| \mathbin{⦶} u) \ominus (v \mathbin{⦶} x)) \\
& \phantom{\mathbf{let}}\ \ x''' = drop_c\ 1\ (drop_r\ 1\ x'') \\
& \phantom{\mathbf{let}}\ \ x'''' = \mathsf{zipwith}\ \ p\ x\ x''' \\
& \mathbf{in}\ \ \ \mathsf{reduce}(\oplus', \otimes')\ x''''
\end{array}
$$

Proof. The theorem is proved by induction on the structure of abide-trees.

The base case is proved as follows.

$$
\begin{array}{cl}
& \text{LHS}\ |a|\ (e, |u|, |v|) \\
= & \quad \{ \text{ The left hand side } \} \\
& [\![(\oplus', \otimes', p), (\oplus, \otimes, q)]\!]\ |a|\ (e, |u|, |v|) \\
= & \quad \{ \text{ Definition of } \mathsf{accumulate} \ \} \\
& p\ a\ ((e \otimes u) \oplus (v \otimes q\ a)) \\
= & \quad \{ \text{ Definition of } \mathsf{reduce} \ \} \\
& \mathsf{reduce}\ \ (\oplus', \otimes')\ |p\ a\ ((e \otimes u) \oplus (v \otimes q\ a))| \\
= & \quad \{ \text{ Definition of } \mathsf{zipwith} \ \} \\
& \mathbf{let}\ \ x'''' = \mathsf{zipwith}\ \ p\ |a|\ |((e \otimes u) \oplus (v \otimes q\ a))| \\
& \mathbf{in}\ \ \mathsf{reduce}\ \ (\oplus', \otimes')\ x'''' \\
= & \quad \{ \text{ Definition of } \mathsf{scan},\ drop_r \text{ and } drop_c \ \} \\
& \mathbf{let}\ \ x'' = \mathsf{scan}\ \ (\oplus, \otimes)\ ((|e| \mathbin{⦶} |u|) \ominus (|v| \mathbin{⦶} |a|)) \\
& \phantom{\mathbf{let}}\ \ x''' = drop_c\ 1\ (drop_r\ 1\ x'') \\
& \phantom{\mathbf{let}}\ \ x'''' = \mathsf{zipwith}\ \ p\ |a|\ |((e \otimes u) \oplus (v \otimes q\ a))| \\
& \mathbf{in}\ \ \mathsf{reduce}\ \ (\oplus', \otimes')\ x'''' \\
= & \quad \{ \text{ Definition of } \mathsf{map} \ \} \\
& \mathbf{let}\ \ x' = \mathsf{map}\ \ q\ |a| \\
& \phantom{\mathbf{let}}\ \ x'' = \mathsf{scan}\ \ (\oplus, \otimes)\ ((|e| \mathbin{⦶} |u|) \ominus (|v| \mathbin{⦶} x')) \\
& \phantom{\mathbf{let}}\ \ x''' = drop_c\ 1\ (drop_r\ 1\ x'') \\
& \phantom{\mathbf{let}}\ \ x'''' = \mathsf{zipwith}\ \ p\ |a|\ |((e \otimes u) \oplus (v \otimes q\ a))| \\
& \mathbf{in}\ \ \mathsf{reduce}\ \ (\oplus', \otimes')\ x'''' \\
= & \quad \{ \text{ The right hand side } \} \\
& \text{RHS}\ |a|\ (e, |u|, |v|)
\end{array}
$$

The induction case for $\ominus$ is proved as follows.

$$
\begin{array}{cl}
& \text{LHS}\ (x_1 \ominus x_2)\ (e, u, v_1 \ominus v_2) \\
= & \quad \{ \text{ The left hand side } \} \\
& [\![(\oplus', \otimes', p), (\oplus, \otimes, q)]\!]\ (x_1 \ominus x_2)\ (e, u, v_1 \ominus v_2)
\end{array}
$$

$$
\begin{array}{ll}
= & \quad \{ \text{ Definition of } \mathsf{accumulate} \ \} \\
& \textbf{let } e_2 = e \oplus \mathsf{reduce}(\oplus, \otimes)\ v_1 \\
& \qquad u_2 = \mathsf{zipwith}\ (\oplus)\ u\ (\mathsf{reduce}_c(\oplus)\ (\mathsf{map}\ q\ x_1)) \\
& \qquad y_1 = \mathsf{accumulate}\ x_1\ (e, u, v_1) \\
& \qquad y_2 = \mathsf{accumulate}\ x_2\ (e_2, u_2, v_2) \\
& \textbf{in } y_1 \oplus' y_2 \\
= & \quad \{ \text{ Hypothesis of induction } \} \\
& \textbf{let } e_2 = e \oplus \mathsf{reduce}\ (\oplus, \otimes)\ v_1 \\
& \qquad u_2 = \mathsf{zipwith}\ (\oplus)\ u\ (\mathsf{reduce}_c\ (\oplus)\ (\mathsf{map}\ q\ x_1)) \\
& \qquad y_1 = \textbf{let } x_1' = \mathsf{map}\ q\ x_1 \\
& \qquad\qquad\qquad x_1'' = \mathsf{scan}\ (\oplus, \otimes)\ ((|e|\ \phi\ u) \ominus (v_1\ \phi\ x_1')) \\
& \qquad\qquad\qquad x_1''' = drop_c\ 1\ (drop_r\ 1\ x_1'') \\
& \qquad\qquad\qquad x_1'''' = \mathsf{zipwith}\ p\ x_1\ x_1''' \\
& \qquad\qquad\quad \textbf{in } \mathsf{reduce}\ (\oplus', \otimes')\ x_1'''' \\
& \qquad y_2 = \textbf{let } x_2' = \mathsf{map}\ q\ x_2 \\
& \qquad\qquad\qquad x_2'' = \mathsf{scan}\ (\oplus, \otimes)\ ((|e_2|\ \phi\ u_2) \ominus (v_2\ \phi\ x_2')) \\
& \qquad\qquad\qquad x_2''' = drop_c\ 1\ (drop_r\ 1\ x_2'') \\
& \qquad\qquad\qquad x_2'''' = \mathsf{zipwith}\ p\ x_2\ x_2''' \\
& \qquad\qquad\quad \textbf{in } \mathsf{reduce}\ (\oplus', \otimes')\ x_2'''' \\
& \textbf{in } y_1 \oplus' y_2 \\
= & \quad \{ \text{ Expansion of inner } \textbf{let} \ \} \\
& \textbf{let } e_2 = e \oplus \mathsf{reduce}\ (\oplus, \otimes)\ v_1 \\
& \qquad u_2 = \mathsf{zipwith}\ (\oplus)\ u\ (\mathsf{reduce}_c\ (\oplus)\ (\mathsf{map}\ q\ x_1)) \\
& \qquad x_1' = \mathsf{map}\ q\ x_1 \\
& \qquad x_2' = \mathsf{map}\ q\ x_2 \\
& \qquad x_1'' = \mathsf{scan}\ (\oplus, \otimes)\ ((|e|\ \phi\ u) \ominus (v_1\ \phi\ x_1')) \\
& \qquad x_2'' = \mathsf{scan}\ (\oplus, \otimes)\ ((|e_2|\ \phi\ u_2) \ominus (v_2\ \phi\ x_2')) \\
& \qquad x_1''' = drop_c\ 1\ (drop_r\ 1\ x_1'') \\
& \qquad x_2''' = drop_c\ 1\ (drop_r\ 1\ x_2'') \\
& \qquad x_1'''' = \mathsf{zipwith}\ p\ x_1\ x_1''' \\
& \qquad x_2'''' = \mathsf{zipwith}\ p\ x_2\ x_2''' \\
& \textbf{in } \mathsf{reduce}\ (\oplus', \otimes')\ x_1'''' \oplus' \mathsf{reduce}\ (\oplus', \otimes')\ x_2'''' \\
= & \quad \{ \text{ Property of } \mathsf{scan} \text{ and edges proved below } \} \\
& \textbf{let } x_1' = \mathsf{map}\ q\ x_1 \\
& \qquad x_2' = \mathsf{map}\ q\ x_2 \\
& \qquad \underline{x_1''} = \mathsf{scan}\ (\oplus, \otimes)\ ((|e|\ \phi\ u) \ominus (v_1\ \phi\ x_1')) \\
& \qquad \overline{x_2''} = \mathsf{map}_r\ (\mathsf{zipwith}\ (\oplus)\ (bottom\ x_1''))\ (\mathsf{scan}\ (\oplus, \otimes)\ (v_2\ \phi\ x_2')) \\
& \qquad x_1''' = drop_c\ 1\ \underline{(drop_r\ 1\ x_1'')} \\
& \qquad x_2''' = drop_c\ 1\ \overline{x_2''} \\
& \qquad x_1'''' = \mathsf{zipwith}\ p\ x_1\ x_1''' \\
& \qquad x_2'''' = \mathsf{zipwith}\ p\ x_2\ x_2''' \\
& \textbf{in } \mathsf{reduce}\ (\oplus', \otimes')\ x_1'''' \oplus' \mathsf{reduce}\ (\oplus', \otimes')\ x_2''''
\end{array}
$$

$$
\begin{array}{ll}
= & \quad \{ \text{ Combining } x_1 \text{ and } x_2 \text{ by } \ominus \ \} \\
& \textbf{let } (x'_1 \ominus x'_2) = \mathsf{map}\ \ q\ (x_1 \ominus x_2) \\
& \quad\ (x''_1 \ominus x''_2) = \mathsf{scan}\ \ (\oplus, \otimes)\ (((|e| \phi\ u) \ominus (v_1 \phi\ x'_1)) \ominus (v_2 \phi\ x'_2)) \\
& \quad\ (x'''_1 \ominus x'''_2) = \mathit{drop}_c\ 1\ (\mathit{drop}_r\ 1\ (x''_1 \ominus x''_2)) \\
& \quad\ (x''''_1 \ominus x''''_2) = \mathsf{zipwith}\ \ p\ (x_1 \ominus x_2)\ (x'''_1 \ominus x'''_2) \\
& \textbf{in } \ \mathsf{reduce}\ \ (\oplus', \otimes')\ (x''''_1 \ominus x''''_2) \\
= & \quad \{ \text{ Renaming internal variables } \ \} \\
& \textbf{let } x' = \mathsf{map}\ \ q\ (x_1 \ominus x_2) \\
& \quad\ x'' = \mathsf{scan}\ \ (\oplus, \otimes)\ (((|e| \phi\ u) \ominus ((v_1 \ominus v_2) \phi\ x'))) \\
& \quad\ x''' = \mathit{drop}_c\ 1\ (\mathit{drop}_r\ 1\ x'') \\
& \quad\ x'''' = \mathsf{zipwith}\ \ p\ (x_1 \ominus x_2)\ x''' \\
& \textbf{in } \ \mathsf{reduce}\ \ (\oplus', \otimes')\ x'''' \\
= & \quad \{ \text{ The right hand side } \} \\
& \text{RHS } (x_1 \ominus x_2)\ (e, u, v_1 \ominus v_2)
\end{array}
$$

The induction case for ϕ is proved similarly.

To complete the above proof, we prove the following property of scan and edges. This property says that the bottom of the result of scan to an array is equivalent to that of scan applied to the row vector resulting by collapsing each column of the array.

$$
\mathit{bottom}\ (\mathsf{scan}\ \ (\oplus, \otimes)\ x) = \mathsf{scan}\ \ (\oplus, \otimes)\ (\mathsf{reduce}_c\ \ (\oplus)\ x)
$$

The following instantiation of this property completes the above proof.

$$
\begin{array}{l}
\mathit{bottom}\ (\mathsf{scan}\ \ (\oplus, \otimes)\ ((|e| \phi\ u) \ominus (v_1 \phi\ x'_1))) = \mathsf{scan}\ \ (\oplus, \otimes)\ (|e_2| \phi\ u_2) \\
\textbf{where } \ (|e_2| \phi\ u_2) = (e \oplus \mathsf{reduce}\ \ (\oplus, \otimes) v_1) \phi\ (\mathsf{zipwith}\ \ (\oplus)\ u\ (\mathsf{reduce}_c\ \ (\oplus)\ x'_1)) \\
\qquad\qquad\qquad\quad = \mathsf{reduce}_c\ \ (\oplus)\ ((|e| \phi\ u) \ominus (v_1 \phi\ x'_1))
\end{array}
$$

The property is proved by induction on the structure of abide-trees. Base case is proved as follows.

$$
\begin{array}{ll}
& \mathit{bottom}\ (\mathsf{scan}\ \ (\oplus, \otimes)\ |a|) \\
= & \quad \{ \text{ Definition of } \mathsf{scan} \ \} \\
& \mathit{bottom}\ |a| \\
= & \quad \{ \text{ Definition of } \mathit{bottom} \ \} \\
& |a| \\
= & \quad \{ \text{ Definition of } \mathsf{scan} \ \} \\
& \mathsf{scan}\ \ (\oplus, \otimes)\ |a| \\
= & \quad \{ \text{ Definition of } \mathsf{reduce}_c \ \} \\
& \mathsf{scan}\ \ (\oplus, \otimes)\ (\mathsf{reduce}_c\ \ (\oplus)\ |a|)
\end{array}
$$

Inductive case for ϕ is proved as follows.

$$
\begin{array}{ll}
 & \mathit{bottom}\ (\mathsf{scan}\ (\oplus,\otimes)\ (x_1 \phi x_2)) \\
= & \{\ \text{Definition of } \mathsf{scan}\ \} \\
 & \mathit{bottom}\ (\mathsf{scan}\ (\oplus,\otimes)\ x_1 \\
 & \qquad \phi\ \mathsf{map}_c\ (\mathsf{zipwith}(\otimes)\ (\mathit{right}\ (\mathsf{scan}\ (\oplus,\otimes)\ x_1)))\ (\mathsf{scan}\ (\oplus,\otimes)\ x_2)) \\
= & \{\ \mathit{bottom} \text{ distributes over } \phi\ \} \\
 & \mathit{bottom}\ (\mathsf{scan}\ (\oplus,\otimes)\ x_1) \\
 & \qquad \phi\, \mathit{bottom}\ (\mathsf{map}_c\ (\mathsf{zipwith}(\otimes)\ (\mathit{right}\ (\mathsf{scan}\ (\oplus,\otimes)\ x_1)))\ (\mathsf{scan}\ (\oplus,\otimes)\ x_2)) \\
= & \{\ \text{Promote } \mathit{bottom} \text{ through } \mathsf{map}_c\ \} \\
 & \mathit{bottom}\ (\mathsf{scan}\ (\oplus,\otimes)\ x_1) \\
 & \phi\ \mathsf{map}_c\,(\mathsf{zipwith}(\otimes)\,(\mathit{bottom}\,(\mathit{right}\,(\mathsf{scan}\,(\oplus,\otimes)\ x_1))))\,(\mathit{bottom}\,(\mathsf{scan}\,(\oplus,\otimes)\,x_2))) \\
= & \{\ \mathit{bottom} \circ \mathit{right} = \mathit{right} \circ \mathit{bottom}\ \} \\
 & \mathit{bottom}\ (\mathsf{scan}\ (\oplus,\otimes)\ x_1) \\
 & \phi\ \mathsf{map}_c\,(\mathsf{zipwith}(\otimes)\,(\mathit{right}\,(\mathit{bottom}\,(\mathsf{scan}\,(\oplus,\otimes)\,x_1))))\,(\mathit{bottom}\,(\mathsf{scan}\,(\oplus,\otimes)\,x_2))) \\
= & \{\ \text{Hypothesis of induction}\ \} \\
 & \mathsf{scan}\ (\oplus,\otimes)\ (\mathsf{reduce}_c\ (\oplus)\ x_1) \\
 & \qquad \phi\ \mathsf{map}_c\ (\mathsf{zipwith}(\otimes)\ (\mathit{right}\ (\mathsf{scan}\ (\oplus,\otimes)\ (\mathsf{reduce}_c\ (\oplus)\ x_1)))) \\
 & \qquad\qquad (\mathsf{scan}\ (\oplus,\otimes)\ (\mathsf{reduce}_c\ (\oplus)\ x_2)) \\
= & \{\ \text{Definition of } \mathsf{scan}\ \} \\
 & \mathsf{scan}\ (\oplus,\otimes)\ (\mathsf{reduce}_c\ (\oplus)\ x_1\ \phi\ \mathsf{reduce}_c\ (\oplus)\ x_2) \\
= & \{\ \text{Definition of } \mathsf{reduce}_c\ \} \\
 & \mathsf{scan}\ (\oplus,\otimes)\ (\mathsf{reduce}_c\ (\oplus)\ (x_1 \phi x_2))
\end{array}
$$

Inductive case for $\ominus$ is proved as follows.

$$
\begin{array}{ll}
 & \mathit{bottom}\ (\mathsf{scan}\ (\oplus,\otimes)\ (x_1 \ominus x_2)) \\
= & \{\ \text{Definition of } \mathsf{scan}\ \} \\
 & \mathit{bottom}\ (\mathsf{scan}\ (\oplus,\otimes)\ x_1 \\
 & \qquad \ominus\, \mathsf{map}_r\ (\mathsf{zipwith}(\oplus)\ (\mathit{bottom}\ (\mathsf{scan}\ (\oplus,\otimes)\ x_1)))\ (\mathsf{scan}\ (\oplus,\otimes)\ x_2)) \\
= & \{\ \mathit{bottom} \text{ eliminates the upper subarray of } \ominus\ \} \\
 & \mathit{bottom}\ (\mathsf{map}_r\ (\mathsf{zipwith}(\oplus)\ (\mathit{bottom}\ (\mathsf{scan}\ (\oplus,\otimes)\ x_1)))\ (\mathsf{scan}\ (\oplus,\otimes)\ x_2)) \\
= & \{\ \text{Promote } \mathit{bottom} \text{ through } \mathsf{map}_r\ \} \\
 & \mathsf{map}_r\ (\mathsf{zipwith}(\oplus)\ (\mathit{bottom}\ (\mathsf{scan}\ (\oplus,\otimes)\ x_1)))\ (\mathit{bottom}\ (\mathsf{scan}\ (\oplus,\otimes)\ x_2)) \\
= & \{\ \text{Application of } \mathsf{map}_r\ f \text{ to a row vector is application of } f \text{ to the vector}\ \} \\
 & \mathsf{zipwith}(\oplus)\ (\mathit{bottom}\ (\mathsf{scan}\ (\oplus,\otimes)\ x_1))\ (\mathit{bottom}\ (\mathsf{scan}\ (\oplus,\otimes)\ x_2)) \\
= & \{\ \text{Hypothesis of induction}\ \} \\
 & \mathsf{zipwith}(\oplus)\ (\mathsf{scan}\ (\oplus,\otimes)\ (\mathsf{reduce}_c\ (\oplus)\ x_1))\ (\mathsf{scan}\ (\oplus,\otimes)\ (\mathsf{reduce}_c\ (\oplus)\ x_2)) \\
= & \{\ \text{Promotion of } \mathsf{zipwith}(\oplus) \text{ through } \mathsf{scan}\ (\oplus,\otimes) \text{ to row vectors}\ \} \\
 & \mathsf{scan}\ (\oplus,\otimes)\ (\mathsf{zipwith}(\oplus)\ (\mathsf{reduce}_c\ (\oplus)\ x_1)\ (\mathsf{reduce}_c\ (\oplus)\ x_2)) \\
= & \{\ \text{Definition of } \mathsf{reduce}_c\ \} \\
 & \mathsf{scan}\ (\oplus,\otimes)\ (\mathsf{reduce}_c\ (\oplus)\ (x_1 \ominus x_2))
\end{array}
$$

$\square$

This theorem says that we have an efficient parallel implementation of accumulate if the other skeletons have efficient parallel implementations. Moreover, we can write map, reduce, scan in accumulate as follows.

$$
\begin{array}{lcl}
\mathsf{map}\ f\ x & = & [\![(\ominus, \phi, \lambda a.\lambda e.(|\cdot| \circ f)\ a), (_, _, _)]\!]\ x\ _ \\
\mathsf{reduce}\ (\oplus)\ (\otimes)\ x & = & [\![(\oplus, \otimes, \lambda a.\lambda e.a), (_, _, _)]\!]\ x\ _ \\
\mathsf{scan}\ (\oplus)\ (\otimes)\ x & = & [\![(\ominus, \phi, \lambda a.\lambda e.|\cdot|\ e), (\oplus, \otimes, id)]\!]\ x\ e \\
 & & \mathbf{where}\ \ e = (\imath, [\imath, \ldots, \imath], [\imath, \ldots, \imath]) \\
 & & \qquad\quad \imath \oplus a = a, \imath \otimes a = a
\end{array}
$$

Here, '_' denotes "don't care". This property is sometimes useful to develop general theory for manipulating skeletons.

Relationship between Directed and Non-directed Skeletons

It should be noted that reduce can be expressed by its directed versions reduce_c and reduce_r, when two binary operators $\oplus$ and $\otimes$ are abiding.

$$
\begin{array}{lcl}
\mathsf{reduce}\ (\oplus, \otimes) & = & \mathsf{the} \circ \mathsf{reduce}_c\ (\oplus) \circ \mathsf{reduce}_r\ (\otimes) \\
\mathsf{reduce}\ (\oplus, \otimes) & = & \mathsf{the} \circ \mathsf{reduce}_r\ (\otimes) \circ \mathsf{reduce}_c\ (\oplus)
\end{array} \tag{3.2}
$$

Like reduce, we may define $\mathsf{scan}_\downarrow$ and $\mathsf{scan}_\rightarrow$ that are specialization of scan and scan a two-dimensional array in column and row direction respectively:

$$
\begin{array}{lcl}
\mathsf{scan}_\downarrow\ (\oplus) & = & \mathsf{scan}\ (\oplus, \gg) \\
\mathsf{scan}_\rightarrow\ (\otimes) & = & \mathsf{scan}\ (\gg, \otimes)\ ;
\end{array}
$$

scan can be expressed by $\mathsf{scan}_\downarrow$ and $\mathsf{scan}_\rightarrow$ when two binary operators $\oplus$ and $\otimes$ are abiding.

$$
\begin{array}{lcl}
\mathsf{scan}\ (\oplus, \otimes) & = & \mathsf{scan}_\downarrow\ (\oplus) \circ \mathsf{scan}_\rightarrow\ (\otimes) \\
\mathsf{scan}\ (\oplus, \otimes) & = & \mathsf{scan}_\rightarrow\ (\otimes) \circ \mathsf{scan}_\downarrow\ (\oplus)
\end{array} \tag{3.3}
$$

3.2.2 Example Complex Programs Described with Parallel Skeletons

In this section, we will show two programs written with parallel skeletons defined so far. One example is matrix multiplication, which will be fully described with the skeletons. The other example is QR factorization, which will be partially described with the skeletons, and the whole computation will be defined as a mutual recursive function. The important point here is that we do not necessarily need to describe the whole computation with skeletons to get parallel programs.

Matrix Multiplication

As an involved example, we will describe a parallel algorithm for matrix multiplication, which is a primitive operation of matrices, with the defined parallel skeletons on two-dimensional arrays.

The following program is an intuitively-understandable description of matrix multiplication, in which an element of the resulting matrix is an inner product of a row vector and a column vector of the input matrices. This program simply implements the definition of matrix multiplication. Although users do not need to consider parallelism at all, this program can be executed in parallel due to parallelism of each skeleton.

$$
\begin{aligned}
mm \;&=\; \mathsf{zipwith}_P\ iprod \circ (allrows \times allcols) \\
&\quad \mathbf{where} \\
&\qquad allrows \;=\; allred_r(\phi) \circ \mathsf{map}\,|\cdot| \\
&\qquad allcols \;=\; allred_c(\ominus) \circ \mathsf{map}\,|\cdot| \\
&\qquad iprod \;=\; (\mathsf{reduce}\ \ (+,+)\circ) \circ \mathsf{zipwith}(\times) \circ tr \\
&\qquad \mathsf{zipwith}_P\ (\otimes)\ (x,y) = \mathsf{zipwith}\ (\otimes)\ x\ y
\end{aligned}
$$

Although this definition seems to use $\mathrm{O}(n^3)$ memory space for $n \times n$ matrices due to duplications of rows and columns with $allred_r$ and $allred_c$, we can execute this multiplication using $\mathrm{O}(n^2)$ memory space. This is because we can use references instead of duplications in the implementation of $allred_r$ and $allred_c$ due to the properties of the operators ($\gg$ and $\ll$) in their definitions. This kind of optimization is currently done by hand. However, we think it will be automatically done by compilers or libraries when they take the properties of the operators into account.

Since the program is written with skeletons, we can easily generalize the computation of matrix multiplication as follows.

$$
\begin{aligned}
gemm\ (\oplus,\otimes) \;&=\; \mathsf{zipwith}_P\ iprod \circ (allrows \otimes allcols) \\
&\quad \mathbf{where} \\
&\qquad allrows \;=\; allred_r(\phi) \circ \mathsf{map}\,|\cdot| \\
&\qquad allcols \;=\; allred_c(\ominus) \circ \mathsf{map}\,|\cdot| \\
&\qquad iprod \;=\; (\mathsf{reduce}\ \ (\oplus,\oplus)\circ) \circ \mathsf{zipwith}(\otimes) \circ tr \\
&\qquad \mathsf{zipwith}_P\ (\otimes)\ (x,y) = \mathsf{zipwith}\ (\otimes)\ x\ y
\end{aligned}
$$

The generalized matrix multiplication $gemm\ (\oplus,\otimes)$ computes the multiplication with the given two operators $\oplus$ and $\otimes$ instead of the usual operators $+$ and $\times$, respectively. For example, the generalized matrix multiplication is useful to compute the distance matrices with the maximum operator and the plus operator; the shortest one-hop distances between points are computed by multiplication of matrices of which (i,j) element is the direct distance between point i and point j. Other application of the generalized matrix multiplication will be found in Chapter 4.3.

QR Factorization

As the other nontrivial example, we show descriptions of two parallel algorithms for QR factorization [EGJK04]. Given a matrix A, QR factorization computes an orthogonal matrix Q and an upper-triangular matrix R such that $A = QR$. We will not explain the details, but we hope to show that these algorithms can be dealt with in our framework. The important point here is that the whole algorithm does not necessarily be able to be written with skeletons, but we can write some parts of the algorithm with skeletons, and the whole computation may be written as a recursive function. In this case, we can do parallel computation in both those parts described with skeletons and independent recursive calls in the recursive definition of the algorithm.

We give the recursive description of a QR factorization algorithm based on Householder transform. This function returns Q and R which satisfy $A = QR$ where A is a matrix of $m \times n$, Q an orthogonal matrix of $m \times m$ and R an upper-triangular matrix of $m \times n$.

$$
\begin{array}{lcl}
qr\ ((A_{11} \ominus A_{21}) \phi\ (A_{12} \ominus A_{22})) & & \\
 & = & \textbf{let}\ (Q_1, R_{11} \ominus O) = qr\ (A_{11} \ominus A_{21}) \\
 & & \quad\ (R_{12} \ominus \hat{A_{22}}) = mm\ (trQ_1)\ (A_{12} \ominus A_{22}) \\
 & & \quad\ (\hat{Q_2}, R_{22}) = qr\ \hat{A_{22}} \\
 & & \quad\ Q = mm\ Q_1\ ((I \phi\ O) \ominus (O \phi\ \hat{Q_2})) \\
 & & \textbf{in}\ (Q, (R_{11} \phi\ R_{12}) \ominus (O \phi\ R_{22})) \\
qr\ (|a| \ominus x) & = & hh\ (|a| \ominus x) \\
hh\ v & = & \textbf{let}\ v' = madd\ v\ e \\
 & & \quad\ a = \sqrt{\mathsf{reduce}\ (+,+)\ (\mathsf{zipwith}(\times)\ v'\ v')} \\
 & & \quad\ u = \mathsf{map}\ (/a)\ v' \\
 & & \quad\ Q = msub\ I\ (\mathsf{map}\ (\times 2)\ (mm\ u\ (tr\ u))) \\
 & & \textbf{in}\ (Q, e)
\end{array}
$$

Here e is a vector (a matrix of which width is 1) whose first element is 1 and the other elements are 0, and I and O represent an identity matrix and a zero matrix of suitable size respectively.

Furthermore, we give the recursive description of QR factorization algorithm on quadtree [FW03]; describing algorithms on quadtrees to those on abide-trees is always possible because abide-trees is more flexible than quadtrees, although we need to impose some restriction on division of the argument arrays. This function qr_q is mutual recursively defined with an extra function e, and returns Q and R that satisfy $A = QR$ where A is a matrix of $n \times n$ ($n = 2^k$ for a natural number k), Q is

an orthogonal matrix of $n \times n$ and R is an upper-triangular matrix of $n \times n$.

$$
\begin{array}{lll}
qr_q\ |a| & = & (|1|, |a|) \\
\multicolumn{3}{l}{qr_q\ ((A_{11} \ominus A_{21}) \phi (A_{12} \ominus A_{22}))} \\
& = & \textbf{let}\ (Q_1, R_1) = qr_q\ A_{11} \\
& & \quad\ (Q_2, R_2) = qr_q\ A_{21} \\
& & \quad\ Q_{12} = (Q_1 \phi O) \ominus (O \phi Q_2) \\
& & \quad\ (Q_3, R_3) = e\ (R_1, R_2) \\
& & \quad\ Q_4 = mm\ Q_{12}\ Q_3 \\
& & \quad\ (U_n \ominus U_s) = mm\ (tr\ Q_4)\ (A_{12} \ominus A_{22}) \\
& & \quad\ (Q_6, R_6) = qr_q\ U_s \\
& & \quad\ Q = mm\ Q_4\ ((I \phi O) \ominus (O \phi Q_6)) \\
& & \quad\ R = (R_3 \phi U_n) \ominus (O \phi R_6) \\
& & \textbf{in}\ (Q, R)
\end{array}
$$

Note that A_{ij} $(i, j \in \{1, 2\})$ have the same shape. A definition of the involved extra function e is as follows.

$$
\begin{array}{lll}
e\ (N, O) & = & (I, N) \\
e\ (|n|, |s|) & = & \textbf{let}\ Q = g(n, s) \\
& & \quad\ (N, O) = mm(tr\ Q)\ (|n| \ominus |s|) \\
& & \textbf{in}\ (Q, N) \\
\multicolumn{3}{l}{e\ ((N_{11} \ominus N_{21}) \phi (N_{12} \ominus N_{22}), (S_{11} \ominus S_{21}) \phi (S_{12} \ominus S_{22}))} \\
\multicolumn{3}{l}{\quad = \textbf{let}} \\
\multicolumn{3}{l}{\qquad\quad ((Q_1^{11} \ominus Q_1^{21}) \phi (Q_1^{12} \ominus Q_1^{22}), N_1) = e\ (N_{11}, S_{11})} \\
\multicolumn{3}{l}{\qquad\quad ((Q_2^{11} \ominus Q_2^{21}) \phi (Q_2^{12} \ominus Q_2^{22}), N_2) = e\ (N_{22}, S_{22})} \\
\multicolumn{3}{l}{\qquad\quad Q_{12} = (Q_1^{11} \phi O \phi Q_1^{12} \phi O) \ominus (O \ominus Q_2^{11} \phi O \phi Q_2^{12})} \\
\multicolumn{3}{l}{\qquad\qquad\quad \ominus (Q_1^{21} \phi O \phi Q_1^{22} \phi O) \ominus (O \ominus Q_2^{21} \phi O \phi Q_2^{22})} \\
\multicolumn{3}{l}{\qquad\quad Q_1 = (Q_1^{11} \ominus Q_1^{21}) \phi (Q_1^{12} \ominus Q_1^{22})} \\
\multicolumn{3}{l}{\qquad\quad (U_n \ominus U_s) = mm\ (tr\ Q_1)\ (N_{12} \ominus S_{12})} \\
\multicolumn{3}{l}{\qquad\quad (Q_4, R_4) = qr_q\ U_s} \\
\multicolumn{3}{l}{\qquad\quad Q_4' = (I \phi O \phi O \phi O) \ominus (O \phi I \phi O \phi O)} \\
\multicolumn{3}{l}{\qquad\qquad\qquad \ominus (O \phi O \phi Q_4 \phi O) \ominus (O \phi O \phi O \phi I)} \\
\multicolumn{3}{l}{\qquad\quad Q_5 = mm\ Q_{12}\ Q4'} \\
\multicolumn{3}{l}{\qquad\quad ((Q_6^{11} \ominus Q_6^{21}) \phi (Q_6^{12} \ominus Q_6^{22}), N_6) = e\ (N_2, R_4)} \\
\multicolumn{3}{l}{\qquad\quad Q_6' = (I \phi O \phi O \phi O) \ominus (O \phi Q_6^{11} \phi Q_6^{12} \phi O)} \\
\multicolumn{3}{l}{\qquad\qquad\qquad \ominus (O \phi Q_6^{21} \phi Q_6^{22} \phi O) \ominus (O \phi O \phi O \phi I)} \\
\multicolumn{3}{l}{\qquad \textbf{in}\ (mm\ Q_5\ Q_6', (N_1 \phi U_n) \ominus (O \phi N_6))} \\
g\ (a, b) & = & (|c| \phi |s|) \ominus (|-s| \phi |c|) \\
& & \textbf{where}\ c = \dfrac{a}{\sqrt{a^2 + b^2}},\ s = \dfrac{-b}{\sqrt{a^2 + b^2}}
\end{array}
$$

Note that N_{ij} and S_{ij} $(i, j \in \{1, 2\})$ have the same shape and Q_k^{ij} $(i, j, k \in \{1, 2\})$ have the same shape.

We can efficiently parallelize some parts of these complicated recursive functions, such as matrix multiplication and independent calculations like $(Q_1, R_1) = qr_q\ A_{11}$ and $(Q_2, R_2) = qr_q\ A_{21}$. These independent calculations are explicitly parallelized by describing them with the map skeleton as follows.

$$\left\{ \begin{array}{l} (Q_1, R_1) = qr_q\ A_{11} \\ (Q_2, R_2) = qr_q\ A_{21} \end{array} \right\}$$

$$\Downarrow$$

$$|(Q_1, R_1)| \mathbin{\ominus} |(Q_2, R_2)| = \mathsf{map}\ \ apply\ (|(qr_q, A_{11})| \mathbin{\ominus} |(qr_q, A_{21})|)$$

Here, the function *apply* is defined as $apply\ (f, x) = f\ x$. It is, however, still an open problem whether the complicated recursive functions can be parallelized with our defined skeletons.

3.3 Homomorphism-based Parallel Skeletons for Trees

We will introduce the following five binary-tree skeletons, which are first proposed by Skillicorn [Ski96]. These skeletons are basic primitives in the parallel computation for trees.

- Two node-wise computations: $\mathsf{map_b}$ and $\mathsf{zipwith_b}$
- Two bottom-up computations: $\mathsf{reduce_b}$ and $\mathsf{uAcc_b}$ (upwards accumulate)
- One top-down computation: $\mathsf{dAcc_b}$ (downwards accumulate)

In the following, we will give the formal denotational definition of the skeletons as sequential recursive functions, and show the intuitive meaning of the basic binary-tree skeletons together with additional conditions for parallel implementation [Mat07].

Skeletons for node-wise computations

The skeletons $\mathsf{map_b}$ and $\mathsf{zipwith_b}$ for node-wise computations on trees are defined as follows.

$$\begin{array}{lcl} \mathsf{map_b}\ k_l\ k_n\ (\mathsf{BLeaf}\ n) & = & \mathsf{BLeaf}\ (k_l\ n) \\ \mathsf{map_b}\ k_l\ k_n\ (\mathsf{BNode}\ l\ n\ r) & = & \mathsf{BNode}\ (\mathsf{map_b}\ k_l\ k_n\ l)\ (k_n\ n)\ (\mathsf{map_b}\ k_l\ k_n\ r) \end{array}$$

$$\begin{array}{l} \mathsf{zipwith_b}\ k_l\ k_n\ (\mathsf{BLeaf}\ n)\ (\mathsf{BLeaf}\ n') = \mathsf{BLeaf}\ (k_l\ n\ n') \\ \mathsf{zipwith_b}\ k_l\ k_n\ (\mathsf{BNode}\ l\ n\ r)\ (\mathsf{BNode}\ l'\ n'\ r') \\ \qquad = \mathsf{BNode}\ (\mathsf{zipwith_b}\ k_l\ k_n\ l\ l')\ (k_n\ n\ n')\ (\mathsf{zipwith_b}\ k_l\ k_n\ r\ r') \end{array}$$

The parallel skeleton map_b takes two functions k_l and k_n and a binary tree, and applies k_l to each leaf and k_n to each internal node. The parallel skeleton $\mathsf{zipwith}_\mathrm{b}$ takes two functions k_l and k_n and two binary trees of the same shape, and zips the trees up by applying k_l to each corresponding pair of leaves and k_n to each corresponding pair of internal nodes. Since the functions are applied independently to the nodes, these two skeletons require no additional condition for their parallel implementation.

The map_b skeleton can be defined in the form of tree homomorphism as follows.

$$\begin{aligned} \mathsf{map}_\mathrm{b}\ k_l\ k_n &= (\![k_l', k_n']\!)_\mathrm{b} \\ &\quad \textbf{where}\ k_l'\ a = \mathsf{BLeaf}\ (k_l\ a) \\ &\quad\quad\quad\ \ k_n'\ l\ b\ r = \mathsf{BNode}\ l\ (k_n\ b)\ r \end{aligned}$$

The $\mathsf{zipwith}_\mathrm{b}$ skeleton cannot be defined as a tree homomorphism, but we can formalize it with tree anamorphism [MFP91].

Skeletons for bottom-up computations

The skeletons $\mathsf{reduce}_\mathrm{b}$ and uAcc_b for bottom-up computations on trees are defined as follows.

$$\begin{aligned} \mathsf{reduce}_\mathrm{b}\ k\ (\mathsf{BLeaf}\ n) &= n \\ \mathsf{reduce}_\mathrm{b}\ k\ (\mathsf{BNode}\ l\ n\ r) &= k\ (\mathsf{reduce}_\mathrm{b}\ k\ l)\ n\ (\mathsf{reduce}_\mathrm{b}\ k\ r) \\ \\ \mathsf{uAcc}_\mathrm{b}\ k\ (\mathsf{BLeaf}\ n) &= \mathsf{BLeaf}\ n \\ \mathsf{uAcc}_\mathrm{b}\ k\ (\mathsf{BNode}\ l\ n\ r) &= \textbf{let}\ l' = \mathsf{uAcc}_\mathrm{b}\ k\ l \\ &\quad\quad\ \ r' = \mathsf{uAcc}_\mathrm{b}\ k\ r \\ &\quad \textbf{in}\ \mathsf{BNode}\ l'\ (k\ (root_\mathrm{b}\ l')\ n\ (root_\mathrm{b}\ r'))\ r' \end{aligned}$$

The parallel skeleton $\mathsf{reduce}_\mathrm{b}$ takes a function k and a binary tree, and collapses the tree into a value by applying the function k to each internal node in a bottom-up manner. The parallel skeleton uAcc_b takes a function k and a binary tree, and applies the function k in a bottom-up manner while putting the intermediate result on each node. These two skeletons require an additional condition for the existence of an efficient parallel implementation of them. The skeletons $\mathsf{reduce}_\mathrm{b}$ and uAcc_b called with parameter function k require the existence of four auxiliary functions ϕ, ψ_n, ψ_l, and ψ_r satisfying the following three equations.

$$\begin{aligned} k\ x\ n\ y &= \psi_n\ x\ (\phi\ n)\ y \\ \psi_n\ (\psi_n\ n'\ x\ y)\ n\ r &= \psi_n\ x\ (\psi_l\ n'\ n\ r)\ y \\ \psi_n\ l\ n\ (\psi_n\ n'\ x\ y) &= \psi_n\ x\ (\psi_r\ l\ n\ n')\ y \end{aligned}$$

The $\mathsf{reduce}_\mathrm{b}$ and uAcc_b skeletons can be defined in the form of tree homomor-

phism as follows.

$$\begin{array}{lcl} \mathsf{reduce}_\mathrm{b}\ k & = & (\![id, k]\!)_\mathrm{b} \\ \mathsf{uAcc}_\mathrm{b}\ k & = & (\![k'_l, k'_n]\!)_\mathrm{b} \\ & & \textbf{where}\ k'_l\ a \quad = \mathsf{BLeaf}\ a \\ & & \qquad\quad k'_n\ l\ b\ r = \mathsf{BNode}\ l\ (k\ (root_\mathrm{b}\ l)\ b\ (root_\mathrm{b}\ r))\ r \end{array}$$

Skeleton for top-down computations

The skeleton dAcc_b for top-down computations on trees is defined as follows.

$$\begin{array}{lcl} \mathsf{dAcc}_\mathrm{b}\ (g_l, g_r)\ c\ (\mathsf{BLeaf}\ n) & = & \mathsf{BLeaf}\ c \\ \mathsf{dAcc}_\mathrm{b}\ (g_l, g_r)\ c\ (\mathsf{BNode}\ l\ n\ r) & = & \mathsf{BNode}\ (\mathsf{dAcc}_\mathrm{b}\ (g_l, g_r)\ (g_l\ c\ b)\ l)\ c \\ & & \qquad (\mathsf{dAcc}_\mathrm{b}\ (g_l, g_r)\ (g_r\ c\ b)\ r) \end{array}$$

The parallel skeleton dAcc_b takes a pair of functions g_l and g_r, a parameter c and a binary tree. This skeleton updates parameter c in a top-down manner using g_l for the left child and g_r for the right child, and puts the parameter c on each node. The parameter c is called accumulative parameter. This skeleton also requires an additional condition for an efficient parallel implementation. The dAcc_b skeleton called with parameter functions g_l and g_r requires the existence of auxiliary functions ϕ_l, ϕ_r, ψ_u, and ψ_d satisfying the following three equations.

$$\begin{array}{lcl} g_l\ c\ n & = & \psi_d\ c\ (\phi_l\ n) \\ g_r\ c\ n & = & \psi_d\ c\ (\phi_r\ n) \\ \psi_d\ (\psi_d\ c\ n)\ m & = & \psi_d\ c\ (\psi_u\ n\ m) \end{array}$$

The dAcc_b skeleton can be defined as the following higher-order tree homomorphism.

$$\begin{array}{lcl} \mathsf{dAcc}_\mathrm{b}\ (g_l, g_r)\ c & = & \lambda t.(\![k'_l, k'_n]\!)_\mathrm{b}\ t\ c \\ & & \textbf{where}\ k'_l\ a \qquad\ = \lambda c'.\mathsf{BLeaf}\ c' \\ & & \qquad\quad k'_n\ f_l\ b\ f_r = \lambda c'.\mathsf{BNode}\ (f_l\ (g_l\ c'\ b))\ c'\ (f_r\ (g_r\ c'\ b)) \end{array}$$

Worth noting is the application scope of the binary-tree skeletons under the conditions for parallel implementations. Our conditions cover a wider class of tree manipulations than those introduced by Skillicorn [Ski94, Ski96] do. For algebraic tree computations our conditions cover the same class as those studied by Abrahamson et al. [ADKP89], even though they are specified in a different way.

Example programming with the tree skeletons can be found in [Mat07, Ski97], which includes XPath queries [BBC+06] and maximum marking problems [SHTO00, Bir01] on trees.

Implementation of Parallel Tree Skeletons

The defined parallel tree skeletons can be implement efficiently in parallel for many parallel architectures.

The binary tree skeletons have parallel implementation based on tree contraction algorithms [MR85, ADKP89, RT94, BSWB02, MW97, CV88, DH98, LM88]. Here, let the model of parallel computers be EREW PRAM with P processors, and N denote the number of nodes of a tree. We assume that all the functions including auxiliary functions passed to skeletons are computed sequentially in constant time, and that the conditions of the $\mathsf{reduce}_\mathrm{b}$, uAcc_b and dAcc_b are satisfied. The map_b and $\mathsf{zipwith}_\mathrm{b}$ skeletons can be implemented just by applying functions independently to each node, and they run in $\mathrm{O}(N/P)$ parallel time. The $\mathsf{reduce}_\mathrm{b}$ skeleton can be implemented by tree contraction algorithms and it runs in $\mathrm{O}(N/P + \log P)$ parallel time. The uAcc_b and dAcc_b skeletons are generalized computational patterns of the algebraic tree computations [ADKP89], and Gibbons et al. [GCS94] developed parallel implementations of them based on tree contraction algorithms. With their implementations, the uAcc_b and dAcc_b skeletons can be computed in $\mathrm{O}(N/P + \log P)$ parallel time. In other words, the map_b and $\mathsf{zipwith}_\mathrm{b}$ skeletons achieve linear speedup under $P \leq \mathrm{O}(N)$, and the $\mathsf{reduce}_\mathrm{b}$, uAcc_b, and dAcc_b skeletons achieve linear speedup under $P \leq \mathrm{O}(N/\log N)$.

Although the original tree contraction algorithms have been designed for shared-memory computers, Matsuzaki [Mat07] has adapted the algorithms for use in distributed parallel machines. With their implementation, the tree skeletons achieve linear speedup under $P \leq \mathrm{O}(\sqrt{N})$.

Please refer for the details to Matsuzaki's thesis [Mat07], because the implementation is too complicated to show here.

Chapter 4

Fusion Optimizations of Parallel Skeletons

It has been shown so far that compositions of homomorphisms and skeletons provide us with a powerful mechanism to describe parallel algorithms, where parallelism in the original parallel algorithms can be well captured. In this chapter, we move on from issues of parallelism to the issues of efficiency.

Since skeleton programs are developed in the compositional style, they often have overheads of redundantly many loops and unnecessary intermediate data. To make skeleton programs efficient, not only each skeleton is implemented efficiently in parallel, but also optimizations over skeleton compositions are necessary.

The inefficiency problem arises also in sequential programming of the compositional style, and much research has been done deeply to solve the problem in the field of functional programming. One well known technique to solve the problem is fusion optimization [Wad88, GLJ93, TM95, Sve02], in which consecutive functions are fused into one function to remove redundant generation of intermediate data structures between them.

In this chapter, we will study domain-independent fusion optimizations to improve efficiency of skeleton compositions. Fusion of consecutive skeletons is important and effective, since it can remove redundant generation of intermediate data structures. It is especially effective in the case that elements of the input arrays have some structures, such as nested lists and arrays, because the overhead of allocation and deallocation of elements becomes serious and sometimes it becomes the main reason of inefficiency.

First, we will study fusion optimization of list skeletons, in which the good fusibility of the homomorphism plays an important role. Then, we will proceed to fusion optimization of skeletons on two-dimensional arrays. After that, using the fusion optimizations, we will show derivation of non-trivial efficient program for the maximum rectangle sum problem on two-dimensional arrays. A general derivation strategy with fusions will be given there.

4.1 Useful Fusion Laws for Skeletons on Lists

We will introduce several interesting algebraic laws for fusion of list skeletons. We will first introduce an important law of homomorphism. Then, we will derive interesting laws for skeletons from the law of homomorphism.

4.1.1 Fusion of List Homomorphism

Homomorphism enjoys many nice transformation rules, among which the following fusion rule is of particular importance. The fusion rule gives us a way to create a new homomorphism from composition of a function and a homomorphism, which plays a key role in derivation of efficient parallel programs.

Lemma 4.1 (Fusion law of list homomorphism [Bir88])**.** Let g and $(\![f, \oplus]\!)$ be given. If there exists $\odot$ such that for any x and y the equation

$$g\ (x \oplus y) \quad = \quad g\ x \odot g\ y$$

holds, then

$$g \circ (\![f, \oplus]\!) = (\![g \circ f, \odot]\!)\ .$$

Proof. The lemma is proved by induction on the structure of lists. □

Here, we will show an example fusion of the program $sum \circ double$, which doubles all elements of a list and sums them up. Its fusion is shown below.

$$\begin{array}{ll} & sum \circ double \\ = & \quad \{\ double \text{ is a homomorphism } \} \\ & sum \circ (\![[\cdot] \circ dbl, +\!\!+]\!) \\ = & \quad \{\ sum\ (x +\!\!+ y) = sum\ x + sum\ y, \text{ and the fusion lemma } \} \\ & (\![sum \circ [\cdot] \circ dbl, +]\!) \\ = & \quad \{\ sum \circ [\cdot] = id\ \} \\ & (\![dbl, +]\!) \end{array}$$

This fused function is efficient because it has no intermediate data, while the original program has intermediate data between two functions sum and $double$.

4.1.2 Fusion Laws for Skeletons

We will introduce some instances of the homomorphism fusion. Since the fusion theorem requires us to derive new operator satisfying the condition, automatic use of the theorem is difficult. Thus, we will instantiate the theorem for easy use of fusions.

We first show an interesting law that shows any homomorphism can be written as a composition of map and reduce. In other words, a composition of map and reduce can be fused into one homomorphism.

Lemma 4.2 (Homomorphism [Bir88])**.** A homomorphism $(\![f, \oplus]\!)$ can be written as a composition of map and reduce:

$$(\![f, \oplus]\!) = \mathsf{reduce}\ (\oplus) \circ \mathsf{map}\ f\ .$$

Proof. We can prove the lemma with Lemma 4.1, substituting $\mathsf{reduce}\ (\oplus)$ and $\mathsf{map}\ f = (\![[\cdot] \circ f, +\!\!+]\!)$ for the function g and the homomorphism $(\![f, \oplus]\!)$ in the lemma, respectively. □

This lemma implies that if we have efficient parallel implementations for reduce and map, we get an efficient implementation for homomorphism. Conversely, it implies that we can compute efficiently the composition of two skeletons if we have efficient implementation for homomorphism.

The following lemma eliminates intermediate data between two maps by fusing them into a single map.

Lemma 4.3 (Map-map fusion)**.** The following equation holds for any functions f and g.

$$\mathsf{map}\ g \circ \mathsf{map}\ f = \mathsf{map}\ (g \circ f)$$

Proof. We can prove the lemma with Lemma 4.1, substituting $\mathsf{map}\ g$ and $\mathsf{map}\ f = (\![[\cdot] \circ f, +\!\!+]\!)$ for the function g and the homomorphism $(\![f, \oplus]\!)$ in the lemma, respectively. □

This lemma is useful because we can fuse a sequence of maps into only one map; we can write a program step by step with many maps, and we can get efficient program by fusing them with the lemma.

4.1.3 Accumulate-buildJ Fusion

We will introduce another fusion based on a general form of skeletons on lists [HIT02], of which foundation is also the homomorphism. The general form is called $\mathsf{accumulate}$ The general form $\mathsf{accumulate}$ is equivalent to a specific composition of other skeletons, and thus can represent various computation of skeleton compositions. It is defined as follows.

$$\begin{array}{lcl} \mathsf{accumulate}\ (\oplus)\ p\ (\otimes)\ q\ g\ x\ c & = & \textbf{let}\ \ es +\!\!+ [e] = \mathsf{scan}\ \ (\otimes)\ ([c] +\!\!+ \mathsf{map}\ \ q\ x) \\ & & \qquad ac = \mathsf{zip}\ \ x\ \ es \\ & & \textbf{in}\ \ \mathsf{reduce}\ \ (\oplus)\ (\mathsf{map}\ \ p\ ac)\ \ \oplus g\ e \end{array}$$

We will denote the $\mathsf{accumulate}\ (\oplus)\ p\ (\otimes)\ q\ g$ with special brackets $[\![g, (p, \oplus), (q, \otimes)]\!]$.

The fusion using $\mathsf{accumulate}$ is based on the idea of short-cut fusions [Wad88, GLJ93], in which the generation of intermediate data structures between a producer and a consumer is canceled by fusing the producer and the consumer. We will

use the skeleton accumulate as a consumer, and the following function $\mathsf{build_J}$ as a producer.

$$\mathsf{build_J}\ gen = gen\ (+\!\!+)\ [\cdot]\ [\,]$$

We can express the list skeletons by using the functions defined above.

$$\begin{array}{l} \mathsf{map}\ f = \mathsf{build_J}\ (\lambda c\ s\ e.\ [\![\lambda_.e, (s \circ f \circ \pi_1, c), (_, _)]\!]) \\ \mathsf{reduce}\ (\oplus) = [\![\lambda_.\imath_\oplus, (\pi_1, \oplus), (_, _)]\!] \\ \mathsf{scan}\ (\oplus)\ x = \mathsf{build_J}\ (\lambda c\ s\ e.\ [\![s, (\lambda(a, e').\ s\ e', c), (id, \oplus)]\!])\ x\ \imath_\oplus \end{array}$$

The following theorem gives us the fusion rule to fuse the skeletons described with accumulate and $\mathsf{build_J}$.

Theorem 4.4 (BuildJ-accumulate-buildJ fusion)**.** Provided that function gen have the type forall $\alpha.(\alpha \to \alpha \to \alpha) \to (\beta \to \alpha) \to \alpha \to \alpha$, the following equation holds.

$$\begin{array}{l} \mathsf{build_J}\ (\lambda c\ s\ e.\ [\![g, (p, \oplus), (q, \otimes)]\!])\ (\mathsf{build_J}\ gen\ x)\ e \\ \quad = \pi_1\ (\mathsf{build_J}\ (\lambda c\ s\ e.\ gen\ (\odot)\ f\ d)\ x\ e) \\ \qquad \begin{array}{lcl} \textbf{where}\ (u \odot v)\ e & = & \textbf{let}\ (r_1, s_1, t_1) \ = \ u\ e \\ & & \phantom{\textbf{let}}\ (r_2, s_2, t_2) \ = \ v\ (e \otimes t_1) \\ & & \textbf{in}\ (s_1 \oplus r_2,\ s_1 \oplus s_2,\ t_1 \otimes t_2) \\ \phantom{\textbf{where}}\ f\ a\ e & = & (p\ (a, e) \oplus g\ (e \otimes q\ a), p\ (a, e), q\ a)) \\ \phantom{\textbf{where}}\ d\ e & = & (g\ e, _, _) \end{array} \end{array}$$

□

A variant of the above fusion rule is the following buildJ-cataJ-buildJ rule.

$$\mathsf{build_J}\,(\lambda c\,s\,e.\,[\![\lambda_.\phi_1, (\phi_2 \circ \pi_1, \phi_3), (_, _)]\!]) \circ (\mathsf{build_J}\,gen) = \mathsf{build_J}\,(\lambda c\,s\,e.gen\,\phi_1\,\phi_2\,\phi_3)$$

For example, we can fuse two maps into one map using the above rule as follows.

$$\begin{array}{cl} & \mathsf{map}\ f \circ \mathsf{map}\ g \\ = & \{\ \text{description of map with}\ \mathsf{build_J}\ \} \\ & \mathsf{build_J}\ (\lambda c\ s\ e.\ [\![\lambda_.e, (s \circ f \circ \pi_1, c), (_, _)]\!]) \\ & \quad \circ\, \mathsf{build_J}\ (\lambda c\ s\ e.\ [\![\lambda_.e, (s \circ g \circ \pi_1, c), (_, _)]\!]) \\ = & \{\ \text{the fusion rule}\ \} \\ & \mathsf{build_J}\ (\lambda c\ s\ e.\ [\![\lambda_.e, (s \circ f \circ g \circ \pi_1, c), (_, _)]\!]) \\ = & \{\ \text{description of map with}\ \mathsf{build_J}\ \} \\ & \mathsf{map}\ (f \circ g) \end{array}$$

4.2 Useful Fusion Laws for Skeletons on Two-dimensional Arrays

We will introduce several interesting algebraic laws for fusion of two-dimensional array skeletons. We will first introduce an important law of homomorphism. Then, we will derive interesting laws for skeletons from the law of homomorphism.

4.2.1 Fusion of Homomorphism for Two-dimensional Arrays

Homomorphism enjoys many nice transformation rules, among which the following fusion rule is of particular importance. The fusion rule gives us a way to create a new homomorphism from composition of a function and a homomorphism. As will be seen in Chapter 4.3, it plays a key role in derivation of efficient parallel programs on abide-trees.

Theorem 4.5 (Fusion law of abide-tree homomorphism)**.** Let g and $(\!|f, \oplus, \otimes|\!)$ be given. If there exist $\odot$ and $\ominus$ such that for any x and y the following equations

$$\begin{cases} g\ (x \oplus y) & = & g\ x \odot g\ y \\ g\ (x \otimes y) & = & g\ x \ominus g\ y \end{cases}$$

hold, then

$$g \circ (\!|f, \oplus, \otimes|\!) = (\!|g \circ f, \odot, \ominus|\!)\ .$$

Proof. The theorem is proved by induction on the structure of abide-trees.
The base case is shown below.

$$\begin{array}{ll} & (g \circ (\!|f, \oplus, \otimes|\!))\ |a| \\ = & \quad \{ \text{ Definition of } (\!|f, \oplus, \otimes|\!) \ \} \\ & g\ (f\ a) \\ = & \quad \{ \text{ Definition of } (\!|g \circ f, \odot, \ominus|\!) \ \} \\ & (\!|g \circ f, \odot, \ominus|\!)\ |a| \end{array}$$

Then, the inductive case for $-\!\!\!\circ\!\!\!-$ is shown as follows.

$$\begin{array}{ll} & (g \circ (\!|f, \oplus, \otimes|\!))\ (x -\!\!\!\circ\!\!\!- y) \\ = & \quad \{ \text{ Definition of } (\!|f, \oplus, \otimes|\!) \ \} \\ & g\ ((\!|f, \oplus, \otimes|\!)\ x \oplus (\!|f, \oplus, \otimes|\!)\ y) \\ = & \quad \{ \text{ Definition of } h \ \} \\ & g\ ((\!|f, \oplus, \otimes|\!)\ x) \odot g\ ((\!|f, \oplus, \otimes|\!)\ y) \\ = & \quad \{ \text{ Hypothesis of induction } \} \\ & (\!|g \circ f, \odot, \ominus|\!)\ x \odot (\!|g \circ f, \odot, \ominus|\!)\ y \\ = & \quad \{ \text{ Definition of } (\!|g \circ f, \odot, \ominus|\!) \ \} \\ & (\!|g \circ f, \odot, \ominus|\!)\ (x -\!\!\!\circ\!\!\!- y) \end{array}$$

The inductive case for ϕ is proved similarly. □

Here, we will show an example fusion of the program $sum \circ allsquare$, which squares all elements of the input and sums them up. Definitions of sum and $allsquare$ are given as follows: $sum = (\!|id, +, +|\!)$ and $allsquare = (\!|\,|\cdot| \circ sqr, -\!\!\!\circ\!\!\!-, \phi|\!)$, where

$sqr\ x = x^2$. Its fusion is shown below.

$$
\begin{array}{cl}
& sum \circ allsquare \\
= & \{\ allsquare \text{ is a homomorphism }\ \} \\
& sum \circ ([\,|\cdot| \circ sqr, \text{+\!\!\!\!-}, \phi\,]) \\
= & \left\{ \text{ the fusion with } \left\{ \begin{array}{lcl} sum\ (x \text{+\!\!\!\!-} y) & = & sum\ x + sum\ y \\ sum\ (x \phi\ y) & = & sum\ x + sum\ y \end{array} \right. \right\} \\
& ([\,sum \circ |\cdot| \circ sqr, +, +\,]) \\
= & \{\ sum \circ |\cdot| = id\ \} \\
& ([\,sqr, +, +\,])
\end{array}
$$

This fused function is efficient because it has no intermediate data, while the original program has intermediate data between two functions *sum* and *allsquare* .

4.2.2 Fusion Laws of Skeleton Compositions

Although the fusion law of homomorphism is very strong, there is a problem that we have to find the new operators $\odot$ and $\ominus$ for the given function g, and this is difficult in general. Therefore, we will derive more easy-to-use fusion laws for skeletons from the homomorphism's fusion law.

The first fusion law shows that any homomorphism can be written as a composition of map and reduce. In other words, the pair of map and reduce can be fused into a homomorphism.

Lemma 4.6 (Homomorphism)**.** A homomorphism $([f, \oplus, \otimes])$ can be written as a composition of map and reduce:

$$([f, \oplus, \otimes]) = \mathsf{reduce}\ \ (\oplus, \otimes) \circ \mathsf{map}\ \ f\ .$$

Proof. We can prove the lemma with Theorem 4.5, substituting reduce $(\oplus, \otimes)$ and map $f = ([\,|\cdot| \circ f, \text{+\!\!\!\!-}, \phi\,])$ for the function g and the homomorphism $([f, \oplus, \otimes])$ in the statement, respectively. □

The next lemma gives us a way to fuse consecutive maps into one map.

Lemma 4.7 (Map-map fusion)**.** For any functions f and g, two consecutive maps are fused into one map as follows.

$$\mathsf{map}\ \ g \circ \mathsf{map}\ \ f = \mathsf{map}\ \ (g \circ f)$$

Proof. We can prove the lemma with Theorem 4.5, substituting map g and map $f = ([\,|\cdot| \circ f, \text{+\!\!\!\!-}, \phi\,])$ for the function g and the homomorphism $([f, \oplus, \otimes])$ in the statement, respectively. □

Using the above two lemmas, we can optimize the following naively composed skeleton program for computing the second moment (variance) with the parameter c against an array. The definition of the second moment of an array $a_{(i,j)}$ is as follows: $\sum_{i,j}(a_{(i,j)} - c)^2$. Here, we can use $\mathsf{reduce}\ (+,+)$ to compute the sum, $\mathsf{map}\ \mathit{sqr}$ to take squares, and $\mathsf{map}\ (\mathit{subtract}\ c)$ to take the subtractions, where $\mathit{subtract}\ c\ x = x - c$, and $\mathit{sqr}\ x = x^2$.

$$\mathit{sndMom}\ c = \mathsf{reduce}\ (+,+) \circ \mathsf{map}\ \mathit{sqr} \circ \mathsf{map}\ (\mathit{subtract}\ c)$$

Now, we perform calculation with the above fusion lemmas.

$$\begin{array}{cl} & \mathit{sndMom}\ c \\ = & \{\ \text{Map-map fusion of Lemma 4.7}\ \} \\ & \mathsf{reduce}\ (+,+) \circ \mathsf{map}\ (\mathit{sqr} \circ \mathit{subtract}\ c) \\ = & \{\ \text{Map-reduce fusion of Lemma 4.6}\ \} \\ & (\![\mathit{sqr} \circ \mathit{subtract}\ c, +, +]\!) \end{array}$$

Now, if we have implementation of the homomorphism, we can compute the second moment without any intermediate data structure. Or, at least we can compute the second moment with only one intermediate data structure using implementations of reduce and map skeletons. In both cases, the fused program has less intermediate data structures and thus more efficient than the original naive program.

4.3 Developing Efficient Programs with Parallel Skeletons

In this section, we will illustrate a strategy to guide programmers to develop *efficient* parallel algorithms systematically through program transformation with the fusion laws. Recall homomorphisms have efficient parallel implementation. Thus, a goal of this derivation may be to write a program by a homomorphism. We will focus on developing parallel programs on two-dimensional arrays, although the strategy is applicable for other data structures.

Unfortunately, not all functions can be specified by a single homomorphism. Therefore, we first introduce a more powerful tool called almost-homomorphism [Col95]. After that, we will introduce two important theorems for deriving efficient programs, one of which is the special case of the fusion law of homomorphism. Then, we demonstrate the strategy with an example problem.

4.3.1 Almost-homomorphism

Not all functions can be specified by a single homomorphism, but we can always tuple these functions with some extra functions so that the tupled functions can be specified by a homomorphism. An *almost homomorphism* is a composition of a projection function and a homomorphism.

Definition 4.8 (Almost-homomorphism)**.** A function g is said to be almost-homomorphism, if g is defined as a composition of a homomorphism $(\![f, \oplus, \otimes]\!)$ and a projection function π_1 as follows.

$$g = \pi_1 \circ (\![f, \oplus, \otimes]\!)$$

□

Since projection functions are simple, almost homomorphisms are suitable for parallel computation as homomorphisms are.

In fact, every function can be represented in terms of an almost homomorphism at the cost of redundant computation. Let k be a nonhomomorphic function, and consider a new function g such that $g\ x = (k\ x, x)$. The tupled function g can be a homomorphism as follows.

$$\begin{array}{lcl} g\ |a| & = & (k\ |a|, |a|) \\ g\ (x \ominus y) & = & g\ x \oplus g\ y \\ & & \textbf{where}\ (k_1, x_1) \oplus (k_2, x_2) = (k\ (x_1 \ominus x_2), x_1 \ominus x_2) \\ g\ (x \obar y) & = & g\ x \otimes g\ y \\ & & \textbf{where}\ (k_1, x_1) \otimes (k_2, x_2) = (k\ (x_1 \obar x_2), x_1 \obar x_2) \end{array}$$

Since the first component of the result of g is the result of k, we have the following equation:

$$k = \pi_1 \circ g = \pi_1 \circ (\![g \circ |\cdot|, \oplus, \otimes]\!) \ .$$

This equation shows that k is an almost-homomorphism.

However, the definition above is not efficient because binary operators $\oplus$ and $\otimes$ do not use the previously computed values k_1 and k_2. In order to derive a good almost homomorphism, we should carefully define a suitable tupled function, making full use of previously computed values. We will see this in our parallel program development later. It is an open problem to determine the class of problems that have good (efficient) almost-homomorphic implementation.

4.3.2 Two Theorems for Deriving Efficient Parallel Programs

We will introduce two important theorems for deriving efficient parallel programs using almost-homomorphism.

First, we will propose a way of deriving almost homomorphism from mutual recursive definitions. For notational convenience, we define

$$\begin{array}{r} \bigtriangleup_1^n f_i = f_1 \bigtriangleup f_2 \bigtriangleup \cdots \bigtriangleup f_n \\ x(\bigtriangleup_1^n \oplus_i)y = (x \oplus_1 y, x \oplus_2 y, \ldots, x \oplus_n y) \ . \end{array}$$

Our main idea is based on the following theorem.

Theorem 4.9 (Tupling)**.**
Let $h_1, h_2, \ldots, h_n$ be mutual recursively defined by

$$\begin{cases} h_i\ |a| & = \ f_i\ a\ , \\ h_i\ (x \ominus y) & = \ ((\triangle_1^n h_i)\ x) \oplus_i ((\triangle_1^n h_i)\ y)\ , \\ h_i\ (x \phi y) & = \ ((\triangle_1^n h_i)\ x) \otimes_i ((\triangle_1^n h_i)\ y)\ . \end{cases} \tag{4.10}$$

Then $\triangle_1^n h_i$ is a homomorphism $(\![\ \triangle_1^n f_i, \triangle_1^n \oplus_i, \triangle_1^n \otimes_i\ ]\!)$.

Proof. The theorem is proved based on the definition of homomorphisms. According to the definition of array homomorphisms, it is sufficient to prove that

$$\begin{array}{ll} (\triangle_1^n h_i)\ |a| & = (\triangle_1^n f_i)\ a\ , \\ (\triangle_1^n h_i)\ (x \ominus y) & = ((\triangle_1^n h_i)\ x)\ (\triangle_1^n \oplus_i)\ ((\triangle_1^n h_i)\ y)\ , \\ (\triangle_1^n h_i)\ (x \phi y) & = ((\triangle_1^n h_i)\ x)\ (\triangle_1^n \otimes_i)\ ((\triangle_1^n h_i)\ y)\ . \end{array}$$

The first equation is proved by the following calculation.

$$\begin{array}{ll} & (\triangle_1^n h_i)\ |a| \\ = & \qquad \{\ \text{Definition of } \triangle\ \} \\ & (h_1\ |a|, \ldots, h_n\ |a|) \\ = & \qquad \{\ \text{Definition of } h_i\ \} \\ & (f_1\ a, \ldots, f_n\ a) \\ = & \qquad \{\ \text{Definition of } \triangle\ \} \\ & (\triangle_1^n f_i)\ a \end{array}$$

The second is proved as follows.

$$\begin{array}{ll} & (\triangle_1^n h_i)\ (x \ominus y) \\ = & \qquad \{\ \text{Definition of } \triangle\ \} \\ & (h_1\ (x \ominus y), \ldots, h_n\ (x \ominus y)) \\ = & \qquad \{\ \text{Definition of } h_i\ \} \\ & (((\triangle_1^n h_i)\ x) \oplus_1 ((\triangle_1^n h_i)\ y), \ldots, ((\triangle_1^n h_i)\ x) \oplus_n ((\triangle_1^n h_i)\ y)) \\ = & \qquad \{\ \text{Definition of } \triangle\ \} \\ & ((\triangle_1^n h_i)\ x)\ (\triangle_1^n \oplus_i)\ ((\triangle_1^n h_i)\ y) \end{array}$$

The third is proved similarly. □

Theorem 4.9 says that if function h_1 is mutually defined with other functions (i.e., $h_2, \ldots, h_n$) which traverse over the same array in the specific form of Eq. (4.10), then tupling $h_1, \ldots, h_n$ will give a homomorphism. It follows that every h_i is an almost homomorphism.

Next, we will give the following theorem showing how to fuse a function with an almost homomorphism to get new another almost homomorphism, which is an extension of the fusion theorem of homomorphism(Theorem 4.5).

Theorem 4.11 (Almost Fusion)**.**
Let h and $(\![\, \Delta_1^n f_i, \Delta_1^n \oplus_i, \Delta_1^n \otimes_i \,]\!)$ be given. If there exist $\odot_i, \ominus_i \, (i = 1, \ldots, n)$ and $H = h_1 \times h_2 \times \cdots \times h_n \, (h_1 = h)$ such that for any i, x and y,

$$h_i \; (x \oplus_i y) = H \; x \odot_i H \; y$$
$$h_i \; (x \otimes_i y) = H \; x \ominus_i H \; y$$

hold, then

$$h \circ (\pi_1 \circ (\![\, \Delta_1^n f_i, \Delta_1^n \oplus_i, \Delta_1^n \otimes_i \,]\!)) = \pi_1 \circ (\![\, \Delta_1^n (h_i \circ f_i), \Delta_1^n \odot_i, \Delta_1^n \ominus_i \,]\!) \; . \tag{4.12}$$

Proof. The theorem is proved by some calculation and Theorem 4.5.
We prove it by the following calculation.

$$\begin{array}{ll} & h \circ (\pi_1 \circ (\![\, \Delta_1^n f_i, \Delta_1^n \oplus_i, \Delta_1^n \otimes_i \,]\!)) \\ = & \quad \{ \text{ Definition of } H \text{ and } \pi_1 \} \\ & \pi_1 \circ H \circ (\![\, \Delta_1^n f_i, \Delta_1^n \oplus_i, \Delta_1^n \otimes_i \,]\!) \\ = & \quad \{ \text{ Theorem 4.5 and the proofs below } \} \\ & \pi_1 \circ (\![\, \Delta_1^n (h_i \circ f_i), \Delta_1^n \odot_i, \Delta_1^n \ominus_i \,]\!) \end{array}$$

To complete the above proof, we need to show

$$\left\{ \begin{array}{lcl} H \circ (\Delta_1^n f_i) & = & \Delta_1^n (h_i \circ f_i) \\ H(x \; (\Delta_1^n \oplus_i) \; y) & = & (H \; x) \; (\Delta_1^n \odot_i) \; (H \; y) \\ H(x \; (\Delta_1^n \otimes_i) \; y) & = & (H \; x) \; (\Delta_1^n \ominus_i) \; (H \; y) \; . \end{array} \right.$$

These equations are proved as follows.

$$\begin{array}{ll} & (H \circ (\Delta_1^n f_i)) \; a \\ = & \quad \{ \text{ Definition of } \Delta \text{ and } H \} \\ & ((h_1 \circ f_1) \; a, \ldots, (h_n \circ f_n) \; a) \\ = & \quad \{ \text{ Definition of } \Delta \} \\ & (\Delta_1^n (h_i \circ f_i)) \; a \end{array}$$

$$\begin{array}{ll} & H \; (x \; (\Delta_1^n \oplus_i) \; y) \\ = & \quad \{ \text{ Definition of } \Delta \text{ and } H \} \\ & (h_1 \; (x \oplus_1 y), \ldots, h_n \; (x \oplus_n y)) \\ = & \quad \{ \text{ Assumption of } h_i \} \\ & ((H \; x) \odot_1 (H \; y), \ldots, (H \; x) \odot_n (H \; y)) \\ = & \quad \{ \text{ Definition of } \Delta \} \\ & (H \; x) \; (\Delta_1^n \odot_i) \; (H \; y) \end{array}$$

The third is similar to the second, and can be proved similarly. □

Theorem 4.11 says that we can fuse a function with an almost homomorphism to get another almost homomorphism by finding $h_2, \ldots, h_n$ together with $\odot_1, \ldots, \odot_n$, $\ominus_1, \ldots, \ominus_n$ that satisfy Eq. (4.12).

4.3.3 A Strategy for Deriving Efficient Parallel Programs

Now, we will give a strategy to derive efficient parallel program using almost-homomorphism and the theorems.

Our strategy for deriving efficient parallel programs on two-dimensional arrays consists of the following four steps, extending the result of lists [HIT97]. The goal of the strategy is to write a given program by an efficient almost-homomorphism that has an efficient implementation in parallel.

Step 1. Define the target program p as a composition of $p_1, \ldots, p_n$ that are already defined, i.e. $p = p_n \circ \cdots \circ p_1$. Each of $p_1, \ldots, p_n$ may be defined as a composition of small functions or a recursive function.

Step 2. Derive an almost homomorphism from the recursive definition of p_1 using the tupling of Theorem 4.9.

Step 3. Fuse p_2 into the derived almost homomorphism to obtain a new almost homomorphism for $p_2 \circ p_1$, and repeat this derivation until p_n is fused. In this step, we can use the almost-fusion of Theorem 4.11.

Step 4. Let $\pi_1 \circ (\![f, \oplus, \otimes]\!)$ be the resulting almost homomorphism for $p_n \circ \cdots \circ p_1$ obtained at Step 3. For the functions inside the homomorphism, namely f, $\oplus$ and $\otimes$, try to repeat Steps 2 and 3 to find efficient parallel implementations for them.

The key idea is to make a seed almost-homomorphism from p_1 by the tupling, and make the almost-homomorphism bigger by fusing the functions $p_2, \ldots, p_n$.

4.3.4 Deriving an Efficient Parallel Program for Maximum Rectangle Sum Problem

Now, we demonstrate the strategy through a derivation of an efficient program for the maximum rectangle sum problem: compute the maximum of sums of all rectangle areas in a two-dimensional data. This problem was originated by Bentley [Ben84a,Ben84b] and improved by Takaoka [Tak02]. The solution can be used in a sort of data mining and pattern matching of two dimensional data. For example, for the following two-dimensional data

$$\begin{pmatrix} 3 & -1 & \mathbf{4} & \mathbf{-1} & -5 \\ 1 & -4 & \mathbf{-1} & \mathbf{5} & -3 \\ -4 & 1 & \mathbf{5} & \mathbf{3} & 1 \end{pmatrix}$$

the result should be 15, which denotes the maximum sum contributed by the sub-rectangular area with bold numbers above. To appreciate difficulty of this problem, we ask the reader to pause for a while to think of how to solve it.

Step 1. Defining a Clear Parallel Program

We can compute the maximum rectangle sum as follows: enumerating all possible rectangles, then computing sums for all rectangles, and finally returning the maximum value as the result. Therefore, a clear, straightforward program *mrs* can be described as follows.

$$
\begin{aligned}
mrs \;&=\; max \circ \mathsf{map}\ sum \circ rects' \\
&\quad \textbf{where} \\
&\quad\quad max \;=\; \mathsf{reduce}\ (\uparrow, \uparrow) \\
&\quad\quad sum \;=\; \mathsf{reduce}\ (+, +) \\
&\quad\quad rects' \;=\; \mathit{flatten} \circ \mathsf{map}\ \mathsf{TLs} \circ \mathsf{BRs} \\
&\quad\quad \mathsf{TLs} \;=\; \mathsf{scan}\ (\ominus, \phi) \circ \mathsf{map}\ |\cdot| \\
&\quad\quad \mathsf{BRs} \;=\; \mathsf{scanr}\ (\ominus, \phi) \circ \mathsf{map}\ |\cdot|
\end{aligned}
$$

Here, $rects'$ generates all possible rectangles of the given array, using TLs and BRs to generate top-left and bottom-right rectangles, respectively. For example, applying TLs, BRs, and $rects'$ to $\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$, we get the following results.

$$
\mathsf{TLs}\ \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} = \begin{pmatrix} \begin{pmatrix} (1) \\ \begin{pmatrix} 1 \\ 4 \end{pmatrix} \end{pmatrix} & \begin{pmatrix} (1\ \ 2) \\ \begin{pmatrix} 1 & 2 \\ 4 & 5 \end{pmatrix} \end{pmatrix} & \begin{pmatrix} (1\ \ 2\ \ 3) \\ \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \end{pmatrix} \end{pmatrix}
$$

$$
\mathsf{BRs}\ \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} = \begin{pmatrix} \begin{pmatrix} \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \\ (4\ \ 5\ \ 6) \end{pmatrix} & \begin{pmatrix} \begin{pmatrix} 2 & 3 \\ 5 & 6 \end{pmatrix} \\ (5\ \ 6) \end{pmatrix} & \begin{pmatrix} \begin{pmatrix} 3 \\ 6 \end{pmatrix} \\ (6) \end{pmatrix} \end{pmatrix}
$$

$$
rects'\ \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} = \begin{pmatrix} \begin{pmatrix} (1) \\ \begin{pmatrix} 1 \\ 4 \end{pmatrix} \\ (4) \end{pmatrix} & \begin{pmatrix} (1\ \ 2) \\ \begin{pmatrix} 1 & 2 \\ 4 & 5 \end{pmatrix} \\ (4\ \ 5) \end{pmatrix} & \begin{pmatrix} (1\ \ 2\ \ 3) \\ \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \\ (4\ \ 5\ \ 6) \end{pmatrix} & \begin{pmatrix} (2) \\ \begin{pmatrix} 2 \\ 5 \end{pmatrix} \\ (5) \end{pmatrix} & \begin{pmatrix} (2\ \ 3) \\ \begin{pmatrix} 2 & 3 \\ 5 & 6 \end{pmatrix} \\ (5\ \ 6) \end{pmatrix} & \begin{pmatrix} (3) \\ \begin{pmatrix} 3 \\ 6 \end{pmatrix} \\ (6) \end{pmatrix} \end{pmatrix}
$$

Apparently, *mrs* is a clear parallel program, because it is described with skeletons and easy to understand. However, it is inefficient in the sense that it needs to execute $O(n^6)$ addition operations for the input of $n \times n$ array. Therefore, we want to develop a more efficient parallel program according to the strategy.

Step 2. Driving Almost Homomorphism

The second step is to make a seed almost-homomorphism for the following steps. To this end, we will first do "dilation" of the function $rects'$. Then, we will derive a seed almost-homomorphism.

Although the current *mrs* is clear to understand, the function $rects'$ to generate all possible rectangles cuts information of rectangles too much to manipulate the program further. Therefore, we "dilate" the function $rects'$ to hold more information

in the result. The dilation is given as follows (please see Appendix D for the proof of the dilation).

$$
\begin{array}{lll}
\multicolumn{3}{l}{rects' = pack \circ rects} \\
\quad \textbf{where}\ pack & = & \mathit{flatten} \circ unwind_{\ominus} \circ \mathsf{map}\ \ unwind_{\phi} \\
\quad\quad unwind_{\oplus}\ |a| & = & |a| \\
\quad\quad unwind_{\oplus}\ (|a| \phi x) & = & |a| \oplus \mathsf{reduce}\ \ (_, \oplus)\ (\mathsf{map}\ \ |\cdot|\ x) \\
\quad\quad unwind_{\oplus}\ ((|a| \phi x) \ominus (NIL \phi y)) & = & unwind_{\oplus}\ (|a| \phi x) \oplus unwind_{\oplus}\ y
\end{array}
$$

Here, *rects* is a dilated function to generate possible rectangles of the given array, and function *pack* removes information of dilated structures by the function *unwind*. An example use of *rects* is shown below. The return value of *rects* is an array of arrays of arrays; the (k, l)-element of the (i, j)-element of the resulting array is a sub-rectangle consisting of the rows from the ith to the jth and the columns from the kth to the lth of the original array. Note that the dilated structure contains special value *NIL* to denote blank portions of the above-mentioned array. *NIL* is seen as a suitable-size array (of arrays) of $-\infty$ (the identity of $\uparrow$).

$$
rects\ \begin{pmatrix} 1 & 2 & 3 \\ 5 & 6 & 7 \end{pmatrix} =
\begin{pmatrix}
\begin{pmatrix} (1) & (1\ \ 2) & (1\ \ 2\ \ 3) \\ & (2) & (2\ \ 3) \\ & & (3) \end{pmatrix} &
\begin{pmatrix} \begin{pmatrix} 1 \\ 5 \end{pmatrix} & \begin{pmatrix} 1 & 2 \\ 5 & 6 \end{pmatrix} & \begin{pmatrix} 1 & 2 & 3 \\ 5 & 6 & 7 \end{pmatrix} \\ & \begin{pmatrix} 2 \\ 6 \end{pmatrix} & \begin{pmatrix} 2 & 3 \\ 6 & 7 \end{pmatrix} \\ & & \begin{pmatrix} 3 \\ 7 \end{pmatrix} \end{pmatrix} \\
 & \begin{pmatrix} (5) & (5\ \ 6) & (5\ \ 6\ \ 7) \\ & (6) & (6\ \ 7) \\ & & (7) \end{pmatrix}
\end{pmatrix}
$$

Here is a precise definition of the function *rects*. Function *rects* is mutual recursively defined with functions *bottoms*, *tops*, *rights*, and *lefts* (Figure 4.1) as follows.

$$
\begin{array}{lll}
rects\, |a| & = & |||a||| \\
rects\, (x \ominus y) & = & (rects\, x \phi\ gemm(_, \mathsf{zipwith}(\ominus))\ (bottoms\, x)\, (tops\, y)) \\
 & & \qquad\qquad\qquad\qquad\qquad \ominus(NIL \phi\ rects\, y) \\
rects\, (x \phi y) & = & \mathsf{zipwith}_4\ f_s\, (rects\, x)\, (rects\, y)\, (rights\, x)\, (lefts\, y) \\
 & & \textbf{where}\ f_s\, s_1\, s_2\, r_1\, l_2 = (s_1 \phi\ gemm(_, \phi)\, r_1\, l_2) \ominus (NIL \phi\ s_2)
\end{array}
$$

where '_' indicates "don't care", and generalized matrix multiplication *gemm* is defined as follows[1]. Here, we assume that the division of argument arrays are con-

[1]Actually, we have defined the same function $gemm(\oplus, \otimes)$ in the previous section in the different from. But, both definitions returns the same result.

sistent.

$$
\begin{array}{lll}
\multicolumn{3}{l}{gemm(\oplus, \otimes) = g} \\
\quad \textbf{where}\ g\,(X_1 \phi X_2)\,(Y_1 \ominus Y_2) & = & \mathsf{zipwith}(\oplus)\,(g\,X_1\,Y_1)\,(g\,X_2\,Y_2) \\
\qquad\qquad\ \ g\,(X_1 \ominus X_2)\,Y & = & (g\,X_1\,Y) \ominus (g\,X_2\,Y) \\
\qquad\qquad\ \ g\,X\,(Y_1 \phi Y_2) & = & (g\,X\,Y_1) \phi (g\,X\,Y_2) \\
\qquad\qquad\ \ g\,|a|\,|b| & = & |a \otimes b|
\end{array}
$$

Figure 4.2 shows recursive computation of *rects* for the $\ominus$ case. Since indices i and j of the resulting array mean the top row and the bottom row of the generated rectangles within the original array, the upper-left block ($i \leq m_x$ and $j \leq m_x$) of the resulting array contains rectangles included in the upper block x of the original array. Thus, the upper-left block is equal to *rects* x. Similarly, the lower-right block is *rects* y (Figure 4.2-(a)). Each rectangle in the the upper-right block ($i \leq m_x$ and $j > m_x$) consists of a rectangle touching the bottom of x and a rectangle touching the top of y as shown in Figure 4.2-(b). Since we have to consider all combinations of i and j, we compute this block by a general matrix multiplication on *bottoms* x and *tops* y as shown in Figure 4.2-(c). The computation of the $\ominus$ case is similar.

Functions *bottoms*, *tops*, *rights*, and *lefts* are similarly defined as mutual recursive functions shown in Figure 4.1, and their examples are shown in Figure 4.3. Each of these functions is the partial result of *rects* in the sense that it returns the rectangles that are restricted to include some edges of the input as shown in Figure 4.4. For example, the *tops* returns the rectangles that include the top edges (the top-most row) of the input array. Similarly, *bottoms*, *rights* and *lefts* return the rectangles that include the bottom edges, the right edge and the left edge respectively. The other functions returns the rectangles that include two edges of the input. Usually, users do not need to understand well these functions because those well-used functions are provided by experts.

The function *pack* gives us the relation between the dilated structure and the original structure. It uses the function *unwind* to unfold the dilated triangular structures. Its image is shown below.

$$
unwind_{\phi} \begin{pmatrix} a & b & c \\ & d & e \\ & & f \end{pmatrix} = (a \quad b \quad c \quad d \quad e \quad f)
$$

The function $unwind_{\phi}$ similarly unfolds the dilated triangular structures into column vectors.

It is easily seen the following equation, because the function *pack* simply changes the dilated structure and does not change the values, and a reduction of the special value *NIL* results in the identity of $\uparrow$.

$$
max \circ \mathsf{map}\ \ sum \circ pack = max \circ \mathsf{map}\ \ max \circ \mathsf{map}\ \ (\mathsf{map}\ \ sum)
$$

The dilation is important for derivation of efficient parallel programs on two-dimensional arrays, so that we can successfully derive operators that satisfy the

$$
\begin{array}{lll}
\mathit{tops}\ |a| &=& |||a||| \\
\mathit{tops}\ (x \ominus y) &=& \mathit{tops}\ x \obar \mathsf{map}\ (\mathsf{zipwith}(\ominus)\ (\mathit{cols}'\ x))\ (\mathit{tops}\ y) \\
\mathit{tops}\ (x \obar y) &=& \mathsf{zipwith}_4\ f_t\ (\mathit{tops}\ x)\ (\mathit{tops}\ y)\ (\mathit{toprights}\ x)\ (\mathit{toplefts}\ y) \\
 & & \quad \textbf{where}\ f_t\ t_1\ t_2\ tr_1\ tl_2 = (t_1 \obar \mathit{gemm}\ (_, \obar)\ tr_1\ tl_2) \ominus (\mathit{NIL} \obar t_2)
\end{array}
$$

$$
\begin{array}{lll}
\mathit{bottoms}\ |a| &=& |||a||| \\
\mathit{bottoms}\ (x \ominus y) &=& \mathsf{map}\ (\lambda z.\, \mathsf{zipwith}(\ominus)\ z\ (\mathit{cols}'\ y))\ (\mathit{bottoms}\ x) \ominus \mathit{bottoms}\ y \\
\mathit{bottoms}\ (x \obar y) &=& \mathsf{zipwith}_4\ f_b\ (\mathit{bottoms}\ x)\ (\mathit{bottoms}\ y)\ (\mathit{bottomrights}\ x)\ (\mathit{bottomlefts}\ y) \\
 & & \quad \textbf{where}\ f_b\ b_1\ b_2\ br_1\ bl_2 = (b_1 \obar \mathit{gemm}\ (_, \obar)\ br_1\ bl_2) \ominus (\mathit{NIL} \obar b_2)
\end{array}
$$

$$
\begin{array}{lll}
\mathit{rights}\ |a| &=& |||a||| \\
\mathit{rights}\ (x \ominus y) &=& (\mathit{rights}\ x \obar \mathit{gemm}\ (_, \mathsf{zipwith}(\ominus))\ (\mathit{bottomrights}\ x)\ (\mathit{toprights}\ y)) \\
 & & \quad \ominus (\mathit{NIL} \obar \mathit{rights}\ y) \\
\mathit{rights}\ (x \obar y) &=& \mathsf{zipwith}_3\ f_r\ (\mathit{rights}\ x)\ (\mathit{rights}\ y)\ (\mathit{rows}'\ y) \\
 & & \quad \textbf{where}\ f_r\ r_1\ r_2\ ro_2 = \mathsf{map}\ (\obar ro_2)\ r_1 \ominus r_2
\end{array}
$$

$$
\begin{array}{lll}
\mathit{lefts}\ |a| &=& |||a||| \\
\mathit{lefts}\ (x \ominus y) &=& (\mathit{lefts}\ x \obar \mathit{gemm}\ (_, \mathsf{zipwith}(\ominus))\ (\mathit{bottomlefts}\ x)\ (\mathit{toplefts}\ y)) \ominus (\mathit{NIL} \obar \mathit{lefts}\ y) \\
\mathit{lefts}\ (x \obar y) &=& \mathsf{zipwith}_3\ f_l\ (\mathit{lefts}\ x)\ (\mathit{lefts}\ y)\ (\mathit{rows}'\ x) \\
 & & \quad \textbf{where}\ f_l\ l_1\ l_2\ ro_1 = l_1 \obar \mathsf{map}\ (ro_1 \obar)\ l_2
\end{array}
$$

$$
\begin{array}{lll}
\mathit{toprights}\ |a| &=& |||a||| \\
\mathit{toprights}\ (x \ominus y) &=& \mathit{toprights}\ x \obar \mathsf{map}\ (\mathsf{zipwith}(\ominus)\ (\mathit{right}'\ (\mathit{toprights}\ x)))\ (\mathit{toprights}\ y) \\
\mathit{toprights}\ (x \obar y) &=& \mathsf{zipwith}\ f_{tr}\ (\mathit{toprights}\ x)\ (\mathit{toprights}\ y) \\
 & & \quad \textbf{where}\ f_{tr}\ tr_1\ tr_2 = \mathsf{map}\ (\obar \mathit{top}'\ tr_2)\ tr_1 \ominus tr_2
\end{array}
$$

$$
\begin{array}{lll}
\mathit{bottomrights}\ |a| &=& |||a||| \\
\mathit{bottomrights}\ (x \ominus y) &=& \mathsf{map}\ (\lambda z.\, \mathsf{zipwith}(\ominus)\ z\ (\mathit{top}'\ (\mathit{bottomrights}\ y)))\ (\mathit{bottomrights}\ x) \\
 & & \quad \ominus \mathit{bottomrights}\ y \\
\mathit{bottomrights}\ (x \obar y) &=& \mathsf{zipwith}\ f_{br}\ (\mathit{bottomrights}\ x)\ (\mathit{bottomrights}\ y) \\
 & & \quad \textbf{where}\ f_{br}\ br_1\ br_2 = \mathsf{map}\ (\obar \mathit{top}'\ br_2)\ br_1 \ominus br_2
\end{array}
$$

$$
\begin{array}{lll}
\mathit{toplefts}\ |a| &=& |||a||| \\
\mathit{toplefts}\ (x \ominus y) &=& \mathit{toplefts}\ x \obar \mathsf{map}\ (\mathsf{zipwith}(\ominus)\ (\mathit{right}'\ (\mathit{toplefts}\ x)))\ (\mathit{toplefts}\ y) \\
\mathit{toplefts}\ (x \obar y) &=& \mathsf{zipwith}\ f_{tl}\ (\mathit{toplefts}\ x)\ (\mathit{toplefts}\ y) \\
 & & \quad \textbf{where}\ f_{tl}\ tl_1\ tl_2 = tl_1 \obar \mathsf{map}\ (\mathit{right}'\ tl_1 \obar)\ tl_2
\end{array}
$$

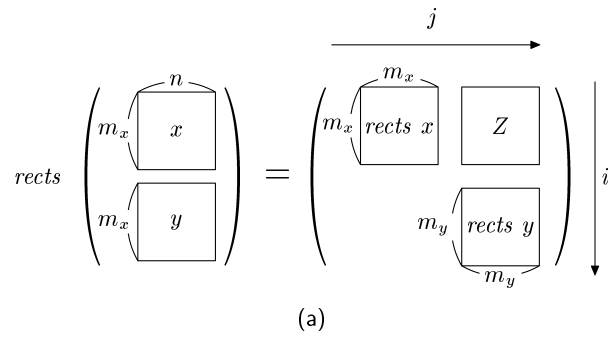
$$
\begin{array}{lll}
\mathit{bottomlefts}\ |a| &=& |||a||| \\
\mathit{bottomlefts}\ (x \ominus y) &=& \mathsf{map}\ (\lambda z.\, \mathsf{zipwith}(\ominus)\ z\ (\mathit{top}'\ (\mathit{bottomlefts}\ y)))\ (\mathit{bottomlefts}\ x) \\
 & & \quad \ominus \mathit{bottomlefts}\ y \\
\mathit{bottomlefts}\ (x \obar y) &=& \mathsf{zipwith}\ f_{bl}\ (\mathit{bottomlefts}\ x)\ (\mathit{bottomlefts}\ y) \\
 & & \quad \textbf{where}\ f_{bl}\ bl_1\ bl_2 = bl_1 \obar \mathsf{map}\ (\mathit{right}'\ bl_1 \obar)\ bl_2
\end{array}
$$

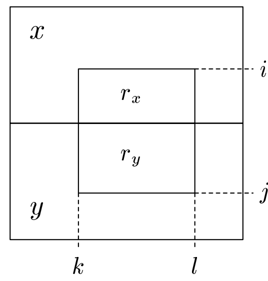
$$
\begin{array}{lll}
\mathit{cols}'\ |a| &=& ||a|| \\
\mathit{cols}'\ (x \ominus y) &=& \mathsf{zipwith}(\ominus)\ (\mathit{cols}'\ x)\ (\mathit{cols}'\ y) \\
\mathit{cols}'\ (x \obar y) &=& (\mathit{cols}'\ x \obar \mathit{gemm}\ (_, \obar)\ (\mathit{right}\ (\mathit{cols}'\ x))\ (\mathit{top}\ (\mathit{cols}'\ y))) \ominus (\mathit{NIL} \obar \mathit{cols}'\ y)
\end{array}
$$

$$
\begin{array}{lll}
\mathit{rows}'\ |a| &=& ||a|| \\
\mathit{rows}'\ (x \ominus y) &=& (\mathit{rows}'\ x \obar \mathit{gemm}\ (_, \ominus)\ (\mathit{right}\ (\mathit{rows}'\ x))\ (\mathit{top}\ (\mathit{rows}'\ y))) \ominus (\mathit{NIL} \obar \mathit{rows}'\ y) \\
\mathit{rows}'\ (x \obar y) &=& \mathsf{zipwith}(\obar)\ (\mathit{rows}'\ x)\ (\mathit{rows}'\ y)
\end{array}
$$

$$
\begin{array}{lllllll}
\mathit{top} &=& \mathsf{reduce}(\ll, \obar) \circ \mathsf{map}\ |\cdot| & \qquad & \mathit{top}' &=& \mathsf{the} \circ \mathit{top} \\
\mathit{bottom} &=& \mathsf{reduce}(\gg, \obar) \circ \mathsf{map}\ |\cdot| & & \mathit{bottom}' &=& \mathsf{the} \circ \mathit{bottom} \\
\mathit{right} &=& \mathsf{reduce}(\ominus, \gg) \circ \mathsf{map}\ |\cdot| & & \mathit{right}' &=& \mathsf{the} \circ \mathit{right} \\
\mathit{left} &=& \mathsf{reduce}(\ominus, \ll) \circ \mathsf{map}\ |\cdot| & & \mathit{left}' &=& \mathsf{the} \circ \mathit{left}
\end{array}
$$

Figure 4.1. Auxiliary functions for *rects*.





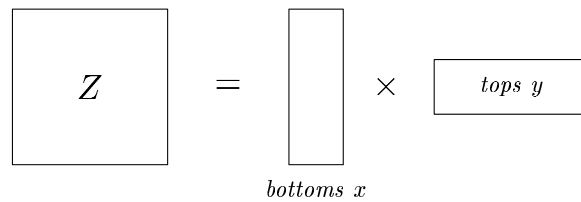


Figure 4.2. Recursive computation of *rects*.

$$rects\ \begin{pmatrix} a & b \\ c & d \end{pmatrix} = \begin{pmatrix} \begin{pmatrix} (a) & (a\ \ b) \\ & (b) \end{pmatrix} & \begin{pmatrix} \begin{pmatrix} a \\ c \end{pmatrix} & \begin{pmatrix} a & b \\ c & d \end{pmatrix} \\ & \begin{pmatrix} b \\ d \end{pmatrix} \end{pmatrix} \\ & \begin{pmatrix} (c) & (c\ \ d) \\ & (d) \end{pmatrix} \end{pmatrix}$$

$$tops\ \begin{pmatrix} a & b \\ c & d \end{pmatrix} = \begin{pmatrix} \begin{pmatrix} (a) & (a\ \ b) \\ & (b) \end{pmatrix} & \begin{pmatrix} \begin{pmatrix} a \\ c \end{pmatrix} & \begin{pmatrix} a & b \\ c & d \end{pmatrix} \\ & \begin{pmatrix} b \\ d \end{pmatrix} \end{pmatrix} \end{pmatrix}$$

$$bottoms\ \begin{pmatrix} a & b \\ c & d \end{pmatrix} = \begin{pmatrix} \begin{pmatrix} \begin{pmatrix} a \\ c \end{pmatrix} & \begin{pmatrix} a & b \\ c & d \end{pmatrix} \\ & \begin{pmatrix} b \\ d \end{pmatrix} \end{pmatrix} \\ \begin{pmatrix} (c) & (c\ \ d) \\ & (d) \end{pmatrix} \end{pmatrix}$$

$$rights\ \begin{pmatrix} a & b \\ c & d \end{pmatrix} = \begin{pmatrix} \begin{pmatrix} (a\ \ b) \\ (b) \end{pmatrix} & \begin{pmatrix} \begin{pmatrix} a & b \\ c & d \end{pmatrix} \\ \begin{pmatrix} b \\ d \end{pmatrix} \end{pmatrix} \\ & \begin{pmatrix} (c\ \ d) \\ (d) \end{pmatrix} \end{pmatrix}$$

$$lefts\ \begin{pmatrix} a & b \\ c & d \end{pmatrix} = \begin{pmatrix} \big((a)\ \ (a\ \ b)\big) & \begin{pmatrix} \begin{pmatrix} a \\ c \end{pmatrix} & \begin{pmatrix} a & b \\ c & d \end{pmatrix} \end{pmatrix} \\ & \big((c)\ \ (c\ \ d)\big) \end{pmatrix}$$

$$toprights\ \begin{pmatrix} a & b \\ c & d \end{pmatrix} = \begin{pmatrix} \begin{pmatrix} (a\ \ b) \\ (b) \end{pmatrix} & \begin{pmatrix} \begin{pmatrix} a & b \\ c & d \end{pmatrix} \\ \begin{pmatrix} b \\ d \end{pmatrix} \end{pmatrix} \end{pmatrix}$$

$$bottomrights\ \begin{pmatrix} a & b \\ c & d \end{pmatrix} = \begin{pmatrix} \begin{pmatrix} \begin{pmatrix} a & b \\ c & d \end{pmatrix} \\ \begin{pmatrix} b \\ d \end{pmatrix} \end{pmatrix} \\ \begin{pmatrix} (c\ \ d) \\ (d) \end{pmatrix} \end{pmatrix}$$

$$toplefts\ \begin{pmatrix} a & b \\ c & d \end{pmatrix} = \begin{pmatrix} \big((a)\ \ (a\ \ b)\big) & \begin{pmatrix} \begin{pmatrix} a \\ c \end{pmatrix} & \begin{pmatrix} a & b \\ c & d \end{pmatrix} \end{pmatrix} \end{pmatrix}$$

$$bottomlefts\ \begin{pmatrix} a & b \\ c & d \end{pmatrix} = \begin{pmatrix} \begin{pmatrix} \begin{pmatrix} a \\ c \end{pmatrix} & \begin{pmatrix} a & b \\ c & d \end{pmatrix} \end{pmatrix} \\ \big((c)\ \ (c\ \ d)\big) \end{pmatrix}$$

$$cols\ \begin{pmatrix} a & b \\ c & d \end{pmatrix} = \begin{pmatrix} \begin{pmatrix} a \\ c \end{pmatrix} & \begin{pmatrix} a & b \\ c & d \end{pmatrix} \\ & \begin{pmatrix} b \\ d \end{pmatrix} \end{pmatrix}$$

$$rows\ \begin{pmatrix} a & b \\ c & d \end{pmatrix} = \begin{pmatrix} (a\ \ b) & \begin{pmatrix} a & b \\ c & d \end{pmatrix} \\ & (c\ \ d) \end{pmatrix}$$

Figure 4.3. Examples of auxiliary functions for *rects*.

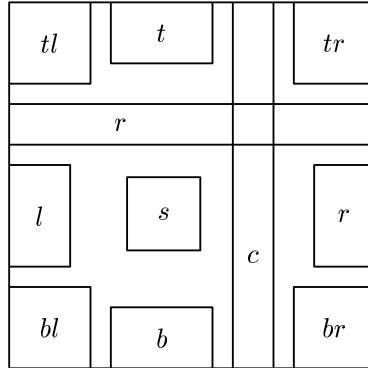


Figure 4.4. Corresponding parts of elements in the tuple.

abide property. Usually, operands of operators with the abide property should have capability for computation in two directions. The dilated information is very useful to create the capability.

Now, our program is described as follows, and our goal of this step is to make the first function *rects* an almost-homomorphism. The tupling theorem (Theorem 4.9) gives us a systematic way to execute this step of the strategy.

$$mrs \;=\; max \circ \mathsf{map}\ \ max \circ \mathsf{map}\ \ (\mathsf{map}\ \ sum) \circ rects$$

We apply the tupling theorem to derive an almost homomorphism for *rects*. The definition of *rects* and the extra functions are in the form of Eq. (4.10). Thus, we can obtain the almost homomorphism shown in Figure 4.5 by tupling these functions. The operators are straightforward rewriting of the definition of *rects*.

Step 3. Fusing with Almost Homomorphisms

We aim to derive an efficient almost homomorphism for mrs. The almost-fusion theorem (Theorem 4.11) gives us a systematic way to execute this step the strategy.

Now, we will apply this theorem to mrs repeatedly.

The second function p_2 of our example is $\mathsf{map}\ (\mathsf{map}\ \ sum)$, so $h_1 = \mathsf{map}\ (\mathsf{map}\ \ sum)$.

$$rects = \pi_1 \circ (\triangle_1^{11} h_i) = \pi_1 \circ (\!| \triangle_1^{11} f_i, \triangle_1^{11} \oplus_i, \triangle_1^{11} \otimes_i |\!)$$

where

$$\triangle_1^{11} f_i\ |a| \quad = \quad (|||a|||, |||a|||, |||a|||, |||a|||, |||a|||, |||a|||, |||a|||, |||a|||, |||a|||, ||a||, ||a||)$$

$$(s_1, t_1, b_1, r_1, l_1, tr_1, br_1, tl_1, bl_1, c_1, ro_1)\ (\triangle_1^{11} \oplus_i)\ (s_2, t_2, b_2, r_2, l_2, tr_2, br_2, tl_2, bl_2, c_2, ro_2)$$
$$= (s_0, t_0, b_0, r_0, l_0, tr_0, br_0, tl_0, bl_0, c_0, ro_0)$$

where

$$\begin{aligned}
s_0 &= (s_1 \phi\ gemm\ (_, \mathsf{zipwith}(\ominus))\ b_1\ t_2) \ominus (NIL \phi\ s_2) \\
t_0 &= t_1 \phi\ \mathsf{map}\ (\mathsf{zipwith}(\ominus)\ c_1)\ t_2 \\
b_0 &= \mathsf{map}\ (\lambda z.\, \mathsf{zipwith}(\ominus)\ z\ c_2)\ b_1 \ominus b_2 \\
r_0 &= (r_1 \phi\ gemm\ (_, \mathsf{zipwith}(\ominus))\ br_1\ tr_2) \ominus (NIL \phi\ r_2) \\
l_0 &= (l_1 \phi\ gemm\ (_, \mathsf{zipwith}(\ominus))\ bl_1\ tl_2) \ominus (NIL \phi\ l_2) \\
tr_0 &= tr_1 \phi\ \mathsf{map}\ (\mathsf{zipwith}(\ominus)\ (right'\ tr_1))\ tr_2 \\
br_0 &= \mathsf{map}\ (\lambda z.\, \mathsf{zipwith}(\ominus)\ z\ (top'\ br_2))\ br_1 \ominus br_2 \\
tl_0 &= tl_1 \phi\ \mathsf{map}\ (\mathsf{zipwith}(\ominus)\ (right'\ tl_1))\ tl_2 \\
bl_0 &= \mathsf{map}\ (\lambda z.\, \mathsf{zipwith}(\ominus)\ z\ (top'\ bl_2))\ bl_1 \ominus bl_2 \\
c_0 &= \mathsf{zipwith}(\ominus)\ c_1\ c_2 \\
ro_0 &= (ro_1 \phi\ gemm\ (_, \ominus)\ (right\ ro_1)\ (top\ ro_2)) \ominus (NIL \phi\ ro_2)
\end{aligned}$$

$$(s_1, t_1, b_1, r_1, l_1, tr_1, br_1, tl_1, bl_1, c_1, ro_1)\ (\triangle_1^{11} \otimes_i)\ (s_2, t_2, b_2, r_2, l_2, tr_2, br_2, tl_2, bl_2, c_2, ro_2)$$
$$= (s_0, t_0, b_0, r_0, l_0, tr_0, br_0, tl_0, bl_0, c_0, ro_0)$$

where

$$\begin{aligned}
s_0 &= \mathsf{zipwith}_4\ f_s\ s_1\ s_2\ r_1\ l_2 \\
&\quad \textbf{where}\ f_s\ s_1\ s_2\ r_1\ l_2 = (s_1 \phi\ gemm\ (_, \phi)\ r_1\ l_2) \ominus (NIL \phi\ s_2) \\
t_0 &= \mathsf{zipwith}_4\ f_t\ t_1\ t_2\ tr_1\ tl_2 \\
&\quad \textbf{where}\ f_t\ t_1\ t_2\ tr_1\ tl_2 = (t_1 \phi\ gemm\ (_, \phi)\ tr_1\ tl_2) \ominus (NIL \phi\ t_2) \\
b_0 &= \mathsf{zipwith}_4\ f_b\ b_1\ b_2\ br_1\ bl_2 \\
&\quad \textbf{where}\ f_b\ b_1\ b_2\ br_1\ bl_2 = (b_1 \phi\ gemm\ (_, \phi)\ br_1\ bl_2) \ominus (NIL \phi\ b_2) \\
r_0 &= \mathsf{zipwith}_3\ f_r\ r_1\ r_2\ ro_2 \\
&\quad \textbf{where}\ f_r\ r_1\ r_2\ ro_2 = \mathsf{map}\ (\phi ro_2)\ r_1 \ominus r_2 \\
l_0 &= \mathsf{zipwith}_3\ f_l\ l_1\ l_2\ ro_1 \\
&\quad \textbf{where}\ f_l\ l_1\ l_2\ ro_1 = l_1 \phi\ \mathsf{map}\ (ro_1 \phi)\ l_2 \\
tr_0 &= \mathsf{zipwith}\ f_{tr}\ tr_1\ tr_2 \\
&\quad \textbf{where}\ f_{tr}\ tr_1\ tr_2 = \mathsf{map}\ (\phi\, top'\ tr_2)\ tr_1 \ominus tr_2 \\
br_0 &= \mathsf{zipwith}\ f_{br}\ br_1\ br_2 \\
&\quad \textbf{where}\ f_{br}\ br_1\ br_2 = \mathsf{map}\ (\phi\, top'\ br_2)\ br_1 \ominus br_2 \\
tl_0 &= \mathsf{zipwith}\ f_{tl}\ tl_1\ tl_2 \\
&\quad \textbf{where}\ f_{tl}\ tl_1\ tl_2 = tl_1 \phi\ \mathsf{map}\ (right'\ tl_1 \phi)\ tl_2 \\
bl_0 &= \mathsf{zipwith}\ f_{bl}\ bl_1\ bl_2 \\
&\quad \textbf{where}\ f_{bl}\ bl_1\ bl_2 = bl_1 \phi\ \mathsf{map}\ (right'\ bl_1 \phi)\ bl_2 \\
c_0 &= (c_1 \phi\ gemm\ (_, \phi)\ (right\ c_1)\ (top\ c_2)) \ominus (NIL \phi\ c_2) \\
ro_0 &= \mathsf{zipwith}(\phi)\ ro_1\ ro_2
\end{aligned}$$

Figure 4.5. Almost-homomorphism definition of *rects*.

Then, we calculate $h_1\ (x \oplus_1 y)$ to find other functions and operators.

$$
\begin{array}{ll}
 & h_1\ (x \oplus_1 y) \\
= & \qquad \{ \text{ Expand } x,\ y \text{ and } h_1 \ \} \\
 & \mathsf{map}\,(\mathsf{map}\ sum)\,((s_1 \phi\ gemm(_, \mathsf{zipwith}(\ominus))\, b_1\, t_2) \ominus (NIL \phi\ s_2)) \\
= & \qquad \{ \text{ Definition of } \mathsf{map} \ \} \\
 & (\mathsf{map}\,(\mathsf{map}\ sum) s_1 \phi\ \mathsf{map}\,(\mathsf{map}\ sum)\,(gemm(_, \mathsf{zipwith}(\ominus))\, b_1\, t_2)) \\
 & \qquad\qquad\qquad\qquad \ominus(NIL \phi\ \mathsf{map}\,(\mathsf{map}\ sum) s_2) \\
= & \qquad \{ \text{ Promotion of } \mathsf{map}, \text{ folding } \} \\
 & (h_1\, s_1 \phi\ gemm(_, \mathsf{zipwith}(+))(\mathsf{map}\,(\mathsf{map}\ sum)\, b_1)\,(\mathsf{map}\,(\mathsf{map}\ sum)\, t_2)) \\
 & \qquad\qquad\qquad\qquad \ominus(NIL \phi\ h_1\, s_2)
\end{array}
$$

In the last formula, functions applied to t_1 and b_1 should be h_2 and h_3, respectively, which suggests us to define h_2, h_3 and $\odot_1$ as follows.

$$
\begin{array}{l}
h_1 = h_2 = h_3 = \mathsf{map}\,(\mathsf{map}\ sum) \\
(s_1, t_1, b_1, r_1, l_1, tr_1, br_1, tl_1, bl_1, c_1, ro_1) \\
\quad \odot_1\ \ (s_2, t_2, b_2, r_2, l_2, tr_2, br_2, tl_2, bl_2, c_2, ro_2) \\
\qquad = (s_1 \phi\ gemm\ (_, \mathsf{zipwith}(+))\ b_1\ t_2) \ominus (NIL \phi\ s_2)
\end{array}
$$

Similarly, we can derive $\ominus_1$ by calculating $h_1\ (x \otimes_1 y)$ as follows.

$$
\begin{array}{l}
(s_1, t_1, b_1, r_1, l_1, tr_1, br_1, tl_1, bl_1, c_1, ro_1) \\
\quad \ominus_1\ \ (s_2, t_2, b_2, r_2, l_2, tr_2, br_2, tl_2, bl_2, c_2, ro_2) \\
\qquad = \mathsf{zipwith}_4\ \ f_s\ s_1\ s_2\ r_1\ l_2 \\
\qquad\quad \textbf{where } f_s\, s_1\, s_2\, r_1\, l_2 = (s_1 \phi\ gemm(_, +)\, r_1\, l_2) \ominus (NIL \phi\ s_2)
\end{array}
$$

Deriving the other functions and operators by doing similar calculation about $\oplus_i$ and $\otimes_i$, we finally get the program shown in Figure 4.6. In this derivation, the function H appeared in Theorem 4.11 is as follows:

$$
\begin{array}{lll}
H & = & h \times h \times h \times h \times h \times h \times h \times h \times h \times (\mathsf{map}\ \ sum) \times (\mathsf{map}\ \ sum) \\
 & & \textbf{where } \ h = \mathsf{map}\ (\mathsf{map}\ \ sum)\ .
\end{array}
$$

Some calculation rules used in this derivation are listed in Appendix A.

We will repeat application of the theorem to our example. Applying the theorem again with the third function $p_3 = \mathsf{map}\ max$, we obtain another almost-homomorphism shown in Figure 4.7 with $H = (\mathsf{map}\ \ max) \times id \times id \times id \times id \times id \times id \times id \times id \times id \times id$.

Finally, applying such fusion with max will yield the result shown in Figure 4.8. The function H for the final fusion is as follows:

$$
\begin{array}{l}
H = max \times (\mathsf{reduce}\ \ (_, \mathsf{zipwith}(\uparrow))) \times (\mathsf{reduce}\ \ (\mathsf{zipwith}(\uparrow), _)) \times (\mathsf{map}\ (\mathsf{reduce}\ \ (\uparrow, _))) \\
\qquad \times (\mathsf{map}\ (\mathsf{reduce}\ \ (_, \uparrow))) \times (\mathsf{reduce}\ \ (_, \phi)) \times (\mathsf{reduce}\ \ (\phi, _)) \\
\qquad \times (\mathsf{reduce}\ \ (_, \ominus)) \times (\mathsf{reduce}\ \ (\ominus, _)) \times id \times id
\end{array}
$$

$$\mathsf{map}\,(\mathsf{map}\ sum) \circ rects = \pi_1 \circ (\![\bigtriangleup_1^{11} f_i', \bigtriangleup_1^{11} \odot_i, \bigtriangleup_1^{11} \ominus_i]\!)$$

where

$$\bigtriangleup_1^{11} f_i'\ |a| \ =\ (||a||, ||a||, ||a||, ||a||, ||a||, ||a||, ||a||, ||a||, ||a||, |a|, |a|)$$

$$(s_1, t_1, b_1, r_1, l_1, tr_1, br_1, tl_1, bl_1, c_1, ro_1)\ (\bigtriangleup_1^{11} \odot_i)\ (s_2, t_2, b_2, r_2, l_2, tr_2, br_2, tl_2, bl_2, c_2, ro_2)$$
$$= (s_0, t_0, b_0, r_0, l_0, tr_0, br_0, tl_0, bl_0, c_0, ro_0)$$

where

$$\begin{array}{lcl}
s_0 &=& (s_1 \phi\ gemm\ (_, \mathsf{zipwith}(+))\ b_1\ t_2) \ominus (NIL \phi\ s_2) \\
t_0 &=& t_1 \phi\ \mathsf{map}\ (\mathsf{zipwith}(+)\ c_1)\ t_2 \\
b_0 &=& \mathsf{map}\ (\lambda z.\, \mathsf{zipwith}(\ominus)\ z\ c_2)\ b_1 \ominus b_2 \\
r_0 &=& (r_1 \phi\ gemm\ (_, \mathsf{zipwith}(+))\ br_1\ tr_2) \ominus (NIL \phi\ r_2) \\
l_0 &=& (l_1 \phi\ gemm\ (_, \mathsf{zipwith}(+))\ bl_1\ tl_2) \ominus (NIL \phi\ l_2) \\
tr_0 &=& tr_1 \phi\ \mathsf{map}\ (\mathsf{zipwith}(+)\ (right'\ tr_1))\ tr_2 \\
br_0 &=& \mathsf{map}\ (\lambda z.\, \mathsf{zipwith}(+)\ z\ (top'\ br_2))\ br_1 \ominus br_2 \\
tl_0 &=& tl_1 \phi\ \mathsf{map}\ (\mathsf{zipwith}(+)\ (right'\ tl_1))\ tl_2 \\
bl_0 &=& \mathsf{map}\ (\lambda z.\, \mathsf{zipwith}(+)\ z\ (top'\ bl_2))\ bl_1 \ominus bl_2 \\
c_0 &=& \mathsf{zipwith}(+)\ c_1\ c_2 \\
ro_0 &=& (ro_1 \phi\ gemm\ (_, +)\ (right\ ro_1)\ (top\ ro_2)) \ominus (NIL \phi\ ro_2)
\end{array}$$

$$(s_1, t_1, b_1, r_1, l_1, tr_1, br_1, tl_1, bl_1, c_1, ro_1)\ (\bigtriangleup_1^{11} \ominus_i)\ (s_2, t_2, b_2, r_2, l_2, tr_2, br_2, tl_2, bl_2, c_2, ro_2)$$
$$= (s_0, t_0, b_0, r_0, l_0, tr_0, br_0, tl_0, bl_0, c_0, ro_0)$$

where

$$\begin{array}{lcl}
s_0 &=& \mathsf{zipwith}_4\ f_s\ s_1\ s_2\ r_1\ l_2 \\
 && \textbf{where}\ f_s\ s_1\ s_2\ r_1\ l_2 = (s_1 \phi\ gemm\ (_, +)\ r_1\ l_2) \ominus (NIL \phi\ s_2) \\
t_0 &=& \mathsf{zipwith}_4\ f_t\ t_1\ t_2\ tr_1\ tl_2 \\
 && \textbf{where}\ f_t\ t_1\ t_2\ tr_1\ tl_2 = (t_1 \phi\ gemm\ (_, +)\ tr_1\ tl_2) \ominus (NIL \phi\ t_2) \\
b_0 &=& \mathsf{zipwith}_4\ f_b\ b_1\ b_2\ br_1\ bl_2 \\
 && \textbf{where}\ f_b\ b_1\ b_2\ br_1\ bl_2 = (b_1 \phi\ gemm\ (_, +)\ br_1\ bl_2) \ominus (NIL \phi\ b_2) \\
r_0 &=& \mathsf{zipwith}_3\ f_r\ r_1\ r_2\ ro_2 \\
 && \textbf{where}\ f_r\ r_1\ r_2\ ro_2 = \mathsf{map}\ (+ro_2)\ r_1 \ominus r_2 \\
l_0 &=& \mathsf{zipwith}_3\ f_l\ l_1\ l_2\ ro_1 \\
 && \textbf{where}\ f_l\ l_1\ l_2\ ro_1 = l_1 \phi\ \mathsf{map}\ (ro_1+)\ l_2 \\
tr_0 &=& \mathsf{zipwith}\ f_{tr}\ tr_1\ tr_2 \\
 && \textbf{where}\ f_{tr}\ tr_1\ tr_2 = \mathsf{map}\ (+top'\ tr_2)\ tr_1 \ominus tr_2 \\
br_0 &=& \mathsf{zipwith}\ f_{br}\ br_1\ br_2 \\
 && \textbf{where}\ f_{br}\ br_1\ br_2 = \mathsf{map}\ (+top'\ br_2)\ br_1 \ominus br_2 \\
tl_0 &=& \mathsf{zipwith}\ f_{tl}\ tl_1\ tl_2 \\
 && \textbf{where}\ f_{tl}\ tl_1\ tl_2 = tl_1 \phi\ \mathsf{map}\ (right'\ tl_1+)\ tl_2 \\
bl_0 &=& \mathsf{zipwith}\ f_{bl}\ bl_1\ bl_2 \\
 && \textbf{where}\ f_{bl}\ bl_1\ bl_2 = bl_1 \phi\ \mathsf{map}\ (right'\ bl_1+)\ bl_2 \\
c_0 &=& (c_1 \phi\ gemm\ (_, +)\ (right\ c_1)\ (top\ c_2)) \ominus (NIL \phi\ c_2) \\
ro_0 &=& \mathsf{zipwith}(+)\ ro_1\ ro_2
\end{array}$$

Figure 4.6. Derived efficient program of $\mathsf{map}\,(\mathsf{map}\ sum) \circ rects$.

$$\mathsf{map}\ \mathit{max} \circ \mathsf{map}\,(\mathsf{map}\ \mathit{sum}) \circ \mathit{rects} = \pi_1 \circ (\!\![\, \triangle_1^{11} f_i'', \triangle_1^{11} \odot_i', \triangle_1^{11} \ominus_i' \,]\!\!)$$

where

$$\triangle_1^{11} f_i''\ |a| \quad = \quad (|a|, ||a||, ||a||, ||a||, ||a||, ||a||, ||a||, ||a||, ||a||, |a|, |a|)$$

$$\begin{array}{l}
(s_1, t_1, b_1, r_1, l_1, tr_1, br_1, tl_1, bl_1, c_1, ro_1)\ (\triangle_1^{11} \odot_i')\ (s_2, t_2, b_2, r_2, l_2, tr_2, br_2, tl_2, bl_2, c_2, ro_2) \\
\quad = (s_0, t_0, b_0, r_0, l_0, tr_0, br_0, tl_0, bl_0, c_0, ro_0) \\
\quad \textbf{where} \\
\qquad \begin{array}{lcl}
s_0 & = & (s_1 \phi\ \mathsf{map}\ \mathit{max}(\mathit{gemm}\ (_, \mathsf{zipwith}(+))\ b_1\ t_2)) \ominus (\mathit{NIL} \phi\ s_2) \\
t_0 & = & t_1 \phi\ \mathsf{map}\ (\mathsf{zipwith}(+)\ c_1)\ t_2 \\
b_0 & = & \mathsf{map}\ (\lambda z.\, \mathsf{zipwith}(\ominus)\ z\ c_2)\ b_1 \ominus b_2 \\
r_0 & = & (r_1 \phi\ \mathit{gemm}\ (_, \mathsf{zipwith}(+))\ br_1\ tr_2) \ominus (\mathit{NIL} \phi\ r_2) \\
l_0 & = & (l_1 \phi\ \mathit{gemm}\ (_, \mathsf{zipwith}(+))\ bl_1\ tl_2) \ominus (\mathit{NIL} \phi\ l_2) \\
tr_0 & = & tr_1 \phi\ \mathsf{map}\ (\mathsf{zipwith}(+)\ (\mathit{right}'\ tr_1))\ tr_2 \\
br_0 & = & \mathsf{map}\ (\lambda z.\, \mathsf{zipwith}(+)\ z\ (\mathit{top}'\ br_2))\ br_1 \ominus br_2 \\
tl_0 & = & tl_1 \phi\ \mathsf{map}\ (\mathsf{zipwith}(+)\ (\mathit{right}'\ tl_1))\ tl_2 \\
bl_0 & = & \mathsf{map}\ (\lambda z.\, \mathsf{zipwith}(+)\ z\ (\mathit{top}'\ bl_2))\ bl_1 \ominus bl_2 \\
c_0 & = & \mathsf{zipwith}(+)\ c_1\ c_2 \\
ro_0 & = & (ro_1 \phi\ \mathit{gemm}\ (_, +)\ (\mathit{right}\ ro_1)\ (\mathit{top}\ ro_2)) \ominus (\mathit{NIL} \phi\ ro_2)
\end{array}
\end{array}$$

$$\begin{array}{l}
(s_1, t_1, b_1, r_1, l_1, tr_1, br_1, tl_1, bl_1, c_1, ro_1)\ (\triangle_1^{11} \ominus_i')\ (s_2, t_2, b_2, r_2, l_2, tr_2, br_2, tl_2, bl_2, c_2, ro_2) \\
\quad = (s_0, t_0, b_0, r_0, l_0, tr_0, br_0, tl_0, bl_0, c_0, ro_0) \\
\quad \textbf{where} \\
\qquad \begin{array}{lcl}
s_0 & = & \mathsf{zipwith}_4\ f_s\ s_1\ s_2\ r_1\ l_2 \\
 & & \textbf{where}\ f_s\ s_1\ s_2\ r_1\ l_2 = s_1 \uparrow \mathit{max}(\mathit{gemm}\ (_, +)\ r_1\ l_2) \uparrow s_2 \\
t_0 & = & \mathsf{zipwith}_4\ f_t\ t_1\ t_2\ tr_1\ tl_2 \\
 & & \textbf{where}\ f_t\ t_1\ t_2\ tr_1\ tl_2 = (t_1 \phi\ \mathit{gemm}\ (_, +)\ tr_1\ tl_2) \ominus (\mathit{NIL} \phi\ t_2) \\
b_0 & = & \mathsf{zipwith}_4\ f_b\ b_1\ b_2\ br_1\ bl_2 \\
 & & \textbf{where}\ f_b\ b_1\ b_2\ br_1\ bl_2 = (b_1 \phi\ \mathit{gemm}\ (_, +)\ br_1\ bl_2) \ominus (\mathit{NIL} \phi\ b_2) \\
r_0 & = & \mathsf{zipwith}_3\ f_r\ r_1\ r_2\ ro_2 \\
 & & \textbf{where}\ f_r\ r_1\ r_2\ ro_2 = \mathsf{map}\ (+ro_2)\ r_1 \ominus r_2 \\
l_0 & = & \mathsf{zipwith}_3\ f_l\ l_1\ l_2\ ro_1 \\
 & & \textbf{where}\ f_l\ l_1\ l_2\ ro_1 = l_1 \phi\ \mathsf{map}\ (ro_1+)\ l_2 \\
tr_0 & = & \mathsf{zipwith}\ f_{tr}\ tr_1\ tr_2 \\
 & & \textbf{where}\ f_{tr}\ tr_1\ tr_2 = \mathsf{map}\ (+\mathit{top}'\ tr_2)\ tr_1 \ominus tr_2 \\
br_0 & = & \mathsf{zipwith}\ f_{br}\ br_1\ br_2 \\
 & & \textbf{where}\ f_{br}\ br_1\ br_2 = \mathsf{map}\ (+\mathit{top}'\ br_2)\ br_1 \ominus br_2 \\
tl_0 & = & \mathsf{zipwith}\ f_{tl}\ tl_1\ tl_2 \\
 & & \textbf{where}\ f_{tl}\ tl_1\ tl_2 = tl_1 \phi\ \mathsf{map}\ (\mathit{right}'\ tl_1+)\ tl_2 \\
bl_0 & = & \mathsf{zipwith}\ f_{bl}\ bl_1\ bl_2 \\
 & & \textbf{where}\ f_{bl}\ bl_1\ bl_2 = bl_1 \phi\ \mathsf{map}\ (\mathit{right}'\ bl_1+)\ bl_2 \\
c_0 & = & (c_1 \phi\ \mathit{gemm}\ (_, +)\ (\mathit{right}\ c_1)\ (\mathit{top}\ c_2)) \ominus (\mathit{NIL} \phi\ c_2) \\
ro_0 & = & \mathsf{zipwith}(+)\ ro_1\ ro_2
\end{array}
\end{array}$$

Figure 4.7. Derived efficient program of $\mathsf{map}\ \mathit{max} \circ \mathsf{map}\,(\mathsf{map}\ \mathit{sum}) \circ \mathit{rects}$

$$
\begin{array}{l}
mrs = \pi_1 \circ (\![\, \mathop{\triangle}_1^{11} f_i''', \mathop{\triangle}_1^{11} \odot_i'', \mathop{\triangle}_1^{11} \ominus_i'' \,]\!) \\
\quad \textbf{where} \\
\quad (\triangle_1^{11} f_i''')\ |a| = (a, |a|, |a|, |a|, |a|, |a|, |a|, |a|, |a|, |a|, |a|) \\
\quad (s_1, t_1, b_1, r_1, l_1, tr_1, br_1, tl_1, bl_1, c_1, ro_1)\ (\triangle_1^{11} \odot_i'')\ (s_2, t_2, b_2, r_2, l_2, tr_2, br_2, tl_2, bl_2, c_2, ro_2) \\
\qquad = (s_0, t_0, b_0, r_0, l_0, tr_0, br_0, tl_0, bl_0, c_0, ro_0) \\
\qquad \textbf{where} \\
\qquad\quad s_0 \ = (s_1 \uparrow max(\mathsf{zipwith}(+)\ b_1\ t_2) \uparrow s_2 \\
\qquad\quad t_0 \ = \mathsf{zipwith}_3\ \ f_t\ t_1\ c_1\ t_2 \\
\qquad\qquad \textbf{where}\ f_t\ t_1\ c_1\ t_2 = t_1 \uparrow (c_1 + t_2) \\
\qquad\quad b_0 \ = \mathsf{zipwith}_3\ \ f_b\ b_1\ c_2\ b_2 \\
\qquad\qquad \textbf{where}\ f_b\ b_1\ c_2\ b_2 = (b_1 + c_2) \uparrow b_2 \\
\qquad\quad r_0 \ = (r_1 \phi\ gemm\ (\uparrow, +)\ (tr\ br_1)\ tr_2) \ominus (NIL \phi\ r_2) \\
\qquad\quad l_0 \ = (l_1 \phi\ gemm\ (\uparrow, +)\ bl_1\ (tr\ tl_2)) \ominus (NIL \phi\ l_2) \\
\qquad\quad tr_0 = tr_1 \phi\ \mathsf{map}_c\ (\mathsf{zipwith}(+)\ (right\ tr_1))\ tr_2 \\
\qquad\quad br_0 = \mathsf{map}_c\ (\mathsf{zipwith}(+)\ (left\ br_2))\ br_1 \phi\ br_2 \\
\qquad\quad tl_0 = tl_1 \ominus \mathsf{map}_r\ (\mathsf{zipwith}(+)\ (bottom\ tl_1))\ tl_2 \\
\qquad\quad bl_0 = \mathsf{map}_r\ (\mathsf{zipwith}(+)\ (top\ bl_2))\ bl_1 \ominus bl_2 \\
\qquad\quad c_0 \ = \mathsf{zipwith}(+)\ c_1\ c_2 \\
\qquad\quad ro_0 = (ro_1 \phi\ gemm(_, +)\ (right\ ro_1)\ (top\ ro_2)) \ominus (NIL \phi\ ro_2) \\
\quad (s_1, t_1, b_1, r_1, l_1, tr_1, br_1, tl_1, bl_1, c_1, ro_1)\ (\triangle_1^{11} \ominus_i'')\ (s_2, t_2, b_2, r_2, l_2, tr_2, br_2, tl_2, bl_2, c_2, ro_2) \\
\qquad = (s_0, t_0, b_0, r_0, l_0, tr_0, br_0, tl_0, bl_0, c_0, ro_0) \\
\qquad \textbf{where} \\
\qquad\quad s_0 \ = s_1 \uparrow max(\mathsf{zipwith}(+)\ r_1\ l_2) \uparrow s_2 \\
\qquad\quad t_0 \ = (t_1 \phi\ gemm\ (\uparrow, +)\ tr_1\ tl_2) \ominus (NIL \phi\ t_2) \\
\qquad\quad b_0 \ = (b_1 \phi\ gemm\ (\uparrow, +)\ br_1\ bl_2) \ominus (NIL \phi\ b_2) \\
\qquad\quad r_0 \ = \mathsf{zipwith}_3\ \ f_r\ r_1\ r_2\ ro_2 \\
\qquad\qquad \textbf{where}\ f_r\ r_1\ r_2\ ro_2 = (r_1 + ro_2) \uparrow r_2 \\
\qquad\quad l_0 \ = \mathsf{zipwith}_3\ \ f_l\ l_1\ l_2\ ro_1 \\
\qquad\qquad \textbf{where}\ f_l\ l_1\ l_2\ ro_1 = l_1 \uparrow (ro_1 + l_2) \\
\qquad\quad tr_0 = \mathsf{map}_r\ (\mathsf{zipwith}(+)(top\ tr_2))\ tr_1 \ominus tr_2 \\
\qquad\quad br_0 = \mathsf{map}_r\ (\mathsf{zipwith}(+)(top\ br_2))\ br_1 \ominus br_2 \\
\qquad\quad tl_0 = tl_1 \phi\ \mathsf{map}_c\ (\mathsf{zipwith}(+)\ (right\ tl_1))\ tl_2 \\
\qquad\quad bl_0 = bl_1 \phi\ \mathsf{map}_c\ (\mathsf{zipwith}(+)\ (right\ bl_1))\ bl_2 \\
\qquad\quad c_0 \ = (c_1 \phi\ gemm(_, +)\ (right\ c_1)\ (top\ c_2)) \ominus (NIL \phi\ c_2) \\
\qquad\quad ro_0 = \mathsf{zipwith}(+)\ ro_1\ ro_2
\end{array}
$$

Figure 4.8. The final efficient program of maximum rectangle sum.

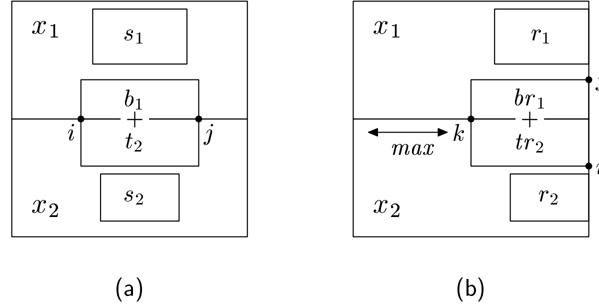


Figure 4.9. Computation of the operator for derived mrs.

In the final program, each of the eleven elements of the resulting tuple is the partial answer of its counterpart in Figure 4.4. Some parts are calculated with a general matrix multiplication, and others are updated with map and zipwith. For example, the element s_0, which is the solution of the maximum rectangle sum, is either the maximum of the solutions of upper and lower subarray, or the maximum of the solutions generated by combining partial answers of the top and the bottom rectangles, as shown in Figure 4.9-(a). Here, the rectangles that share the same edge are combined by zipwith$(+)$. Similarly, some elements of the array r_0, which is the partial solutions of the rectangles on the right edge, are calculated with a general matrix multiplication as shown in Figure 4.9-(b). Here, (i, j) element in the block is the maximum of $br_1(i, k) + tr_2(k, j)$ for all k.

Provided that we divide the input array into two parts evenly, this final parallel program uses only $O(n^3)$ addition operations as follows. For an $n \times n$ input array, the program's cost $T(n, n)$ satisfies the next equation with $gemm$'s cost $T_{gemm}(n, n)$ and some constants c_1 and c_2.

$$T(n, n) \;=\; 4T(n/2, n/2) + c_1 T_{gemm}(n/2, n/2) + c_2 n^2$$

Since the cost of the general matrix multiplication ($gemm$) is $O(n^3)$, the answer of the above equation is $T(n, n) = O(n^3)$. This is much better than the initial one. Moreover, since the general matrix multiplication $gemm(\uparrow, +)$ used in the program is a distance matrix multiplication, we can achieve subcubic cost with the special implementation of the $gemm(\uparrow, +)$ used in Takaoka's algorithm [Tak02]. However, the implementation is somewhat tricky, so that we cannot describe it with our skeletons.

Step 4. Optimizing Inner Functions

For our example, we may proceed to optimize the operators and functions such as f_i''', $\odot_i''$ and $\ominus_i''$ in the program of Step 3. Since they cannot be made efficient any more, we finish our derivation of an efficient parallel program.

4.4 Related Work

There have been several studies on the optimizations over multiple skeletons based on *fusion transformations* [GWL99, WL98, HIT02, MKI+04, GS06], which were studied in depth in the field of functional programming [Wad88, GLJ93]. In particular, general fusion optimizations on lists [GWL99, WL98, HIT02, MKI+04] have achieved good results both in theory and in practice.

Wedler and Lengauer [WL98] proposed fusion rules on a computation pattern called parallel linear recursion. Their parallel linear recursion can perform almost equivalent computation to accumulate, except that parallel linear recursion explicitly return a pair of a value and a list to allow arbitrary number of compositions. Their formalization is close to cellular automaton.

Gorlatch et al. [GWL99] proposed a set of fusion rules for skeletons. Their rules can deal with similar skeleton compositions that the above research can handle, although their rules are, if anything, case by case.

Chapter 5

Domain-Specific Optimization for Skeleton Programs

Although the general fusion optimizations so far are reasonably powerful, there is still large room for further optimizations. Due to the generality of the fusion transformations, some overheads in skeleton programs are left through the fusion optimizations. Also, effective optimization of complicated skeleton compositions by fusion often needs human insights to make efficient operators or functions used in the fused results.

The above problem can be solved when we make fusion optimizations specific to some domain, exploiting knowledge of the domain. This specific fusion optimization may include fusion rules specific to the domain, and shortcuts to the fused results with human insights.

In this chapter, we will concentrate on optimization of skeleton programs for computation involving neighbor elements. The computation can be categorized into two types: the computation involving a finite number of neighbor elements, such as filtering of sequences and images, the finite difference method, and some matrix-vector operations; and the computation involving an infinite number of neighbor elements, such as queries of interesting segments on lists and rectangles (sub-arrays) on two-dimensional arrays.

First, we will develop domain-specific fusion rules for computation involving a finite number of neighbor elements. To this end, we will introduce a new strategy for developing domain-specific fusion optimization of skeleton programs. Then, we will develop the fusion optimization demonstrating the strategy.

Next, we will proceed to optimization of skeleton programs involving an infinite number of elements. To this end, we will formalize the computation by nested reductions, and develop shortcut theorems to provide efficient algorithms to nested reductions by fusion.

5.1 General Strategy for Domain-Specific Fusion Optimization

In skeletal parallel programming, domain-specific programs are often developed with a fixed set of skeletons composed in a specific manner. Based on this observation, we propose the following strategy for developing domain-specific optimizations.

1. Designing a normal form that abstracts target specific computations.

2. Developing specific fusion rules that can completely transform skeleton compositions of specific patterns into the normal forms.

3. Providing efficient implementations of the normal form.

In designing a normal form, we should have the following requirements in mind. A normal form is specified to describe any computation of target programs but should not be too general. A normal form should be specific to the target programs, and should enable us to develop efficient implementation for it. In addition, a normal form should be closed under the fusion rules to maintain the result of optimization in the form.

Once we formalize a normal form with fusion rules and efficient implementation, we can perform the optimization easily: we first transform a skeleton program into the normal form with the fusion rules, and then we dispatch a suitable efficient implementation to the obtained normal form, in which the implementation may be selected according to some mathematical properties of parameters in the normal form.

The big difference from the existing general fusions is that our domain-specific fusions are closed under the target compositions. A big problem of the general fusions is that no one know how widely the general fusions can fuse skeletons in the given programs. They sometimes can not fuse any consecutive skeletons successfully. For example, no proposed general fusion can fuse parallel skeletons zip and reduce.

Another difference is that the optimized program results in implementation specific to the domain of target programs. The general fusions use general computation patterns as the results of fusions, which may cause significant inefficiency due to their generality.

5.2 Computation Involving a Finite Number of Neighbor Elements

We demonstrate our strategy by developing optimization of skeleton programs for computation that involves neighbor elements, which is often seen in scientific computations. In this section, we will concentrate on the computation for one-dimensional data structures, and use skeleton on lists shown in Chapter 2.

First, we will define the target domain as a pattern of skeleton compositions. Next, we will design the normal form to describe computations of the domain. Then, we will develop domain-specific fusion that transforms skeleton compositions of the specific pattern into normal forms. After that, we will design parallel implementation for the normal form. Finally, we will extend the target domain to handle reductions and accumulations.

The optimization developed in this section mainly focuses on domain-specific fusions, and has only one pattern of efficient implementations for the normal form. We will develop another optimization in the next section, which will focus on various theorems used to dispatch efficient implementations for the normal form under some conditions on parameters.

5.2.1 Target Skeleton Composition Patterns

Our targets are skeleton programs that involve neighbor elements using combination of $\mathsf{shift}_{\ll}$, $\mathsf{shift}_{\gg}$, zip and map. We also deal with skeleton programs that perform one accumulation or one reduction by scan' (scanr') or reduce on the result of the above programs. We define the former program as *Program*, a program with accumulation as $Program_S$, and a program with reduction as $Program_R$. A program of $Program_S$ can have arbitrary number of map after scan' or scanr'.

$$
\begin{aligned}
\textbf{data}\ Program\ \alpha = {} & \underline{\mathsf{map}}\ (\beta \to \alpha)\ (Program\ \beta) \\
\mid\ & \underline{\mathsf{shift}_{\ll}}\ \alpha\ (Program\ \alpha) \\
\mid\ & \underline{\mathsf{shift}_{\gg}}\ \alpha\ (Program\ \alpha) \\
\mid\ & \underline{\mathsf{zip}}\ (Program\ \beta)\ (Program\ \gamma) \\
\mid\ & [\alpha] \\
\textbf{data}\ Program_S\ \alpha = {} & \underline{\mathsf{scan}}\ (\alpha \to \alpha \to \alpha)\ \alpha\ (Program\ \alpha) \\
\mid\ & \underline{\mathsf{scanr}}\ (\alpha \to \alpha \to \alpha)\ \alpha\ (Program\ \alpha) \\
\mid\ & \underline{\mathsf{map}}\ (\beta \to \alpha)\ (Program_S\ \beta) \\
\textbf{data}\ Program_R\ \alpha = {} & \underline{\mathsf{reduce}}\ (\alpha \to \alpha \to \alpha)\ (Program\ \alpha)
\end{aligned}
$$

Here, $Program\ \alpha$ and $Program_S\ \alpha$ are programs that generate lists of elements of type α, while $Program_R\ \alpha$ is a program that generates a value of type α. Type β in $\underline{\mathsf{map}}$ is bound locally. Types β and γ in $\underline{\mathsf{zip}}$ are bound by the relation $\alpha = (\beta, \gamma)$. For simplicity, we do not use binding of variables in the above skeleton programs. Since the above skeleton programs are inputs for optimization algorithms, we distinguish skeletons in the above skeleton programs from skeletons used in algorithms by attaching underlines.

Evaluation of the above defined program is given by the following $eval_P$, $eval_{PS}$

and $eval_{PR}$, which evaluate the programs straightforwardly.

$$
\begin{aligned}
&eval_P :: Program\ \alpha \to [\alpha] \\
&eval_P\ (\underline{\mathsf{map}}\ f\ x) = \mathsf{map}\ f\ (eval_P\ x) \\
&eval_P\ (\underline{\mathsf{shift}}_{\ll}\ e\ x) = \mathsf{shift}_{\ll}\ e\ (eval_P\ x) \\
&eval_P\ (\underline{\mathsf{shift}}_{\gg}\ e\ x) = \mathsf{shift}_{\gg}\ e\ (eval_P\ x) \\
&eval_P\ (\underline{\mathsf{zip}}\ y\ z) = \mathsf{zip}\ (eval_P\ y)\ (eval_P\ z) \\
&eval_P\ (x) = x
\end{aligned}
$$

$$
\begin{aligned}
&eval_{PS} :: Program_S\ \alpha \to [\alpha] \\
&eval_{PS}\ (\underline{\mathsf{scan}}\ (\oplus)\ e\ x) = \mathsf{scan}'\ (\oplus)\ e\ (eval_P\ x) \\
&eval_{PS}\ (\underline{\mathsf{scanr}}\ (\oplus)\ e\ x) = \mathsf{scanr}'\ (\oplus)\ e\ (eval_P\ x) \\
&eval_{PS}\ (\underline{\mathsf{map}}\ f\ x) = \mathsf{map}\ f\ (eval_{PS}\ x)
\end{aligned}
$$

$$
\begin{aligned}
&eval_{PR} :: Program_R\ \alpha \to \alpha \\
&eval_{PR}\ (\underline{\mathsf{reduce}}\ (\oplus)\ x) = \mathsf{reduce}\ (\oplus)\ (eval_P\ x)
\end{aligned}
$$

In the rest of this section, we mainly focus on *Program* α to explain our idea of the domain-specific fusion. Then, we discuss $Program_S\ \alpha$ and $Program_R\ \alpha$ in Section 5.2.5 as extensions of the optimization of *Program* α.

A Running Example

An example target problem is computation of a numerical solution of differential equations by difference methods, in which each elements of the solution is computed form its neighboring elements. We will use a simple program for difference method as our running example.

Let's consider the following wave-equation as a concrete differential equation.

$$\frac{\partial u}{\partial t} = -C\frac{\partial u}{\partial x}$$

This equation describes propagation of waves. To calculate the propagation of waves, we replace differential terms by differences. The value of u at time n and at location i is denoted by u_i^n.

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} = \frac{-C}{\Delta x}\left(a_{-2}u_{i-2}^n + a_{-1}u_{i-1}^n + a_0 u_i^n + a_1 u_{i+1}^n\right)$$
$$\textbf{where}\ \ (a_{-2}, a_{-1}, a_0, a_1) = (1/6, -1, 1/2, 1/3)$$

Here, we use a difference computed from two elements on the left and an element on the right. Rearranging the above equation, we get the following recurrence equation.

$$u_i^{n+1} = c_{-2}u_{i-2}^n + c_{-1}u_{i-1}^n + c_0 u_i^n + c_1 u_{i+1}^n$$

Now, let's consider a program *next* to compute the values at the next time from the current values of u. Here, we will use simple boundary conditions: $u_{-1}^n = b_{l_0}$,

$u_0^n = b_{l_1}$ and $u_{N+1}^n = b_r$ for a fixed N. We can implement *next* using skeletons as follows.

$$
\begin{aligned}
next\ u = \mathbf{let}\ & v'_{-2} = \mathsf{map}\ (c_{-2}\times)\ (\mathsf{shift}_{\gg}\ b_{l_0}\ (\mathsf{shift}_{\gg}\ b_{l_1}\ u))\\
& v'_{-1} = \mathsf{map}\ (c_{-1}\times)\ (\mathsf{shift}_{\gg}\ b_{l_1}\ u)\\
& v'_{0} = \mathsf{map}\ (c_{0}\times)\ u\\
& v'_{1} = \mathsf{map}\ (c_{1}\times)\ (\mathsf{shift}_{\ll}\ b_{r}\ u)\\
& v_{-} = \mathsf{map}\ add\ (\mathsf{zip}\ v'_{-2}\ v'_{-1})\\
& v_{+} = \mathsf{map}\ add\ (\mathsf{zip}\ v'_{0}\ v'_{1})\\
\mathbf{in}\ & \mathsf{map}\ add\ (\mathsf{zip}\ v_{-}\ v_{+})
\end{aligned}
$$

Here, intermediate variables v'_{-2}, v'_{-1}, v'_0, and v'_1 are used for readability; we can easily remove these variables to make the program fit to the composition pattern *Program* of target programs. The correspondence of the program *next* and the above recurrence equation is as follows. First, we generate a list v'_{-2} corresponding to a sequence of the first terms $c_{-2}u_{i-2}^n$. To this end, we first apply two $\mathsf{shift}_{\gg}$ to shift the elements twice to the right, so that the shifted lists has the value u_{i-2}^n at the position of u_i^n in the input list u. Then, we use map to multiply the coefficient c_{-2} to the elements. Figure 5.1 illustrates the arrangement of shifted elements. Similarly, lists corresponding to the second through the fourth terms are generated by using $\mathsf{shift}_{\ll}$, $\mathsf{shift}_{\gg}$ and map. Then, zipping these four lists by zip and adding elements by $\mathsf{map}\ add$, we obtain the final result, of which ith element is $c_{-2}u_{i-2} + c_{-1}u_{i-1} + c_0u_i + c_1u_{i+1}$.

In the following sections, we will explain our idea by using this example program *next*.

5.2.2 Normal Form for the Domain

The first step of our strategy is to design a normal form that can describe any computation of target programs. In this section, we formalize a normal form that describes the computation of *Program* α. The objective of this normal form is to hold necessary information to compute the result, which will be collected from skeleton compositions by domain-specific fusions.

Example of Normal Form

Let's consider a single-loop computation of the running example *next*. By a simple observation, its computation can be divided into three parts according to forms of computations of elements: the computation of the center elements, which involves only elements of the given list; the computation of elements on the left edge, which involves constants introduced by $\mathsf{shift}_{\gg}$; and the computation of elements on the right edge, which involves constants introduced by $\mathsf{shift}_{\ll}$. Figure 5.1 illustrates the three parts.

In the computation of *next*, each element in the center part is computed by the following expression. Here, u denotes the element concerned, $u_{\ll 2}$ denotes its second left element, $u_{\ll 1}$ its left element, and $u_{\gg 1}$ its right element.

$$ce = \text{“}add(add(c_{-2} \times u_{\ll 2}, c_{-1} \times u_{\ll 1}), add(c_0 \times u, c_1 \times u_{\gg 1}))\text{”}$$

This expression is common among the computation of elements in the center part.

On the other hand, the leftmost element is computed by the following expression. Here, $u[\overrightarrow{k}]$ denotes the kth element of u from the left edge.

$$l_1 = \text{“}add(add(c_{-2} \times b_{l_0}, c_{-1} \times b_{l_1}), add(c_0 \times u[\overrightarrow{0}], c_1 \times u[\overrightarrow{1}]))\text{”}$$

This computation involves variables ($u[\overrightarrow{0}]$ and $u[\overrightarrow{1}]$) and constants (b_{l_0} and b_{l_1}) introduced by $\mathsf{shift}_{\gg}$. Similarly, computation of the second element and the rightmost element involves constants. Here, $u[\overleftarrow{k}]$ denotes kth element of u from the right edge.

$$\begin{aligned} l_2 &= \text{“}add(add(c_{-2} \times b_{l_1}, c_{-1} \times u[\overrightarrow{0}]), add(c_0 \times u[\overrightarrow{1}], c_1 \times u[\overrightarrow{2}]))\text{”} \\ r_1 &= \text{“}add(add(c_{-2} \times u[\overleftarrow{2}], c_{-1} \times u[\overleftarrow{1}]), add(c_0 \times u[\overleftarrow{0}], c_1 \times b_r))\text{”} \end{aligned}$$

Summarizing these observations, whole computation of *next* can be denoted by the following triple of expressions, i.e., computation trees.

- The list of computation trees for elements on the left edge, i.e., $[l_1, l_2]$.
- The common computation tree for elements in the center part, i.e., ce.
- The list of computation trees for elements on the right edge, i.e., $[r_1]$.

Figure 5.2 shows these computation trees. To obtain the result from this triple, we compute each element on the edges by its computation tree, and compute elements on the center part by the common computation tree against indices in a single loop.

Generally, such a triple denotes a computation of a target skeletal program. Thus, in the next section, we formalize this triple as a normal form of our target skeletal programs. Transformation of a skeleton program into a normal form is later shown in Section 5.2.3.

Definition of Normal Form for the Domain

We will define structure of our normal form. A normal form is defined as a triple of the following three as argued so far: a list of computation trees for elements on the left edge, a common computation tree for the center part, and a list of computation trees for elements on the right edge.

$$\textbf{type} \quad \mathit{NForm}\ \alpha \quad = \quad ([\mathit{Tree}\ \alpha], \mathit{Tree}\ \alpha, [\mathit{Tree}\ \alpha])$$

We denote this triple with special brackets like $[\![\, ls, zms, rs \,]\!]$. Here, ls is the list of computation trees for the left edge, zms is the common computation tree for the

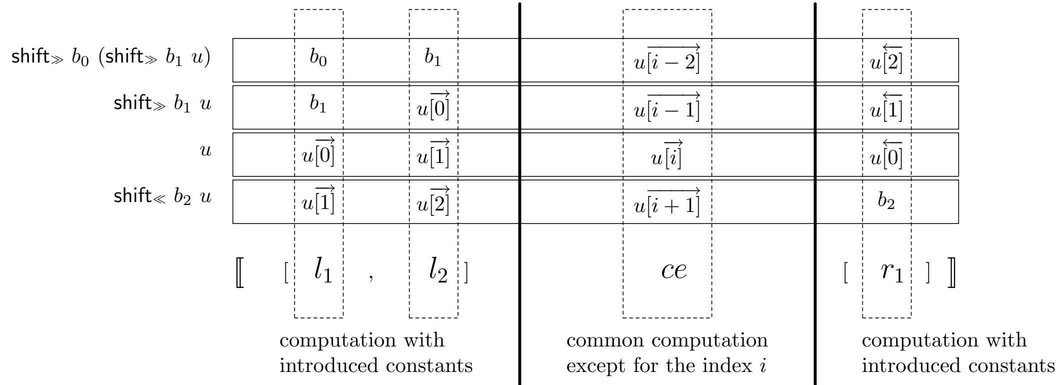


Figure 5.1. Lists arranged by shifts and regions of elements calculated by triples for *next*. A list of computation trees for elements on the left edge $[l_1, l_2]$. A common computation tree for elements in the center part *ce*. A list of computation trees for elements on the right edge $[r_1]$. Elements involved in the computation are boxed by dashed lines.

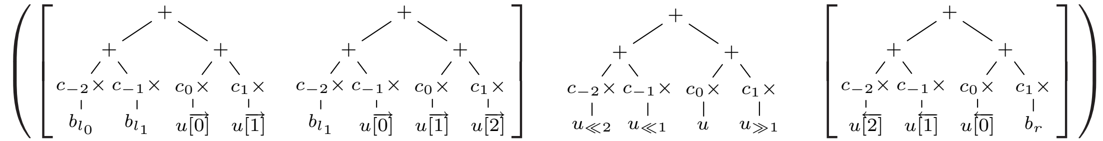


Figure 5.2. The triple describing computation of *next*.

center part, and *rs* is the list of computation trees for the right edge. Computation trees in normal forms are defined as follows.

$$
\begin{aligned}
\textbf{data}\ \mathit{Tree}\ \alpha &= \mathit{Node}\ ((\beta,\gamma) \to \alpha)\ (\mathit{Tree}\ \beta)\ (\mathit{Tree}\ \gamma) \\
&\mid\ \mathit{Leaf}_v\ (\beta \to \alpha)\ (\mathit{Var}\ \beta) \\
&\mid\ \mathit{Leaf}_c\ \alpha \\
\textbf{data}\ \mathit{Var}\ \alpha &= \mathit{Var}\ [\alpha]\ \mathit{Int} \\
&\mid\ \mathit{Fix}\ [\alpha]\ \mathit{Int}\ \mathit{Direction} \\
&\mid\ \mathit{Hole} \\
\textbf{data}\ \mathit{Direction} &= \mathit{FromL} \mid \mathit{FromR}
\end{aligned}
$$

A node of the tree holds the left and right children zipped by zip, and a composed function applied by successive maps. There are two kinds of leaves: Leaf_c denotes a constant introduced by $\mathsf{shift}_{\ll}$ or $\mathsf{shift}_{\gg}$, and Leaf_v denotes input lists and holds a composed function applied by successive maps. The data structure $\mathit{Var}\ \alpha$ denotes access to the input lists: *Var* represents index access of lists in the center part; and *Fix* represents fixed-index access in computations of edge elements. Therefore, *Var* holds the list and the amount of shifting, while *Fix* holds the following three:

the input list to be accessed, the index, and the origin of the index. The origin of the fixed index is specified by *Direction*. A special value *Hole* will be used in formalization of parallel implementation, which indicates that an element of other processor will fill out the hole to complete the tree.

For example, expression $add(c_{-2}{\times}b_{l_1}, c_{-1}{\times}u[\overrightarrow{0}])$ is described as follows.

$$Node\ add\ (Leaf_c\ (c_{-2}{\times}b_{l_1}))\ (Leaf_v\ (c_{-1}\times)\ (Fix\ u\ 0\ FromL))$$

A part $c_{-2}{\times}u_{\ll 2}$ of common tree is described as follows.

$$Leaf_v\ (c_{-2}\times)\ (Var\ u\ (-2))$$

Note that negative value of the amount of shifting means shifting to the left.

Semantics of Normal Form

We will give a semantics of the normal form by sequential evaluation of the normal form. In the evaluation, elements in the center part are computed by a single loop with the common computation tree, and each element on both edges is computed by its own computation tree.

First, we define the evaluation of computation trees as evaluation function $eval_T$.

$$\begin{array}{lcl} eval_T :: Tree\ \alpha \to Int \to \alpha & & \\ eval_T\ (Node\ f\ l\ r)\ i & = & f\ (eval_T\ l\ i, eval_T\ r\ i) \\ eval_T\ (Leaf_v\ f\ v)\ i & = & f\ (eval_V\ v\ i) \\ eval_T\ (Leaf_c\ c)\ i & = & c \end{array}$$

The evaluation function $eval_T$ performs computation according to the definition of computation trees, applying functions stored in the trees to the elements of the input lists along with the tree structures. It flows the index of the element being computed through the evaluation to the other evaluation function $eval_V$. The auxiliary function $eval_V$ processes index accessing of input lists. The result of $eval_V$ for *Hole* is not defined.

$$\begin{array}{lcl} eval_V :: Var\ \alpha \to Int \to \alpha & & \\ eval_V\ (Var\ u\ s)\ i & = & \mathsf{at}\ u\ (i-s) \\ eval_V\ (Fix\ u\ s\ FromL)\ _ & = & \mathsf{at}\ u\ s \\ eval_V\ (Fix\ u\ s\ FromR)\ _ & = & \mathsf{at}\ (reverse\ u)\ s \end{array}$$

Here, function at returns the element at the given index, and defined as follows.

$$\mathsf{at}\ i\ (a : x) = \mathbf{if}\ i = 0\ \mathbf{then}\ a\ \mathbf{else}\ \mathsf{at}\ (i-1)\ x$$

Next, we will define the evaluation function $eval_{T0}$ for elements on the left and right edges. The evaluation is defined by $eval_T$ ignoring the index.

$$\begin{array}{lcl} eval_{T0} :: Tree\ \alpha \to \alpha & & \\ eval_{T0}\ x & = & eval_T\ x\ 0 \end{array}$$

Then, using the above evaluation functions, we define a sequential program *eval* that evaluates the normal form. Here, the lengths of involved lists are supposed to be n.

$$
\begin{array}{l}
eval :: NForm\ \alpha \to [\alpha] \\
eval\ [\![\, ls, zms, rs \,]\!] \;\; = \;\; \mathsf{map}\ eval_{T0}\ ls +\!\!+ \mathsf{map}\ (eval_T\ zms)\ idces +\!\!+ \mathsf{map}\ eval_{T0}\ rs \\
\quad\quad \textbf{where} \;\; l = length\ ls \;;\; r = length\ rs \\
\quad\quad\quad\quad\quad\quad idces = [l..(n-r-1)]
\end{array}
$$

Each element on both edges is calculated by its own computation tree using $eval_{T0}$ defined above. The center part is calculated by a single loop ($\mathsf{map}\ (eval_T\ zms)$) with the common computation tree.

5.2.3 Fusion Rules for Transformation to a Normal Form

The second step of our strategy is to define fusion rules to transform skeleton compositions to normal forms. These rules should be able to transform any of the target skeleton compositions to normal forms. In this section, we give fusion rules to transform a skeleton program *Program* (see Section 5.2.1) into the normal form defined so far.

We first give formal definition of the fusion rules. Then, we will show example transformations with the fusion rules.

Transformation with Fusion Rules

We will define function *compile* that transforms a skeleton program into the normal form by one-by-one application of fusion rules.

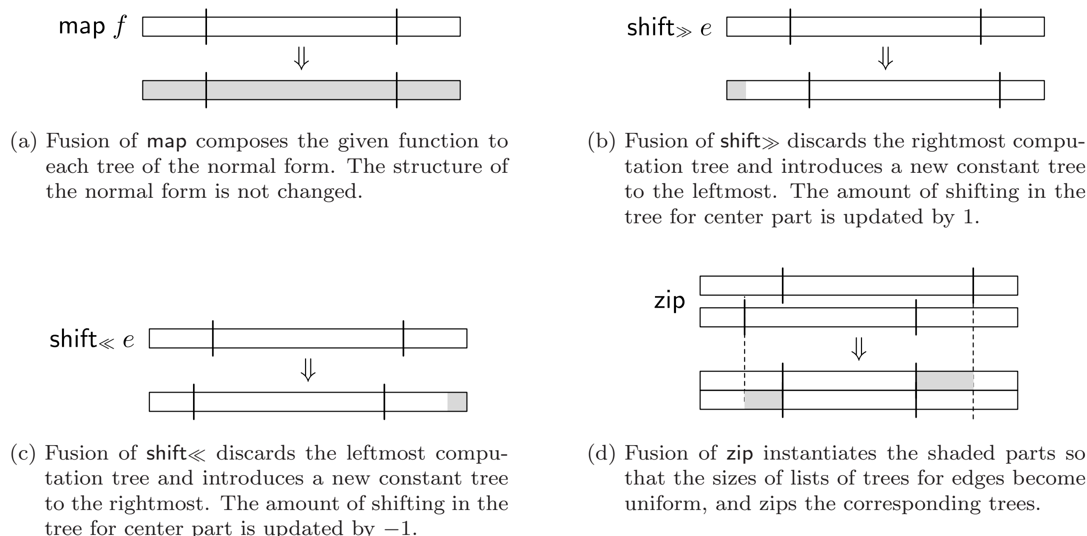
$$
\begin{array}{lll}
compile :: Program\ \alpha \to NForm\ \alpha & & \\
compile\ (\underline{\mathsf{map}}\ f\ x) & = & fuseMap\ f\ (compile\ x) \\
compile\ (\underline{\mathsf{shift}}_{\ll}\ e\ x) & = & fuseShift_{\ll}\ e\ (compile\ x) \\
compile\ (\underline{\mathsf{shift}}_{\gg}\ e\ x) & = & fuseShift_{\gg}\ e\ (compile\ x) \\
compile\ (\underline{\mathsf{zip}}\ x\ y) & = & fuseZip\ (compile\ x)\ (compile\ y) \\
compile\ (x) & = & [\![\, [\,], Leaf_v\ id\ (Var\ x\ 0), [\,] \,]\!]
\end{array}
$$

Fusion rules for skeletons are defined as functions *fuseMap*, $fuseShift_{\ll}$, $fuseShift_{\gg}$, and *fuseZip*, which will be define below. Figure 5.3 illustrates the fusion rules.

Fusion Rule for map

Fusion of skeleton map is performed by composing the given function to roots of computation trees.

$$
\begin{array}{l}
fuseMap :: (\alpha \to \beta) \to NForm\ \alpha \to NForm\ \beta \\
fuseMap\ f\ [\![\, ls, zms, rs \,]\!] = [\![\, \mathsf{map}\ (comp\ f)\ ls, comp\ f\ zms, \mathsf{map}\ (comp\ f)\ rs \,]\!]
\end{array}
$$



(a) Fusion of map composes the given function to each tree of the normal form. The structure of the normal form is not changed.

(b) Fusion of shift$_\gg$ discards the rightmost computation tree and introduces a new constant tree to the leftmost. The amount of shifting in the tree for center part is updated by 1.

(c) Fusion of shift$_\ll$ discards the leftmost computation tree and introduces a new constant tree to the rightmost. The amount of shifting in the tree for center part is updated by -1.

(d) Fusion of zip instantiates the shaded parts so that the sizes of lists of trees for edges become uniform, and zips the corresponding trees.

Figure 5.3. An image of fusion rules. Rectangles represent lists. The three parts separated by vertical lines correspond to the triple of the normal form. Changed parts in the resulting normal form are shaded.

Composition of a function is given by the following *comp*.

$$
\begin{aligned}
&\mathit{comp} :: (\alpha \to \beta) \to \mathit{Tree}\ \alpha \to \mathit{Tree}\ \beta \\
&\mathit{comp}\ f\ (\mathit{Node}\ g\ l\ r) = \mathit{Node}\ (f \circ g)\ l\ r \\
&\mathit{comp}\ f\ (\mathit{Leaf}_v\ g\ v) \ \ = \mathit{Leaf}_v\ (f \circ g)\ v \\
&\mathit{comp}\ f\ (\mathit{Leaf}_c\ c) \quad\ \ = \mathit{Leaf}_c\ (f\ c)
\end{aligned}
$$

For non-constant roots, *comp* composes the given function to the function held in the root. For constant roots (i.e. constant leaves), *comp* applies the given function to the constant to generate a new constant root.

Fusion Rules for shift$_\ll$ and shift$_\gg$

Fusion of skeletons shift$_\ll$ and shift$_\gg$ is performed by insertion and deletion of the leftmost and the rightmost trees, and update of the amount of shifting.

$$
\begin{aligned}
&\mathit{fuseShift}_\ll :: \alpha \to \mathit{NForm}\ \alpha \to \mathit{NForm}\ \alpha \\
&\mathit{fuseShift}_\ll\ e\ [\![\, \mathit{ls}, \mathit{zms}, \mathit{rs}\,]\!] = [\![\, \mathsf{tail}\ \mathit{ls}, \mathit{slide}\ (-1)\ \mathit{zms}, \mathit{rs} +\!\!+ [\mathit{Leaf}_c\ e]\,]\!] \\
&\mathit{fuseShift}_\gg :: \alpha \to \mathit{NForm}\ \alpha \to \mathit{NForm}\ \alpha \\
&\mathit{fuseShift}_\gg\ e\ [\![\, \mathit{ls}, \mathit{zms}, \mathit{rs}\,]\!] = [\![\, [\mathit{Leaf}_c\ e] +\!\!+ \mathit{ls}, \mathit{slide}\ 1\ \mathit{zms}, \mathsf{init}\ \mathit{rs}\,]\!]
\end{aligned}
$$

For the left-shift, the fusion discards the leftmost computation tree, and introduces the constant computation tree of the given constant e to the rightmost. Then, it

updates the amount of shifting in the common computation tree for the center part. This update is performed by the following *slide*.

$$\begin{array}{l}
slide :: Int \to Tree\ \alpha \to Tree\ \alpha \\
slide\ d\ (Node\ f\ l\ r) = Node\ f\ (slide\ d\ l)\ (slide\ d\ r) \\
slide\ d\ (Leaf_v\ f\ (Var\ x\ s)) = Leaf_v\ f\ (Var\ x\ (s + d)) \\
slide\ d\ x = x
\end{array}$$

The right-shift is similar to the left-shift.

Fusion Rule for zip

Fusion of skeleton zip needs unification of the lengths of lists of computation trees for edges. Thus, the common trees for the center parts are instantiated to expand the lists of edge computation trees. Then, each pair of corresponding trees is zipped by introducing a new node with *id* function.

$$\begin{array}{l}
fuseZip :: NForm\ \alpha \to NForm\ \beta \to NForm\ (\alpha, \beta) \\
fuseZip\ [\![\, ls_1, zms_1, rs_1 \,]\!]\ [\![\, ls_2, zms_2, rs_2 \,]\!] \\
\quad = \mathbf{let}\ zms = Node\ id\ zms_1\ zms_2 \\
\quad\quad\quad\quad ls = trim\ FromL\ ls_1\ ls_2\ zms_1\ zms_2 \\
\quad\quad\quad\quad rs = trim\ FromR\ (reverse\ rs_1)\ (reverse\ rs_2)\ zms_1\ zms_2 \\
\quad\quad\ \mathbf{in}\ [\![\, ls, zms, reverse\ rs \,]\!]
\end{array}$$

The function *trim* defined below unifies the lengths of lists by instantiating common tree zms_k by a function *insts* defined below.

$$\begin{array}{l}
trim :: Direction \to [Tree\ \alpha] \to [Tree\ \beta] \quad \to Tree\ \alpha \to Tree\ \beta \to [Tree\ (\alpha, \beta)] \\
trim\ d\ ts_1\ ts_2\ zms_1\ zms_2 \\
\quad = \mathbf{let}\ n_1 = length\ ts_1\ ;\ n_2 = length\ ts_2 \\
\quad\quad\quad\quad (ts_1', ts_2') = (ts_1 +\!\!+\ insts\ d\ zms_1\ n_1\ n_2, ts_2 +\!\!+\ insts\ d\ zms_2\ n_2\ n_1) \\
\quad\quad\ \mathbf{in}\ \mathsf{zipwith}\ (Node\ id)\ ts_1'\ ts_2'
\end{array}$$

The instantiation is performed by the following *insts* and *inst*.

$$\begin{array}{l}
insts :: Direction \to Tree\ \alpha \to Int \to Int \to [Tree\ \alpha] \\
insts\ d\ zms\ s\ e = \mathsf{map}\ (inst\ d\ zms)\ [s..(e-1)] \\
\\
inst\ d\ (Node\ f\ l\ r)\ i = Node\ f\ (inst\ d\ l\ i)\ (inst\ d\ r\ i) \\
inst\ d\ (Leaf_v\ f\ (Var\ x\ s))\ i = \mathbf{let}\ s' = \mathbf{case}\ d\ \mathbf{of}\ FromL \to s;\ FromR \to -s \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\ \mathbf{in}\ Leaf_v\ f\ (Fix\ x\ (-s' + i)\ d) \\
inst\ d\ x\ i = x
\end{array}$$

Completeness of the Fusion Rules

These four fusion rules and the base case rule can transform any skeleton program defined by *Program* into the normal form. We conclude this fact as a theorem.

Theorem 5.1. Any skeleton program defined by *Program* can be transformed into the normal form by using the four fusion rules and the base case rule, while keeping the result of the whole computation. That is, the following equation holds for any *prog* of *Program*.

$$eval_P\ prog = eval\ (compile\ prog)$$

Proof. This is proved by induction on the structure of *Program*. The base case is shown by the transformation of an input list. Induction cases are shown by the four fusion rules.

Complete proof is shown in Appendix B. □

This completeness of fusion rules is the sharp contrast of our domain-specific fusions to general fusions, which do not determine how widely they are applicable.

Example Transformation

As a brief explanation of the rules, we transform the example *next* into the normal form. For readability, we use a brief notation used in the previous examples instead of the actual data structures defined so far. In the rest of this section, we will use $\Rightarrow$ to show the transformation; the left hand side is the structure being transformed, and the right hand side is the resulting normal form.

The most simplest case is the transformation of the argument list u. List u needs only the common computation tree $u[\overrightarrow{i}]$ that is just the element of u.

$$u \Rightarrow [\![\,[\,], id\ u, [\,]\,]\!]$$

Here, $id\ u$ is the brief notation of $Leaf_v\ id\ (Var\ u\ 0)$.

Next, we transform $\mathsf{shift}_{\gg}\ b_{l_1}\ u$. This $\mathsf{shift}_{\gg}$ introduces the constant b_{l_1} to the leftmost element. Thus, a new computation tree of the constant b_{l_1} is introduced to the normal form.

$$\mathsf{shift}_{\gg}\ b_{l_1}\ [\![\,[\,], id\ u, [\,]\,]\!] \Rightarrow [\![\,[b_{l_1}], id\ u_{\gg 1}, [\,]\,]\!]$$

Also, the amount of shifting in the common tree is updated by 1.

Then, we fuse $\mathsf{map}\ (c_{-1}\times)$ to the above result.

$$\mathsf{map}\ (c_{-1}\times)\ [\![\,[b_{l_1}], id\ u_{\gg 1}, [\,]\,]\!] \Rightarrow [\![\,[c_{-1}\times b_{l_1}], c_{-1}\times u_{\gg 1}, [\,]\,]\!]$$

The constant b_{l_1} is replaced by $c_{-1} \times b_{l_1}$, and the function $(c_{-1}\times)$ is composed to id held in the root of the common tree. Since id is the identity of function compositions, id is removed in the result.

Similarly, other applications of $\mathsf{shift}_{\gg}$, $\mathsf{shift}_{\ll}$ and map result in the following normal forms.

$$\begin{array}{l}
\mathsf{map}\ (c_0\times)\ u \Rightarrow [\![\,[\,], c_0 \times u, [\,]\,]\!] \\
\mathsf{map}\ (c_1\times)\ (\mathsf{shift}_{\ll}\ b_r\ u) \Rightarrow [\![\,[\,], c_1 \times u_{\ll 1}, [c_1 \times b_r]\,]\!] \\
\mathsf{map}\ (c_{-2}\times)\ (\mathsf{shift}_{\gg}\ b_{l_0}\ (\mathsf{shift}_{\gg}\ b_{l_1}\ u)) \Rightarrow [\![\,[c_{-2} \times b_{l_0}, c_{-2} \times b_{l_1}], c_{-2} \times u_{\gg 2}, [\,]\,]\!]
\end{array}$$

In the second transformation, a new tree is introduced by $\mathsf{shift}_{\ll}$ to the rightmost, and the amount of shifting in the common tree is updated by 1 to the left. In the last transformation, there are two computation trees for elements on the left edge, and the amount of shifting in the common tree becomes 2. These are introduced by two $\mathsf{shift}_{\gg}$s.

Next, we perform fusion of zip to transform zip $v'_0\ v'_1$, i.e.

$$\mathsf{zip}\ \ [\![\,[\,], c_0 \times u, [\,]\,]\!]\ \ [\![\,[\,], c_1 \times u_{\ll 1}, [c_1 \times b_r]\,]\!].$$

Since the lengths of lists of two normal forms to be zipped are not the same (i.e. $[\,]$ and $[c_1 \times b_r]$ for the right edges), we have to unify the lengths by instantiating the common trees. Instantiation means fixing the indices in the common trees for elements on the edges. The result of instantiation and zip is as follows.

$$[\![\,[\,], (c_0 \times u, c_1 \times u_{\ll 1}), [(c_0 \times u\overleftarrow{[0]}, c_1 \times b_r)]\,]\!]$$

Here, instantiation of the common tree of the first normal form $c_0 \times u$ results in $c_0 \times u\overleftarrow{[0]}$, and it is zipped with the rightmost tree of the second normal form to make the new rightmost tree.

Similarly, we obtain the following normal form from zip $v'_{-2}\ v'_{-1}$.

$$[\![\ \ [(c_{-2} \times b_{l_0}, c_{-1} \times b_{l_1}), (c_{-2} \times b_{l_1}, c_{-1} \times u\overrightarrow{[0]})], (c_{-2} \times u_{\gg 2}, c_{-1} \times u_{\gg 1}), [\,]\ \ ]\!]$$

Continuing these fusions, we finally obtain the following normal form for the example *next*.

$$\begin{aligned}
[\![\ \ [\ & add(add(c_{-2} \times b_{l_0}, c_{-1} \times b_{l_1}), add(c_0 \times u\overrightarrow{[0]}, c_1 \times u\overrightarrow{[1]})), \\
& add(add(c_{-2} \times b_{l_1}, c_{-1} \times u\overrightarrow{[0]}), add(c_0 \times u\overrightarrow{[1]}, c_1 \times u\overrightarrow{[2]}))], \\
& add(add(c_{-2} \times u_{\ll 2}, c_{-1} \times u_{\ll 1}), add(c_0 \times u, c_1 \times u_{\gg 1})), \\
& [add(add(c_{-2} \times u\overleftarrow{[2]}, c_{-1} \times u\overleftarrow{[1]}), add(c_0 \times u\overleftarrow{[0]}, c_1 \times b_r))]\,]\!]
\end{aligned}$$

5.2.4 Parallel Implementation of Normal Form

The third step of our strategy is to design parallel implementation of the normal form. In this section, we explain the parallel implementation of the normal form. The implementation of the normal form is not necessarily a skeleton composition.

Based on parallel implementation of existing skeletons [MIEH06], we consider parallel implementation of the normal form consisting of four steps: (1) distribution of a normal form (input lists), (2) the first local computation, (3) global communication, (4) the second local computation. We explain the idea of parallel implementation using the example *next*. Note that gathering and redistribution of the result of the computation may be canceled when the result will be used as the input of the next computation. Figure 5.4 shows an image of parallel implementation of

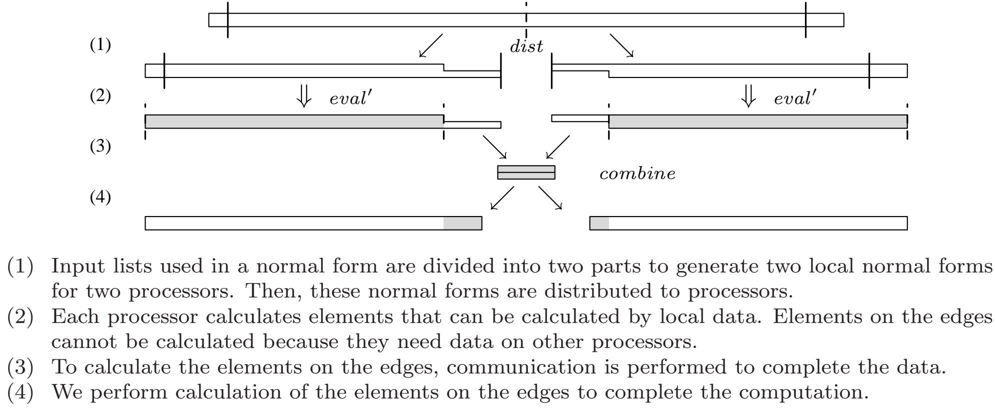


Figure 5.4. An image of parallel implementation of the normal form (two processors).

the normal form using two processors. We will explain the idea with two processors here, and will formalize its generalization later.

First, we distribute the normal form among processors. The input list u is divided into two parts: $u = u_1 +\!\!+ u_2$. There are two processors, and each processor has a part of the divided list. Note that u may already be distributed and the distribution phase may be skipped when u is the result of another computation of a normal form. Letting the original normal form be $[\![\, ls, zms, rs \,]\!]$, each processor has one of the following distributed normal forms.

$$nf_1 = [\![\, ls, zms_1, [\,] \,]\!] \ ; \ nf_2 = [\![\, [\,], zms_2, rs \,]\!]$$

Here, the list of computation trees for the left edge is held in the first normal form nf_1, while the list of computation trees for the right edge is held in the last normal form nf_2. The common computation tree zms_k of the normal form nf_k is created from zms by replacing the input list u with the part of the list u_k. Thus, for the example *next*, the distributed normal form nf_k is as follows.

$$add(add(c_{-2} \times u_{k \ll 2}, c_{-1} \times u_{k \ll 1}), add(c_0 \times u_k, c_1 \times u_{k \gg 1}))$$

Next, in the first local step, processors calculate own partial results in parallel. Since elements on the edges of the distributed normal forms needs elements of both u_1 and u_2, these elements cannot be calculated in this phase. For example, the rightmost element calculated by nf_1 needs an element of u_2, which is underlined in the following expression.

$$add(add(c_{-2} \times u_1\overleftarrow{[2]}, c_{-1} \times u_1\overleftarrow{[1]}), add(c_0 \times u_1\overleftarrow{[0]}, c_1 \times \underline{u_2\overrightarrow{[0]}}))$$

Since these elements cannot be calculated in this local phase, computation trees for these elements are held until the global computation phase. Here, we will put holes

$\bullet$ to the places of elements that are not available in this phase.

$$add(add(c_{-2}{\times}u_1\overleftarrow{[2]}, c_{-1}{\times}u_1\overleftarrow{[1]}), add(c_0{\times}u_1\overleftarrow{[0]}, c_1{\times}\bullet))$$

Similarly, the left edge of nf_2 generates the following computation tree with holes.

$$add(add(c_{-2}{\times}\bullet, c_{-1}{\times}\bullet), add(c_0{\times}\bullet, c_1{\times}u_2\overrightarrow{[0]}))$$

Note that this tree complement the previous tree generated from nf_1; merging them we have the complete tree. Generally, letting the maximum amounts of shifting to the left and the right be l and r, the number of computation trees with holes generated on the break of division is $l + r$. Trees on both sides of the break are complementary to each other.

Third, in the global communication step, neighboring processors communicate incomplete trees with holes to each other to complete the trees. The first local computation can hide the time of this communication phase.

Fourth, in the second local step, each processor calculates the elements on its edges with the completed trees to finish the computation.

In our example program *next*, the first normal form nf_1 generates the following trees with holes.

$$\begin{array}{l}add(add(c_{-2}{\times}u_1\overleftarrow{[2]}, c_{-1}{\times}u_1\overleftarrow{[1]}), add(c_0{\times}u_1\overleftarrow{[0]}, c_1{\times}\bullet))\\ add(add(c_{-2}{\times}u_1\overleftarrow{[1]}, c_{-1}{\times}u_1\overleftarrow{[0]}), add(c_0{\times}\bullet, c_1{\times}\bullet))\\ add(add(c_{-2}{\times}u_1\overleftarrow{[0]}, c_{-1}{\times}\bullet), add(c_0{\times}\bullet, c_1{\times}\bullet))\end{array}$$

The second normal form nf_2 generates the following trees.

$$\begin{array}{l}add(add(c_{-2}{\times}\bullet, c_{-1}{\times}\bullet), add(c_0{\times}\bullet, c_1{\times}u_2\overrightarrow{[0]}))\\ add(add(c_{-2}{\times}\bullet, c_{-1}{\times}\bullet), add(c_0{\times}u_2\overrightarrow{[0]}, c_1{\times}u_2\overrightarrow{[1]}))\\ add(add(c_{-2}{\times}\bullet, c_{-1}{\times}u_2\overrightarrow{[0]}), add(c_0{\times}u_2\overrightarrow{[1]}, c_1{\times}u_2\overrightarrow{[2]}))\end{array}$$

Zipping these trees, we obtain the following complete computation trees for elements on the break.

$$\begin{array}{l}add(add(c_{-2}{\times}u_1\overleftarrow{[2]}, c_{-1}{\times}u_1\overleftarrow{[1]}), add(c_0{\times}u_1\overleftarrow{[0]}, c_1{\times}u_2\overrightarrow{[0]}))\\ add(add(c_{-2}{\times}u_1\overleftarrow{[1]}, c_{-1}{\times}u_1\overleftarrow{[0]}), add(c_0{\times}u_2\overrightarrow{[0]}, c_1{\times}u_2\overrightarrow{[1]}))\\ add(add(c_{-2}{\times}u_1\overleftarrow{[0]}, c_{-1}{\times}u_2\overrightarrow{[0]}), add(c_0{\times}u_2\overrightarrow{[1]}, c_1{\times}u_2\overrightarrow{[2]}))\end{array}$$

After these four steps, these completed results are gathered to the root processor, or become a new input to another normal form. Distribution of input will be skipped in the latter case.

Summarizing the above ideas, we get the following general structure of parallel implementation of the normal form.

$$parEval = globalReduction \circ \mathsf{map}\ (localEval) \circ dist$$

First, a normal form is distributed among processors by *dist* (step (1)). Next, each processor evaluates a part of distributed normal forms by *localEval* (step (2)). Then, these results are combined globally by *globalReduction* (step (3) and (4)). Local computation *localEval* takes a distributed normal form, and generates a triple of its partial result (pls, cs, prs), where *cs* is the calculated elements on the center part, and *pls* and *prs* are computation trees with holes for elements on left and right edges. Global computation *globalReduction* communicates partial results (pls_1, cs_1, prs_1) and (pls_2, cs_2, prs_2) of both sides of an edge, then generates a new partial result $(pls_1, cs_1 +\!\!+ \mathit{glue}\ prs_1\ pls_2 +\!\!+ cs_2, prs_2)$. Here, *glue* denotes the complement process. Then, the final result is obtained by extracting the center result of the final partial result.

We will complete the definition of the general computation in the next section.

Formalization of Parallel Implementation of Normal Form

We formalize the parallel implementation explained in the previous section. In the rest of this section, p means the number of processors and n means the length of the lists involved in the computation.

First, we define the distribution *dist* that divides a normal form into p parts.

$$
\begin{array}{l}
\mathit{dist} :: \mathit{Int} \to \mathit{NForm}\ \alpha \to [\mathit{NForm}\ \alpha] \\
\mathit{dist}\ p\ [\![\ \mathit{ls}, \mathit{zms}, \mathit{rs}\ ]\!] = \textbf{let}\ \mathit{divs} = \mathit{division}\ p\ n \\
\qquad\qquad\qquad\qquad\qquad\quad \mathit{zmss} = \mathit{distribute}\ \mathit{zms}\ \mathit{divs} \\
\qquad\qquad\qquad\qquad\qquad\quad \mathit{lss} = \mathit{ls} : \mathit{dupl}\ (p-1)\ [\,] \\
\qquad\qquad\qquad\qquad\qquad\quad \mathit{rss} = \mathit{dupl}\ (p-1)\ [\,] +\!\!+ [\mathit{rs}] \\
\qquad\qquad\qquad\qquad\quad\ \textbf{in}\ \mathsf{zip}_3\ \mathit{lss}\ \mathit{zmss}\ \mathit{rss}
\end{array}
$$

Here, function *dupl* generates a list of the given element of the given length, and defined as follows.

$$
\begin{array}{l}
\mathit{dupl}\ 0\ a = [\,] \\
\mathit{dupl}\ n\ a = [a] +\!\!+ \mathit{dupl}\ (n-1)\ a
\end{array}
$$

Function *division* $p\ n$ calculates the division of input lists, and *distribute zms divs* distributes the input lists according to the division. This process generates distributes a computation tree zms_i that is generated by replacing a list $u\ (= u_1 +\!\!+ \cdots +\!\!+ u_i +\!\!+ \cdots +\!\!+ u_p)$ in the original tree *zms* with u_i, and zms_i is distributed to ith processor and held in the normal form nf_i (this is generated by $\mathsf{zip}_3\ lss\ zmss\ rss$). The list of computation trees for the left edge is held in the first normal form nf_1, while the list of computation trees for the right edge is held in the last normal form nf_p.

Next, we will define the first local computation *localEval*. To this end, we define the triple of the partial result *PResult* of the local computation, and define some useful functions.

$$
\textbf{type}\quad \mathit{PResult}\ \alpha\ \beta \quad = \quad ([\mathit{Tree}\ \alpha], \beta, [\mathit{Tree}\ \alpha])
$$

We use a special brackets $\langle\!\langle\, pls, cs, prs \,\rangle\!\rangle$ to denote a partial result. Here, cs is the computed elements on the center part, and pls and prs are computation trees with holes for elements on left and right edges. We abstract the type of calculated elements on the center part as β for generality.

An auxiliary function $maxShift$ is defined as follows to calculate the maximum amount of shifting.

$$
\begin{array}{lll}
maxShift :: Tree\ \alpha \rightarrow (Int, Int) & & \\
maxShift\ (Node\ _\ l\ r) & = & \textbf{let}\ (l_1, r_1) = maxShift\ l \\
 & & \quad\ \ (l_2, r_2) = maxShift\ r \\
 & & \textbf{in}\ (\max\ l_1\ l_2, \max\ r_1\ r_2) \\
maxShift\ (Leaf_v\ _\ (Var\ _\ s)) & = & \textbf{if}\ s < 0\ \textbf{then}\ (-s, 0)\ \textbf{else}\ (0, s) \\
maxShift\ _ & = & (0, 0)
\end{array}
$$

We assume that the maximum amount of shifting is less than n/p.

A general function $gEval'$ to perform the local computation is defined as follows.

$$
\begin{array}{l}
gEval' :: (NForm\ \alpha \rightarrow [Int] \rightarrow \beta) \rightarrow NForm\ \alpha \rightarrow PResult\ \alpha\ \beta \\
gEval'\ fc\ [\![\, ls, zms, rs \,]\!] = \langle\!\langle\, pls, cs, prs \,\rangle\!\rangle \\
\quad \textbf{where} \\
\quad idces = [r..(n - l - 1)] \\
\quad (l, r) = maxShift\ zms \\
\quad cs = fc\ (ls, zms, rs)\ idces \\
\quad pls = \mathsf{map}\ (instP\ FromL\ zms)\ [(-l)..(r-1)] \\
\quad prs = \mathsf{map}\ (instP\ FromR\ zms)\ (reverse\ [(-r)..(l-1)])
\end{array}
$$

This $gEval'$ takes a function to calculate the center part of the partial result, and generates a partial result consisting of the result computed by the given function and two lists of computation trees with holes, which are generated by the following instantiation function $instP$.

$$
\begin{array}{l}
instP :: Direction \rightarrow Tree\ \alpha \rightarrow Int \rightarrow Tree\ \alpha \\
instP\ d\ (Node\ f\ l\ r)\ i = Node\ f\ (instP\ d\ l\ i)\ (instP\ d\ r\ i) \\
instP\ d\ (Leaf_v\ f\ (Var\ x\ s))\ i = \textbf{let}\ s' = \textbf{case}\ d\ \textbf{of}\ FromL \rightarrow s;\ FromR \rightarrow -s \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad\ \ i' = (-s' + i) \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{in if}\ (i' \leq (length\ x)) || (i' < 0)\ \textbf{then}\ Leaf_v\ f\ Hole \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else}\ Leaf_v\ f\ (Fix\ x\ i'\ d) \\
instP\ d\ x\ i = x
\end{array}
$$

This $instP$ is the same of $inst$ except that it substitutes a hole for an element that is not available in this phase.

Now, we define the function $localEval$ that performs local computation of the

normal form as follows using $gEval'$.

$$
\begin{array}{l}
localEval :: NForm\alpha \rightarrow PResult\ \alpha\ [\alpha] \\
localEval = gEval'\ fcMap \\
\quad\quad \textbf{where}\ fcMap\ [\![\, ls, zms, rs \,]\!]\ idces \\
\quad\quad\quad\quad\quad\quad = \mathsf{map}\ eval_{T0}\ ls \mathbin{+\!\!+} \mathsf{map}\ (eval_T\ zms)\ idces \mathbin{+\!\!+} \mathsf{map}\ eval_{T0}\ rs
\end{array}
$$

The defined function *fcMap* evaluates the computation tree *zms* against indices given by $gEval'$, and returns a list of the resulting elements.

Next, we will define *globalReduction*: the final global communication and completion of partial computation trees. To this end, we define the process of completion of a computation tree from two trees complementary to each other.

$$
\begin{array}{lcl}
combine :: Tree\,\alpha \rightarrow Tree\,\alpha \rightarrow Tree\,\alpha & & \\
combine\,(Node\,f_1\,l_1\,r_1)\,(Node\,f_2\,l_2\,r_2) & = & Node\,f_1\,(combine\,l_1\,l_2)\,(combine\,r_1\,r_2) \\
combine\,(Leaf_v\,f_1\,Hole)\,(Leaf_v\,f_2\,v) & = & Leaf_v\,f_2\,v \\
combine\,(Leaf_v\,f_1\,v)\,(Leaf_v\,f_2\,Hole)) & = & Leaf_v\,f_1\,v \\
combine\,(Leaf_c\,c)\,(Leaf_c\,c') & = & Leaf_c\,c
\end{array}
$$

This function completes the tree by filling holes with values held in the other tree. Using this auxiliary function, we define an operator $\boxplus_{(\cdot,\cdot)}$ for global reduction.

$$
\begin{array}{l}
(\boxplus_{(\cdot,\cdot)}) :: (\alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow (\beta \rightarrow \alpha) \rightarrow PResult\ \beta\ \alpha \rightarrow PResult\ \beta\ \alpha \rightarrow PResult\ \beta\ \alpha \\
\langle\!\langle\, pls_1, cs_1, prs_1 \,\rangle\!\rangle \boxplus_{(\oplus,f)} \langle\!\langle\, pls_2, cs_2, prs_2 \,\rangle\!\rangle = \langle\!\langle\, pls_1, cs, prs_2 \,\rangle\!\rangle \\
\quad\quad \textbf{where}\ es = \mathsf{zipwith}\ ((eval_{T0} \circ) \circ combine)\ prs_1\ pls_2 \\
\quad\quad\quad\quad\quad cs = cs_1 \oplus \mathsf{cata}\ (\oplus)\ f\ \ es \oplus cs_2
\end{array}
$$

Here, *es* is the list of elements computed from the completed trees by evaluating the trees with $eval_{T0}$. The defined operator takes an associative operator and a function to compute the reduction of the elements on breaks computed from the completed trees. This reduction is done by the following general function.

$$
\begin{array}{lcl}
\mathsf{cata} :: (\alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow (\beta \rightarrow \alpha) \rightarrow [\beta] \rightarrow \alpha & & \\
\mathsf{cata}\ (\oplus)\ f\ [\,] & = & \imath_{\oplus} \\
\mathsf{cata}\ (\oplus)\ f\ (a : x) & = & f\ a \oplus \mathsf{cata}\ (\oplus)\ f\ x
\end{array}
$$

Especially, $\mathsf{cata}\ (\mathbin{+\!\!+})\ [\cdot] = id$, and the operator $\boxplus_{(\mathbin{+\!\!+},[\cdot])}$ simply returns the resulting list.

Using the functions defined above, we define the global computation *globalReduction*.

$$
\begin{array}{lcl}
globalReduction :: [PResult\ \alpha\ [\alpha]] \rightarrow [\alpha] & & \\
globalReduction & = & extract \circ \mathsf{reduce}\ (\boxplus_{(\mathbin{+\!\!+},[\cdot])})
\end{array}
$$

The last function *extract* extracts the center value from the final partial result.

$$
\begin{array}{lcl}
extract :: PResult\ \alpha\ \beta \rightarrow \beta & & \\
extract\ \langle\!\langle\, pls, cs, prs \,\rangle\!\rangle & = & cs
\end{array}
$$

Now, we have the following completed definition of *parEval*.

$$
\begin{array}{lcl}
\mathit{parEval} :: \mathit{Int} \to \mathit{NForm}\ \alpha \to [\alpha] \\
\mathit{parEval}\ p & = & \mathit{globalReduction} \circ \mathsf{map}\ \mathit{localEval} \circ \mathit{dist}\ p
\end{array}
$$

5.2.5 Expansion of Target Programs

Based on the results on *Program*, we expand our target programs to $Program_S$, which includes accumulation by scan′ or scanr′, and $Program_R$, which includes reduction by reduce (see Section 5.2.1).

Target Programs with Accumulation

First, we expand our target programs to $Program_S$. A target skeleton program of $Program_S$ has one accumulation by scan′ or scanr′ after the computation of *Program* that involves neighbor elements using combination of $\mathsf{shift}_{\ll}$, $\mathsf{shift}_{\gg}$, zip and map. The target skeleton program also has arbitrary number of map after the accumulation.

As an example of the target programs, consider a program to solve a tridiagonal linear system of equations. The output of the program is $xs = [x_1, \ldots, x_n]$ that satisfies the following linear equations for the given coefficients $ds = [d_1, \ldots, d_n]$, $es = [e_1, \ldots, e_n]$, $fs = [f_1, \ldots, f_n]$, $bs = [b_1, \ldots, b_n]$.

$$
\underbrace{\begin{pmatrix}
d_1 & f_1 & & & & \\
e_2 & d_2 & f_2 & & & \\
 & e_3 & d_3 & f_3 & & \\
 & & \ddots & \ddots & \ddots & \\
 & & & e_{n-1} & d_{n-1} & f_{n-1} \\
 & & & & e_n & d_n
\end{pmatrix}}_{A}
\underbrace{\begin{pmatrix} x_1 \\ x_2 \\ x_2 \\ \vdots \\ x_{n-1} \\ x_n \end{pmatrix}}_{\boldsymbol{x}}
=
\underbrace{\begin{pmatrix} b_1 \\ b_2 \\ b_2 \\ \vdots \\ b_{n-1} \\ b_n \end{pmatrix}}_{\boldsymbol{b}}
$$

A skeleton program that solves this tridiagonal linear system of equations is given in Figure 5.5. This program is based on the LU decomposition of the coefficient matrix $A = LU$ [Sto73]. Here, *us* corresponds to the upper triangular matrix U, *ms* corresponds to the lower triangular matrix L, *ys* is the solution of the linear equation $Lys = bs$, and *xs* the solution of the linear equation $Axs = bs$. The operators $\otimes_1$, $\otimes_2$ and $\otimes_3$ are multiplication of 2×2 matrices, although $\otimes_2$ and $\otimes_3$ omit the half of the elements. This program consists of three parts: the first part performs the LU decomposition with scan′ to obtain *us* and *ms*, the second part performs the forward substitution by scan′ to obtain *ys*, and the third part performs the backward substitution by scanr′ to obtain *xs*.

Now, we will extend the normal form of *Program* to hold the following three: the direction, the operator, the initial value of the accumulation, and the composed

$$
\begin{array}{l}
\mathit{solveTS}\ ds\ es\ fs\ bs \\
\quad = \textbf{let}\ us = \mathsf{map}\ g_2\ (\mathsf{scan}'\ (\otimes_1)\ eye\ (\mathsf{map}\ g_1\ (\mathsf{zip}\ ds\ (\mathsf{map}\ mul\ (\mathsf{zip}\ es\ (\mathsf{shift}_{\gg}\ 0\ fs)))))) \\
\qquad\quad ms = \mathsf{map}\ div\ (\mathsf{zip}\ es\ (\mathsf{shift}_{\gg}\ \infty\ us)) \\
\qquad\quad ys = \mathsf{map}\ g_5\ (\mathsf{scan}'\ (\otimes_2)\ eye'\ (\mathsf{map}\ g_3\ (\mathsf{zip}\ ms\ bs))) \\
\qquad\quad xs = \mathsf{map}\ g_5\ (\mathsf{scanr}'\ (\otimes_3)\ eye'\ (\mathsf{map}\ g_4\ (\mathsf{zip}\ us\ (\mathsf{zip}\ ys\ fs)))) \\
\quad\ \ \textbf{in}\ xs \\
\quad\ \ \textbf{where}\ \ (a_{11}, a_{12}, a_{21}, a_{22}) \otimes_1 (b_{11}, b_{12}, b_{21}, b_{22}) \\
\qquad\qquad\qquad = (a_{11} * b_{11} + a_{12} * b_{21}, a_{11} * b_{12} + a_{12} * b_{22}, \\
\qquad\qquad\qquad\qquad a_{21} * b_{11} + a_{22} * b_{21}, a_{21} * b_{12} + a_{22} * b_{22}) \\
\qquad\qquad\quad (a_{11}, a_{21}) \otimes_2 (b_{11}, b_{21}) = (a_{11} * b_{11}, a_{21} * b_{11} + b_{21}) \\
\qquad\qquad\quad (a_{11}, a_{12}) \otimes_3 (b_{11}, b_{12}) = (a_{11} * b_{11}, a_{11} * b_{12} + a_{12}) \\
\qquad\qquad\quad g_1\ (d, ef) = (d, 1, -ef, 0) \\
\qquad\qquad\quad g_2\ (a_{11}, a_{12}, a_{21}, a_{22}) = a_{11}/a_{12} \\
\qquad\qquad\quad g_3\ (m, b) = (-m, b) \\
\qquad\qquad\quad g_4\ (u, (y, f)) = (-f/u, y/u) \\
\qquad\qquad\quad g_5\ (a_{11}, a_{12}) = a_{12} \\
\qquad\qquad\quad mul\ a\ b = a * b \\
\qquad\qquad\quad div\ a\ b = a/b \\
\qquad\qquad\quad eye = (1, 0, 0, 1) \\
\qquad\qquad\quad eye' = (1, 0)
\end{array}
$$

Figure 5.5. A skeleton program that solves the tridiagonal linear system of equations.

function applied by the last maps. The extended normal form *NFormS* is defined as follows.

$$\textbf{type}\quad NFormS\ \alpha \quad = \quad (NForm\ \beta, (Direction, \beta \to \beta \to \beta, \beta, \beta \to \alpha))$$

In the extended normal form $([\![\, ls, zms, rs \,]\!], (d, \oplus, e, f))$, $[\![\, ls, zms, rs \,]\!]$ specifies the computation before the accumulation, d is the direction of accumulation, $\oplus$ is the associative binary operator used in the accumulation, e is the initial element of the accumulation, and f is a function applied to each element after the accumulation. The direction d is *FromL* for the accumulation by scan′, and *FromR* for scanr′.

For example, the example *solveTS* is described by the following three normal forms.

$$
\begin{array}{l}
us \Rightarrow ([\![\, [g_1\ (ds, mul\ (es, 0))], g_1\ (ds, mul\ (es, fs_{\gg 1})), [\,] \,]\!], (FromL, \otimes_1, eye, g_2)) \\
ys \Rightarrow ([\![\, [g_3\ (div\ (es, \infty), bs)], g_3\ (div\ (es, us_{\gg 1}), bs), [\,] \,]\!], (FromL, \otimes_2, eye', g_5)) \\
xs \Rightarrow ([\![\, [\,], g_4\ (us, (ys, fs)), [\,] \,]\!], (FromR, \otimes_3, eye', g_5))
\end{array}
$$

Note that the computation of *ms* is absorbed by the computation of *ys*.

Next, we will extend the fusion rules to handle scan′, scanr′ and map after accumulation. The transformation of a skeleton program into an extended normal form

is performed by the following function *compileS* with fusion rules one by one.

$$\begin{aligned}
&\mathit{compileS} :: \mathit{Program}_S\ \alpha \to \mathit{NFormS}\ \alpha\\
&\mathit{compileS}\ (\underline{\mathsf{scan}}\ (\oplus)\ e\ x)\ \ = \mathit{fuseScan}\ (\oplus)\ e\ (\mathit{compile}\ x)\\
&\mathit{compileS}\ (\underline{\mathsf{scanr}}\ (\oplus)\ e\ x) = \mathit{fuseScanr}\ (\oplus)\ e\ (\mathit{compile}\ x)\\
&\mathit{compileS}\ (\underline{\mathsf{map}}\ f\ x) \qquad = \mathit{fuseMapS}\ f\ (\mathit{compileS}\ x)
\end{aligned}$$

Transformation of the computation before the accumulation is done by *compile* defined in Section 5.2.3.

Fusion rules for accumulations are as follows.

$$\begin{aligned}
&\mathit{fuseScanr} :: (\alpha \to \alpha \to \alpha) \to \alpha \to \mathit{NForm}\ \alpha \to \mathit{NFormS}\ \alpha\\
&\mathit{fuseScanr}\ (\oplus)\ e\ [\![\, \mathit{ls}, \mathit{zms}, \mathit{rs}\,]\!] = ([\![\, \mathit{ls}, \mathit{zms}, \mathit{rs}\,]\!], (\mathit{FromR}, (\oplus), e, \mathit{id}))
\end{aligned}$$

$$\begin{aligned}
&\mathit{fuseScan} :: (\alpha \to \alpha \to \alpha) \to \alpha \to \mathit{NForm}\ \alpha \to \mathit{NFormS}\ \alpha\\
&\mathit{fuseScan}\ (\oplus)\ e\ [\![\, \mathit{ls}, \mathit{zms}, \mathit{rs}\,]\!] = ([\![\, \mathit{ls}, \mathit{zms}, \mathit{rs}\,]\!], (\mathit{FromL}, (\oplus), e, \mathit{id}))
\end{aligned}$$

Each rule appends the operator $\oplus$ and the initial element e to the normal form, and marks the direction of the accumulation by *FromL* or *FromR*. The last element of the extended normal form is initialized by the identity function *id*.

The fusion rule for map after accumulation is as follows.

$$\begin{aligned}
&\mathit{fuseMapS} :: (\beta \to \alpha) \to \mathit{NFormS}\ \beta \to \mathit{NFormS}\ \alpha\\
&\mathit{fuseMapS}\ f\ \ = (\mathit{nf}, (d, (\oplus), e, g)) = (\mathit{nf}, (d, (\oplus), e, f \circ g))
\end{aligned}$$

This rule merely composes the given function f to the function g in the extended normal form.

It is obvious that any target program can be transformed into an extended normal form by the above fusion rules and the fusion rules defined in Section 5.2.3.

Finally, we will extend the parallel implementation of the normal form. Based on parallel implementation of skeleton scan′, we design parallel implementation of the normal form. Parallel implementation of skeleton scan′ is given as follows.

$$\begin{aligned}
\mathsf{scan}'\ (\oplus)\ e = {} & \mathsf{reduce}\ (+\!\!+) \circ \mathit{zipwithP}\ (\mathit{afterScan}\ (\oplus))\\
& \circ ((\mathit{prescan}\ (\oplus)\ e \circ \mathsf{map}\ \mathit{last}) \triangle \mathit{id})\\
& \circ \mathsf{map}\ (\mathsf{scan}'\ (\oplus)\ \imath_{\oplus}) \circ \mathit{dist}\ p\\
& \mathbf{where}\ \ \mathit{afterScan}\ (\oplus)\ a\ x = \mathsf{map}\ (a\oplus)\ x
\end{aligned}$$

Here, $\mathit{zipwithP}\ f\ (x, y) = \mathsf{zipwith}\ f\ x\ y$, and $(f \triangle g)\ x = (f\ x, g\ x)$. First, this implementation performs local accumulation in parallel by scan′. Then, it performs global accumulation by *prescan* defined below. Finally, *afterScan* adds the accumulated value calculated by *prescan* to each element of the result of the local accumulation.

$$\begin{aligned}
&\mathit{prescan} :: (\alpha \to \alpha \to \alpha) \to \alpha \to [\alpha] \to [\alpha]\\
&\mathit{prescan}\ (\oplus)\ e\ [\,] \qquad\quad = \ [\,]\\
&\mathit{prescan}\ (\oplus)\ e\ (a : x) \ \ = \ e : \mathit{prescan}\ (\oplus)\ (e \oplus a)\ x
\end{aligned}$$

Based on the parallel implementation of scan', we consider the following parallel implementation of a normal form with accumulation. We only show the implementation of the normal form with accumulation by scan'. The implementation for accumulation by scanr' is similar to that by scan'.

$$\begin{array}{l}
parEvalScan :: Int \to NFormS\ \alpha \to [\alpha] \\
parEvalScan\ p\ (nf, (FromL, \oplus, e, f)) \\
\quad = \ (\mathsf{reduce}\ (+\!\!+) \circ zipwithP\ (evalAfterScan\ (\oplus)\ f) \\
\qquad \circ ((prescan\ (\boxplus_{(\oplus,id)})\ ([\,], e, [\,]) \circ \mathsf{map}\ takeLast) \triangle id) \\
\qquad \circ \mathsf{map}\ (evalScan'\ (\oplus)\ \imath_{\oplus}) \circ dist\ p)\ nf
\end{array}$$

Here, $takeLast\ \langle\!\langle\, pls, cs, prs \,\rangle\!\rangle = \langle\!\langle\, pls, last\ cs, prs \,\rangle\!\rangle$. Basic structure is the same as the implementation of scan'. Main difference is that local and global computations deal with triples of partial results defined in the previous section. The local computation $evalScan'$ that generates a partial result is defined as follows. The generated partial result holds the result of local accumulation in the center of the triple.

$$\begin{array}{l}
evalScan' :: (\alpha \to \alpha \to \alpha) \to \alpha \to NForm\ \alpha \to PResult\ \alpha\ [\alpha] \\
evalScan'\ (\oplus)\ e = gEval'\ fcScan \\
\quad \textbf{where}\ fcScan\ [\![\, ls, zms, rs \,]\!]\ idces = sls +\!\!+ scs +\!\!+ srs \\
\qquad\qquad \textbf{where}\ sls = scata\ (\oplus)\ e\ eval_{T0}\ ls \\
\qquad\qquad\qquad e' = last\ (e : sls) \\
\qquad\qquad\qquad scs = scata\ (\oplus)\ e'\ (eval_T\ zms)\ idces \\
\qquad\qquad\qquad e'' = last\ (e' : scs) \\
\qquad\qquad\qquad srs = scata\ (\oplus)\ e''\ eval_{T0}\ rs
\end{array}$$

Here, $scata$ defined below performs accumulation and evaluation of the computation tree at the same time.

$$\begin{array}{lcl}
scata :: (\alpha \to \alpha \to \alpha) \to \alpha \to (\beta \to \alpha) \to [\beta] \to [\alpha] & & \\
scata\ (\oplus)\ e\ f\ [\,] & = & [\,] \\
scata\ (\oplus)\ e\ f\ (a : x) & = & \textbf{let}\ e' = (e \oplus f\ a) \\
 & & \textbf{in}\ e' : scata\ (\oplus)\ e'\ f\ x
\end{array}$$

The final local computation $evalAfterScan\ (\oplus)$ calculates accumulation of elements on the edges and adds the accumulated value to the result of the local computation.

$$\begin{array}{l}
evalAfterScan :: \ (\alpha \to \alpha \to \alpha) \to (\alpha \to \beta) \to PResult\ \alpha\ \alpha \to PResult\ \alpha\ [\alpha] \to [\beta] \\
evalAfterScan\ (\oplus)\ f\ \langle\!\langle\, epls, e, eprs \,\rangle\!\rangle\ \langle\!\langle\, pls, cs, prs \,\rangle\!\rangle = res \\
\quad \textbf{where}\ fe\ (epr, pl) = eval_{T0}\ (combine\ epr\ pl)) \\
\qquad\quad sls = scata\ (\oplus)\ e\ fe\ (\mathsf{zip}\ eprs\ pls) \\
\qquad\quad e' = last\ (e : sls) \\
\qquad\quad scs = \mathsf{map}\ (f \circ e' \oplus)\ cs \\
\qquad\quad res = \mathsf{map}\ f\ sls +\!\!+ scs
\end{array}$$

Target Programs with Reduction

Next, we expand our target programs to $Program_R$.

We will extend the normal form of *Program* to hold the operator of the reduction. The extended normal form *NFormR* is defined as follows.

$$\textbf{type} \quad NFormR\ \alpha \ = \ (NForm\ \beta, \beta \to \beta \to \beta)$$

In the extended normal form $(\llbracket\, ls, zms, rs\,\rrbracket, \oplus)$, $\llbracket\, ls, zms, rs\,\rrbracket$ specifies the computation before the accumulation, and $\oplus$ is the associative binary operator used in the reduction.

Next, we will extend the fusion rules to handle reduce. The transformation of a skeleton program into an extended normal form is performed by the following function *compileR* and fusion rules one by one.

$$\begin{aligned}&compileS :: Program_S\ \alpha \to NFormS\ \alpha\\&compileR\ (\underline{\mathsf{reduce}}\ (\oplus)\ x) = fuseReduce\ (\oplus)\ (compile\ x)\end{aligned}$$

Transformation of the computation before the reduction is done by *compile* defined in Section 5.2.3.

The fusion rule for reduction is as follows.

$$\begin{aligned}&fuseReduce :: (\alpha \to \alpha \to \alpha) \to NForm\ \alpha \to NFormR\ \alpha\\&fuseReduce\ (\oplus)\ \llbracket\, ls, zms, rs\,\rrbracket = (\llbracket\, ls, zms, rs\,\rrbracket, (\oplus))\end{aligned}$$

The rule appends the operator $\oplus$ to the normal form.

It is obvious that any target program can be transformed into an extended normal form by the above fusion rule and the fusion rules defined in Section 5.2.3.

Now, we will give parallel implementation for $Program_R$ that perform reduction by reduce to the result of a normal form. These programs can be executed in parallel by performing reduction, instead of generation of a list, in the implementation of a normal form. That is, parallel implementation is obtained by giving the reduction operator to $gEval'$ and $\boxplus_{(\cdot,\cdot)}$.

The function $evalReduce'$ that performs local reduction is defined as follows.

$$\begin{aligned}&evalReduce' :: (\alpha \to \alpha \to \alpha) \to NForm\ \alpha \to PResult\ \alpha\ \alpha\\&evalReduce'\ (\oplus) = gEval'\ fcReduce\\&\quad \textbf{where}\ fcReduce\ \llbracket\, ls, zms, rs\,\rrbracket\ idces\\&\qquad = \mathsf{cata}\ (\oplus)\ eval_{T0}\ ls \oplus \mathsf{cata}\ (\oplus)\ (eval_T\ zms)\ idces \oplus \mathsf{cata}\ (\oplus)\ eval_{T0}\ rs\end{aligned}$$

This function performs reduction and evaluation of computation trees at the same time by $\mathsf{cata}\ (\oplus)\ eval_{T0}$ and $\mathsf{cata}\ (\oplus)\ (eval_T\ zms)$. Using this function, parallel implementation of a normal form with reduction is defined as follows.

$$\begin{aligned}&parEvalReduce :: Int \to NFormR\ \alpha \to \alpha\\&parEvalReduce\ p\ (nf, \oplus)\\&\qquad = (extract \circ \mathsf{reduce}\ (\boxplus_{(\oplus,id)}) \circ \mathsf{map}\ (evalReduce'\ (\oplus)) \circ dist\ p)\ nf\end{aligned}$$

5.3 Nested Reductions: Involving an Infinite Number of Neighbor Elements

In this section, we will focus on computations of nested reductions: nested data structures are generated from flat structures, and they are consumed by two-level nested reductions. This is a generalized version of computation discussed in the previous section, in the sense that the computation involving a finite number of neighbor elements can be specified as nested reductions.

First, we will introduce the general form of nested reductions. Then, we will introduce some useful functions to generate nested structures, and show some example concrete problems. Finally, we will develop shortcuts of fusion optimizations, i.e., theorems to dispatch efficient implementations to general forms that satisfy some specific conditions.

5.3.1 Generate-and-test Specifications: General Forms of Nested Reductions

We will introduce general forms "generate-and-test specifications" of nested reductions on lists and two-dimensional arrays.

First, we will introduce the general form for nested reductions on lists.

Definition 5.2 (Generate-and-test specification for lists)**.** The general form for nested reductions on lists is given as follows.

$$(\![f, \oplus]\!) \circ \mathsf{map}\,(\![g, \otimes]\!) \circ \mathsf{filter}\,p \circ gg$$

Here, gg is a function to generate nested lists. The generated lists are filtered with predicate p. Then, the filtered lists are consumed by the nested reductions: $(\![f, \oplus]\!)$ for the outer list and $(\![g, \otimes]\!)$ for the inner lists. □

Note that homomorphism $(\![f, \oplus]\!)$ is equivalent to the composition $\mathsf{reduce}\,(\oplus) \circ \mathsf{map}\,f$. Especially, simple nested reductions with two binary operators $\oplus$ and $\otimes$ for the generation by gg can be described in the generate-and-test specification as follows.

$$\mathsf{reduce}\,(\oplus) \circ \mathsf{map}\,(\mathsf{reduce}\,(\otimes)) \circ gg = (\![id, \oplus]\!) \circ \mathsf{map}\,(\![id, \otimes]\!) \circ \mathsf{filter}\,true \circ gg$$

Here, $true$ is the predicate returning always True. The equality of the left hand side (the simple nested reductions) and the right hand side (the generate-and-test specification) is easily shown. First, by the definition of $\mathsf{reduce}\,(\oplus)$, we have $(\![id, \oplus]\!) = \mathsf{reduce}\,(\oplus)$ and $(\![id, \otimes]\!) = \mathsf{reduce}\,(\otimes)$. Next, by the definition of $\mathsf{filter}\,p$, $\mathsf{filter}\,true = id$ since $true$ returns always True.

It is worth noting that the identity function id can be described with homomorphism, i.e., $id = (\![[\cdot], +\!\!+]\!)$, and the outer reduction may be omitted when it is the identity function.

Next, we will introduce the general form for two-dimensional arrays.

Definition 5.3 (Generate-and-test specification for two-dimensional arrays)**.** The general form for nested reductions on two-dimensional arrays is given as follows.

$$([f, \oplus, \otimes]) \circ \mathsf{map}\ ([g, \ominus, \odot]) \circ gg$$

Here, gg is a function to generate nested arrays. The generated arrays are consumed by the nested reductions: $([f, \oplus, \otimes])$ for the outer structure, and $([g, \ominus, \odot])$ for the inner structures. □

Note that homomorphism $([f, \oplus, \otimes])$ is equivalent to the composition of skeletons: $\mathsf{reduce}\ (\oplus, \otimes) \circ \mathsf{map}\ f$. We do not deal with filtering for two-dimensional arrays.

Here is an extension of the specifications. We will use a product of k homomorphisms, i.e., $([g_1, \ominus_1]) \times \cdots \times ([g_k, \ominus_k])$ and $([g_1, \ominus_1, \odot_1]) \times \cdots \times ([g_k, \ominus_k, \odot_k])$ instead of the simple homomorphism $([g, \ominus])$ and $([g, \ominus, \odot])$, when generation functions generate nested structures in which generated inner structures are tupled.

5.3.2 Functions to Generate Nested Data Structures

This section gives a collection of useful functions to generate nested structures from the given flat structure, which will be used as the first function gg of the general form. For each function, we will give the formal definition and example applications written with the function.

Now, we will introduce some predicates used in the examples applications.

Predicate *ascending* returns True if the given list is ascending (i.e., each element is smaller than the following element). For example, applying *ascending* to a sorted list $[-2, 4, 5]$ we get $ascending\ [-2, 4, 5] = \mathsf{True}$. Conversely, $ascending\ [3, 6, 2] = \mathsf{False}$ since 6 is not smaller than 2.

Predicate *descending* returns True if the given list is descending (i.e., each element is bigger than the following element). For example, applying *descending* to a descendingly sorted list $[5, 3, 0]$ we get $descending\ [5, 3, 0] = \mathsf{True}$. Conversely, $ascending\ [1, 9, 7] = \mathsf{False}$ since 1 is not bigger than 9.

Predicate $smooth_c$ returns True if a difference between any successive elements in the given list is less than or equal to c. For example, $smooth_2\ [6, 5, 7] = \mathsf{True}$, since both of the differences $1 = |6 - 5|$ and $2 = |5 - 7|$ are less than or equal to 2. Conversely, $smooth_2\ [6, 5, 8] = \mathsf{False}$ because the difference $|5 - 8| = 3$ is greater than 2.

Predicate *high* returns True if the maximum element of the given list is greater than the length of the list. For example, $high\ [1, 4, 3] = \mathsf{True}$, since the maximum element 4 is greater than the length of the list, i.e., 3. Conversely, $high\ [1, 4, 3, 2] = \mathsf{False}$ because the maximum element 4 is not greater than the length of the list 4.

Formal definitions of the above predicates will be shown later.

Function inits to Generate Prefix Segments

Function inits generates all prefixes of an input list in the lexicographic order. For example, applying inits to $[1, 3, 1, -7, 2, 4]$, we get the following nested list.

$$\mathsf{inits}\ [1, 3, 1, -7, 2, 4] = [[1], [1, 3], [1, 3, 1], [1, 3, 1, -7], [1, 3, 1, -7, 2], [1, 3, 1, -7, 2, 4]]$$

Here, each element of the resulting list is a prefix of the given list.

The formal definition of inits is given as follows.

Definition 5.4 (Inits)**.** Function inits is defined with homomorphism as follows.

$$\mathsf{inits} = ([[\cdot] \circ [\cdot], \oplus]) \quad \textbf{where}\ \ x \oplus y = x +\!\!+ \mathsf{map}\,((last\ x) +\!\!+)\, y$$

□

The operator $\oplus$ in the above definition makes a list of initial segments of a list $u +\!\!+ v$ from lists (x and y in the above equation) of initial segments of u and v. Since each initial segment of u is also an initial segment of $u +\!\!+ v$, x remains in the result. Since each initial segment of v need to be concatenated with u to become an initial segment of $u +\!\!+ v$, the operator maps $((last\ x) +\!\!+)$ (u is the last element of x) to y.

The most famous application of inits is prefix sums, which has many applications [Ble90]. .

Example 5.5 (Prefix sums)**.** The statement is as follows: Given a list and an associative binary operator, find sums for all prefixes of the given list.

For example, prefix sums of $[1, 3, 1, -7, 2, 4]$ with operator $\oplus$ is $[1, 1 \oplus 3, 1 \oplus 3 \oplus 1, 1 \oplus 3 \oplus 1 \oplus -7, 1 \oplus 3 \oplus 1 \oplus -7 \oplus 2, 1 \oplus 3 \oplus 1 \oplus -7 \oplus 2 \oplus 4]$. Using inits, prefix sums are easily obtained by applying reduction with $\oplus$ to all prefixes generated by inits.

$$\begin{aligned}
&\mathsf{map}\,(\mathsf{reduce}\,(\oplus))\,(\mathsf{inits}\ [1, 3, 1, -7, 2, 4]) \\
&= \mathsf{map}\,(\mathsf{reduce}\,(\oplus))\,[[1], [1, 3], [1, 3, 1], [1, 3, 1, -7], [1, 3, 1, -7, 2], [1, 3, 1, -7, 2, 4]] \\
&= [1, 1 \oplus 3, 1 \oplus 3 \oplus 1, 1 \oplus 3 \oplus 1 \oplus -7, 1 \oplus 3 \oplus 1 \oplus -7 \oplus 2, 1 \oplus 3 \oplus 1 \oplus -7 \oplus 2 \oplus 4]
\end{aligned}$$

Note that the outer reduction may be the identity function $id = ([[\cdot], +\!\!+])$.

The maximum prefix (initial-segment) sum problem is one of optimization problems on sequences.

Example 5.6 (Maximum prefix sum)**.** Its statement is as follows: Given a list, find the maximum sum of a prefix of the given list.

For example, the maximum prefix sum of $[1, 3, 1, -7, 2, 4]$ is 5 since its prefix sums (with the usual plus operator $+$) are $[1, 4, 5, -2, 0, 4]$. So, using inits we can get the

maximum prefix sum easily by applying reduction with the maximum operator $\uparrow$ for the prefix sums.

$$
\begin{aligned}
&\mathsf{reduce}\,(\uparrow)\,(\mathsf{map}\,(\mathsf{reduce}\,(+))\,(\mathsf{inits}\ [1,3,1,-7,2,4]))\\
&= \mathsf{reduce}\,(\uparrow)\,(\mathsf{map}\,(\mathsf{reduce}\,(+))\,[[1],[1,3],[1,3,1],[1,3,1,-7],[1,3,1,-7,2],\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad [1,3,1,-7,2,4]])\\
&= \mathsf{reduce}\,(\uparrow)\,([1,1+3,1+3+1,1+3+1+-7,1+3+1+-7+2,\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad 1+3+1+-7+2+4])\\
&= 5
\end{aligned}
$$

Here is a variant of the maximum prefix sum problem.

Example 5.7 (Maximum p-prefix sum)**.** Its statement is as follows: Given a list and a predicate p, find the maximum sum of its prefix satisfying the given predicate p.

For example, the maximum ascending-prefix sum of $[1,3,1,-7,2,4]$ is 4 since its ascending prefixes are $[1]$ and $[1,3]$. Using inits and filter, we can get the maximum ascending-prefix sum easily as follows.

$$
\begin{aligned}
&\mathsf{reduce}\,(\uparrow)\,(\mathsf{map}\,(\mathsf{reduce}\,(+))\,(\mathsf{filter}\ \mathit{ascending}\,(\mathsf{inits}\ [1,3,1,-7,2,4])))\\
&= \mathsf{reduce}\,(\uparrow)\,(\mathsf{map}\,(\mathsf{reduce}\,(+))\,(\mathsf{filter}\ \mathit{ascending}\,[[1],[1,3],[1,3,1],[1,3,1,-7],\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad [1,3,1,-7,2],[1,3,1,-7,2,4]]))\\
&= \mathsf{reduce}\,(\uparrow)\,(\mathsf{map}\,(\mathsf{reduce}\,(+))\,[[1],[1,3]])\\
&= \mathsf{reduce}\,(\uparrow)\ \ [1,1+3]\\
&= 4
\end{aligned}
$$

All three examples above are instances of the general form. Here is the summary of the example programs.

$$
\begin{array}{lcl}
\text{prefix sums} & : & (\![[\cdot], +\!\!+]\!) \circ \mathsf{map}\,(\mathsf{reduce}\,(\oplus)) \circ \mathsf{inits}\\
\text{maximum prefix sum} & : & \mathsf{reduce}\,(\uparrow) \circ \mathsf{map}\,(\mathsf{reduce}\,(+)) \circ \mathsf{inits}\\
\text{maximum } p\text{-prefix sum} & : & \mathsf{reduce}\,(\uparrow) \circ \mathsf{map}\,(\mathsf{reduce}\,(+)) \circ \mathsf{filter}\,p \circ \mathsf{inits}
\end{array}
$$

Function tails to Generate Suffix Segments

Function tails generates all suffixes of an input list in the lexicographic order[1]. For example, applying tails to $[1,3,1,-7,2,4]$, we get the following nested list.

$$\mathsf{tails}\ [1,3,1,-7,2,4] = [[1,3,1,-7,2,4],[3,1,-7,2,4],[1,-7,2,4],[-7,2,4],[2,4],[4]]$$

Here, each element of the resulting list is a suffix of the given list.

The formal definition of tails is given as follows.

[1]Note that the lexicographic order is that with respect to positions of elements in the input. The following example make this clear: $\mathsf{tails}\ [a_1,a_2,a_3] = [[a_1,a_2,a_3],[a_2,a_3],[a_3]]$. Here, concatenations of the indices of the suffixes are 123, 23, and 3, which is arranged in the lexicographic order.

Definition 5.8 (Tails)**.** Function tails is defined with homomorphism as follows.

$$\mathsf{tails} = (\![[\cdot] \circ [\cdot], \oplus]\!) \quad \textbf{where} \;\; x \oplus y = \mathsf{map}\,(+\!\!+(head\;y))\,x +\!\!+ y$$

□

The operator $\oplus$ in the above definition makes a list of tail segments of a list $u +\!\!+ v$ from lists (x and y in the above equation) of tail segments of u and v. Since each tail segment of v is also a tail segment of $u +\!\!+ v$, y remains in the result. Since each tail segment of u need to be concatenated with v to become a tail segment of $u +\!\!+ v$, the operator maps $(+\!\!+(head\;y))$ (v is the head element of y) to x.

The most famous application of tails is suffix sums, which is converse computation of prefix sums shown in the previous section.

Example 5.9 (Suffix sums)**.** The statement is as follows: Given a list and an associative binary operator, find sums for all suffixes of the given list.

For example, suffix sums of $[1, 3, 1, -7, 2, 4]$ with operator $\oplus$ is $[1 \oplus 3 \oplus 1 \oplus -7 \oplus 2 \oplus 4, 3 \oplus 1 \oplus -7 \oplus 2 \oplus 4, 1 \oplus -7 \oplus 2 \oplus 4, -7 \oplus 2 \oplus 4, 2 \oplus 4, 4]$. Using tails, we can get suffix sums easily by applying reduction with $\oplus$ to all suffixes generated by tails.

$$\begin{aligned}
&\mathsf{map}\,(\mathsf{reduce}\,(\oplus))\,(\mathsf{tails}\;[1, 3, 1, -7, 2, 4])\\
&= \mathsf{map}\,(\mathsf{reduce}\,(\oplus))\,[[1, 3, 1, -7, 2, 4], [3, 1, -7, 2, 4], [1, -7, 2, 4], [-7, 2, 4], [2, 4], [4]]\\
&= [1 \oplus 3 \oplus 1 \oplus -7 \oplus 2 \oplus 4, 3 \oplus 1 \oplus -7 \oplus 2 \oplus 4, 1 \oplus -7 \oplus 2 \oplus 4, -7 \oplus 2 \oplus 4, 2 \oplus 4, 4]
\end{aligned}$$

The maximum suffix sum problem is one of optimization problems on sequences.

Example 5.10 (Maximum suffix sum)**.** Its statement is as follows: Given a list, find the maximum sum of a suffix of the given list.

For example, the maximum suffix sum of $[1, 3, 1, -7, 2, 4]$ is 6 since its suffix sums (with the usual plus operator $+$) are $[4, 3, 0, -1, 6, 4]$. So, using tails we can get the maximum suffix sum easily by applying reduction with the maximum operator $\uparrow$ for suffix sums.

$$\begin{aligned}
&\mathsf{reduce}\,(\uparrow)\,(\mathsf{map}\,(\mathsf{reduce}\,(+))\,(\mathsf{tails}\;[1, 3, 1, -7, 2, 4]))\\
&= \mathsf{reduce}\,(\uparrow)\,(\mathsf{map}\,(\mathsf{reduce}\,(+))\,[[1, 3, 1, -7, 2, 4], [3, 1, -7, 2, 4], [1, -7, 2, 4],\\
&\qquad\qquad [-7, 2, 4], [2, 4], [4]]\\
&= \mathsf{reduce}\,(\uparrow)\,[1 + 3 + 1 + -7 + 2 + 4, 3 + 1 + -7 + 2 + 4, 1 + -7 + 2 + 4,\\
&\qquad\qquad -7 + 2 + 4, 2 + 4, 4]\\
&= 6
\end{aligned}$$

Here is a variant of the maximum suffix sum problem.

Example 5.11 (Maximum p-suffix sum)**.** Its statement is as follows: Given a list and a predicate p, find the maximum sum of its suffix satisfying the given predicate p.

For example, the maximum ascending-suffix sum of $[1,3,1,-7,2,4]$ is 6 since its ascending suffixes are $[-7,2,4]$, $[2,4]$ and $[4]$. Using tails and filter, we can get the maximum ascending-suffix sum easily as follows.

$$\begin{aligned}
&\mathsf{reduce}\,(\uparrow)\,(\mathsf{map}\,(\mathsf{reduce}\,(+))\,(\mathsf{filter}\,\mathit{ascending}\,(\mathsf{tails}\ [1,3,1,-7,2,4])))\\
&= \mathsf{reduce}\,(\uparrow)\,(\mathsf{map}\,(\mathsf{reduce}\,(+))\,(\mathsf{filter}\,\mathit{ascending}\,[[1,3,1,-7,2,4],[3,1,-7,2,4],\\
&\qquad [1,-7,2,4],[-7,2,4],[2,4],[4]]))\\
&= \mathsf{reduce}\,(\uparrow)\,(\mathsf{map}\,(\mathsf{reduce}\,(+))\,[[-7,2,4][2,4],[4]])\\
&= \mathsf{reduce}\,(\uparrow)\,[-7+2+4,2+4,4]\\
&= 6
\end{aligned}$$

Function segs to Generate All Segments

Function segs generates segments (continuous subsequences) of an input list in the lexicographic order. For example, applying segs to $[3,2,-7,4,2]$ we get the following nested list.

$$\begin{aligned}
\mathsf{segs}\ [3,2,-7,4,2] \quad = \quad & [[3],[3,2],[3,2,-7],[3,2,-7,4],[3,2,-7,4,2],[2],[2,-7],\\
& [2,-7,4],[2,-7,4,2],[-7],[-7,4],[-7,4,2],[4],[4,2],[2]]
\end{aligned}$$

The formal definition of segs is given as follows.

Definition 5.12 (Segs)**.** Function segs is defined with homomorphism as follows.

$$\mathsf{segs} = \mathsf{reduce}\,(+\!\!+) \circ \mathsf{map\,inits} \circ \mathsf{tails}$$

□

This definition is based on the fact that a segment of a list is a prefix of a suffix of the list.

The most famous application of segs is maximum segment sum problem [Bir87, Jeu93, SHTO00], which is one of optimization problems on sequences.

Example 5.13 (Maximum segment sum)**.** The statement is as follows: Given a list, find the maximum sum of a segment of the given list.

For example, the maximum segment sum of $[3,2,-7,4,2]$ is 6 that is the sum of its segment $[4,2]$. We can get the maximum segment sum of the given list using segs as follows. First, we generate all segments of the given list by segs. Then, we apply reduction with the usual plus operator $+$ to generated segments to get all segment sums. Finally, applying reduction with maximum-operator to those segment sums, we get the maximum segment sum of the given list.

$$\begin{aligned}
&\mathsf{reduce}\,(\uparrow)\,(\mathsf{map}\,(\mathsf{reduce}\,(+))\,(\mathsf{segs}\ [3,2,-7,4,2]))\\
&= \mathsf{reduce}\,(\uparrow)\,(\mathsf{map}\,(\mathsf{reduce}\,(+))\,[[3],[3,2],[3,2,-7],[3,2,-7,4],[3,2,-7,4,2],\\
&\qquad [2],[2,-7],[2,-7,4],[2,-7,4,2],[-7],[-7,4],\\
&\qquad [-7,4,2],[4],[4,2],[2]])\\
&= \mathsf{reduce}\,(\uparrow)\,[3,5,-2,2,4,2,-5,-1,1,-7,-3,-1,4,6,2]\\
&= 6
\end{aligned}$$

Here is a variant of the maximum segment sum problem.

Example 5.14 (Maximum p-segment sum)**.** Its statement is as follows: Given a list and a predicate p, find the maximum sum of its segment satisfying the given predicate p.

For example, the maximum ascending-segment sum of $[3, 2, -7, 4, 2]$ is 4 since its ascending-segments are all singletons and $[-7, 4]$. The maximum descending-segment sum of $[3, 2, -7, 4, 2]$ is 6 since its descending-segments are $[3, 2]$, $[3, 2, -7]$, $[2, -7]$, $[4, 2]$ and all singletons. Using segs and filter, we can get the maximum descending-segment sum easily as follows.

$$\begin{aligned}
&\mathsf{reduce}\,(\uparrow)\,(\mathsf{map}\,(\mathsf{reduce}\,(+))\,(\mathsf{filter}\;descending\,(\mathsf{segs}\;[3,2,-7,4,2])))\\
&= \mathsf{reduce}\,(\uparrow)\,(\mathsf{map}\,(\mathsf{reduce}\,(+))\,(\\
&\qquad \mathsf{filter}\;descending\,[[3],[3,2],[3,2,-7],[3,2,-7,4],[3,2,-7,4,2],\\
&\qquad\qquad [2],[2,-7],[2,-7,4],[2,-7,4,2],[-7],[-7,4],\\
&\qquad\qquad\qquad [-7,4,2],[4],[4,2],[2]]))\\
&= \mathsf{reduce}\,(\uparrow)\,(\mathsf{map}\,(\mathsf{reduce}\,(+))\,[[3],[3,2],[3,2,-7],[2],[2,-7],[-7],[4],[4,2],[2]])\\
&= \mathsf{reduce}\,(\uparrow)\,[3,5,-2,2,-5,-7,4,6,2]\\
&= 6
\end{aligned}$$

Longest-p segment problem [Zan92] is also one of optimization problems on sequences.

Example 5.15 (Longest-p segment)**.** Its statement is as follows: Given a list and a predicate, find the longest segment (continuous subsequence) of the list that satisfies the predicate.

For example, the longest-ascending segment of $[3, 2, -7, 4, 2]$ is $[-7, 4]$, while the longest-descending segment is $[3, 2, -7]$. There are many instances of longest-p segment for various predicates. The longest segment satisfying a given predicate p is obtained by using segs and filtering. Some instances are shown below.

The longest smooth segment of an input list is obtained by using the predicate $smooth_c$.

$$\begin{aligned}
&\mathsf{reduce}\,(\uparrow_{length})\,(\mathsf{filter}\;smooth_4\,(\mathsf{segs}\;[3,2,-7,4,2]))\\
&= \mathsf{reduce}\,(\uparrow_{length})\,(\mathsf{filter}\;smooth_4\,(\mathsf{segs}\;[[3],[3,2],[3,2,-7],[3,2,-7,4],[3,2,-7,4,2],\\
&\qquad\qquad [2],[2,-7],[2,-7,4],[2,-7,4,2],[-7],[-7,4],\\
&\qquad\qquad\qquad [-7,4,2],[4],[4,2],[2]]))\\
&= \mathsf{reduce}\,(\uparrow_{length})\;\;[[3],[3,2],[2],[-7],[4],[4,2],[2]]\\
&= [3,2]
\end{aligned}$$

The longest high segment of an input list is obtained by using predicate *high*.

$$
\begin{aligned}
&\mathsf{reduce}\,(\uparrow_{length})\,(\mathsf{filter}\; high\;(\mathsf{segs}\;[3,2,-7,4,2]))\\
&= \mathsf{reduce}\,(\uparrow_{length})\,(\mathsf{filter}\; high\;(\\
&\qquad\qquad \mathsf{segs}\;[[3],[3,2],[3,2,-7],[3,2,-7,4],[3,2,-7,4,2],\\
&\qquad\qquad\qquad [2],[2,-7],[2,-7,4],[2,-7,4,2],\\
&\qquad\qquad\qquad [-7],[-7,4],[-7,4,2],[4],[4,2],[2]]))\\
&= \mathsf{reduce}\,(\uparrow_{length})\;\;[[3],[3,2],[2],[4],[4,2],[2]]\\
&= [3,2]
\end{aligned}
$$

Function *neighbors* to Generate Neighbor Segments

Function *neighbors* generates left and right neighbor segments for each element of an input list. For example, applying *neighbors* to $[1,3,1,-7,2,4]$, we get the following list of tuples, in which each tuple consists of the left neighbor segment, the center element, and the right neighbor segment.

$$
\begin{aligned}
neighbors\;[1,3,1,-7,2,4] = [&([\,],1,[3,1,-7,2,4]),([1],3,[1,-7,2,4]),\\
&([1,3],1,[-7,2,4]),([1,3,1],-7,[2,4]),\\
&([1,3,1,-7],2,[4]),([1,3,1,-7,2],4,[\,])]
\end{aligned}
$$

The formal definition of *neighbors* is given as follows.

Definition 5.16 (Neighbors)**.** Function *neighbors* is defined using inits and tails as follows.

$$
\begin{aligned}
neighbors \;\; = \;\; & zipP_3 \circ ((\mathsf{map\;init} \circ \mathsf{inits}) \triangle id \triangle (\mathsf{map\;tail} \circ \mathsf{tails}))\\
& \textbf{where}\;\; zipP_3\;(x,y,z) = \mathsf{zip}\;x\;y\;z
\end{aligned}
$$

□

Basically, *neighbors* is a composition of inits and tails. It uses extra init and tail to remove redundant occurrence of the center element.

Here is a variant of *neighbors* to generate neighbor segments of fixed sizes. This variant is useful for describing examples shown in the previous section (Section 5.2). For example, applying the variant $neighbors'$ to $[1,3,1,-7,2,4]$ with boundaries $[5,6]$ and $[9]$, we get the following list of tuples. Here, each left neighbor segment has two elements, and each right neighbor segment has only one element, which are the same as the given boundaries.

$$
\begin{aligned}
neighbors'\;[5,6]\;\;[9]\;\;[1,3,1,-7,2,4] = [&([5,6],1,[3]),([6,1],3,[1]),([1,3],1,[-7]),\\
&([3,1],-7,[2]),([1,-7],2,[4]),([-7,2],4,[8])]
\end{aligned}
$$

The formal definition of $neighbors'$ is given as follows.

Definition 5.17 (Finite-window neighbors)**.** Function *neighbors′* takes two lists for boundary elements: *ls* for the left edge, and *rs* for the right edge.

$$\begin{array}{lll} \mathit{neighbors}'\ \mathit{ls}\ \mathit{rs} & = & \mathsf{map}\ (\mathit{taker}\ m \circ (\mathit{ls}+\!\!+) \times \mathit{id} \times \mathit{take}\ n \circ (+\!\!+\mathit{rs})) \circ \mathit{neighbors} \\ & & \mathbf{where}\ \ m = \mathit{length}\ \mathit{ls} \\ & & \qquad\quad\ n = \mathit{length}\ \mathit{rs} \end{array}$$

□

Example 5.18 (Running example in Section 5.2)**.** The running example in Section 5.2 is described with *neighbors* as follows.

$$\begin{array}{lll} \mathit{next} & = & \mathsf{map}\ f \circ \mathit{neighbors}'\ [b_{l_0}, b_{l_1}]\ [b_r] \\ & & \mathbf{where}\ \ f\ ([u_{-2}, u_{-1}], u_0, [u_1]) = c_{-2} \times u_{-2} + c_{-1} \times u_{-1} + c_0 \times u_0 + c_1 \times u_1 \end{array}$$

Function subs to Generate All Subsequences

Function subs generates all subsequences (sub-lists) of an input list. Generated subsequences are listed in the lexicographic order. Its definition is as follows.

$$\begin{array}{lll} \mathsf{subs}\ [a] & = & [[a]] \\ \mathsf{subs}\ (a : x) & = & \mathbf{let}\ sx = \mathsf{subs}\ x\ \mathbf{in}\ [a] +\!\!+ \mathsf{map}\ ([a]+\!\!+)\ sx +\!\!+ sx \end{array}$$

For example, applying subs to $[2, 5, -3, 4]$ we get the following nested list.

$$\begin{array}{l} \mathsf{subs}\ [2, 3, -3, 4] = [[2], [2, 3], [2, 3, -3], [2, 3, -3, 4], [2, 3, 4], [2, -3], [2, -3, 4], [2, 4], \\ \qquad\qquad\qquad\qquad\qquad [3], [3, -3], [3, -3, 4], [3, 4], [-3], [-3, 4], [4]] \end{array}$$

Longest-p subsequence problem is one of optimization problems on sequences [Jeu93].

Example 5.19 (Longest-p subsequence)**.** Its statement is as follows: Given a list and a predicate, find the longest subsequence of the list that satisfies the predicate.

For example, the longest-ascending subsequence of $[2, 3, -3, 4]$ is $[2, 3, 4]$, while the longest-descending subsequence is $[2, -3]$. There are many instances of longest-p subsequence for various predicates. The longest subsequence satisfying a given predicate p is obtained by using subs and filter. Some instances are shown blow.

The longest ascending subsequence of an input list is obtained by using subs and the predicate *ascending*.

$$\begin{array}{l} \mathsf{reduce}\ (\uparrow_{\mathit{length}})\ (\mathsf{filter}\ \mathit{ascending}\ (\mathsf{subs}\ [2, 3, -3, 4])) \\ \mathsf{reduce}\ (\uparrow_{\mathit{length}})\ (\mathsf{filter}\ \mathit{ascending} \\ \qquad\qquad [[2], [2, 3], [2, 3, -3], [2, 3, -3, 4], [2, 3, 4], [2, -3], [2, -3, 4], [2, 4], \\ \qquad\qquad\qquad [3], [3, -3], [3, -3, 4], [3, 4], [-3], [-3, 4], [4]]) \\ = \mathsf{reduce}\ (\uparrow_{\mathit{length}})\ [[2], [2, 3], [2, 3, 4], [2, 4], [3], [3, 4], [-3], [-3, 4], [4]] \\ = [2, 3, 4] \end{array}$$

The longest descending subsequence of an input list is obtained by using subs and the predicate *descending*.

$$
\begin{array}{l}
\mathsf{reduce}\,(\uparrow_{length})\,(\mathsf{filter}\ descending\,(\mathsf{subs}\ [2,3,-3,4]))\\
\mathsf{reduce}\,(\uparrow_{length})\,(\mathsf{filter}\ descending\\
\qquad\qquad [[2],[2,3],[2,3,-3],[2,3,-3,4],[2,3,4],[2,-3],[2,-3,4],\\
\qquad\qquad\quad [2,4],[3],[3,-3],[3,-3,4],[3,4],[-3],[-3,4],[4]])\\
= \mathsf{reduce}\,(\uparrow_{length})\,[[2],[2,-3],[3],[3,-3],[-3],[4]]\\
= [2,-3]
\end{array}
$$

Here is another example problem.

Example 5.20 (0-1 knapsack problem)**.** Given a list of items (pairs of value and weight) and a knapsack of fixed capacity, find a subset of items that has the maximum sum of values and its sum of weight is less than or equals to the capacity.

For example, given four items $[(2,1),(5,3),(1,1),(4,2)]$ (first element of a pair is value and the second is weight) and a knapsack of capacity 4, the solution (a set of items to be put into the knapsack) is $[(2,1),(5,3)]$. This solution is given as follows. First, we generate every combination of items by subs. Then, using filtering, we throw away combinations of which weight is greater than the capacity. Finally, we take the combination that have the maximum sum of values.

$$
\begin{array}{l}
\mathsf{reduce}\,(\uparrow_{\mathsf{reduce}\,(+)\,\circ\,\mathsf{map}\,\pi_1})\,(\\
\qquad \mathsf{filter}\,((\leq 4)\circ(\mathsf{reduce}\,(+))\circ(\mathsf{map}\,\pi_2))\,(\mathsf{subs}\ [(2,1),(5,3),(1,1),(4,2)]))\\
= \mathsf{reduce}\,(\uparrow_{\mathsf{reduce}\,(+)\,\circ\,\mathsf{map}\,\pi_1})\,[[(2,1)],[(2,1),(5,3)],[(2,1),(1,1)],[(2,1),(1,1),(4,2)],\\
\qquad [(2,1),(4,2)],[(5,3)],[(5,3),(1,1)],[(1,1)],[(1,1),(4,2)],[(4,2)]]\\
= [(2,1),(5,3)]
\end{array}
$$

Functions Ts, Ls, Bs, Rs, and Their Combinations to Generate Rectangles

We will introduce eight functions to generate various rectangles. Basically, these functions are extensions of the functions inits and tails on lists.

$$
\begin{array}{lcl}
\mathsf{Ts} & = & \mathsf{scan}\ (\ominus,\gg)\circ\mathsf{map}\ |\cdot|\\
\mathsf{Ls} & = & \mathsf{scan}\ (\gg,\phi)\circ\mathsf{map}\ |\cdot|\\
\mathsf{Bs} & = & \mathsf{scanr}\ (\ominus,\ll)\circ\mathsf{map}\ |\cdot|\\
\mathsf{Rs} & = & \mathsf{scanr}\ (\ll,\phi)\circ\mathsf{map}\ |\cdot|\\
\mathsf{TLs} & = & \mathsf{scan}\ (\ominus,\phi)\circ\mathsf{map}\ |\cdot|\\
\mathsf{TRs} & = & \mathsf{scanr}\ (\ll,\phi)\circ\mathsf{scan}\ (\ominus,\gg)\circ\mathsf{map}\ |\cdot|\\
\mathsf{BLs} & = & \mathsf{scanr}\ (\ominus,\ll)\circ\mathsf{scan}\ (\gg,\phi)\circ\mathsf{map}\ |\cdot|\\
\mathsf{BRs} & = & \mathsf{scanr}\ (\ominus,\phi)\circ\mathsf{map}\ |\cdot|
\end{array}
$$

Example use of those functions are shown below.

$$\mathsf{Ts}\begin{pmatrix}2&7&3\\8&4&1\end{pmatrix} = \begin{pmatrix}\begin{pmatrix}2\end{pmatrix} & \begin{pmatrix}7\end{pmatrix} & \begin{pmatrix}3\end{pmatrix}\\ \begin{pmatrix}2\\8\end{pmatrix} & \begin{pmatrix}7\\4\end{pmatrix} & \begin{pmatrix}3\\1\end{pmatrix}\end{pmatrix}$$

$$\mathsf{Ls}\begin{pmatrix}2&7&3\\8&4&1\end{pmatrix} = \begin{pmatrix}\begin{pmatrix}2\end{pmatrix} & \begin{pmatrix}2&7\end{pmatrix} & \begin{pmatrix}2&7&3\end{pmatrix}\\ \begin{pmatrix}8\end{pmatrix} & \begin{pmatrix}8&4\end{pmatrix} & \begin{pmatrix}8&4&1\end{pmatrix}\end{pmatrix}$$

$$\mathsf{Bs}\begin{pmatrix}2&7&3\\8&4&1\end{pmatrix} = \begin{pmatrix}\begin{pmatrix}2\\8\end{pmatrix} & \begin{pmatrix}7\\4\end{pmatrix} & \begin{pmatrix}3\\1\end{pmatrix}\\ \begin{pmatrix}8\end{pmatrix} & \begin{pmatrix}4\end{pmatrix} & \begin{pmatrix}1\end{pmatrix}\end{pmatrix}$$

$$\mathsf{Rs}\begin{pmatrix}2&7&3\\8&4&1\end{pmatrix} = \begin{pmatrix}\begin{pmatrix}2&7&3\end{pmatrix} & \begin{pmatrix}7&3\end{pmatrix} & \begin{pmatrix}3\end{pmatrix}\\ \begin{pmatrix}8&4&1\end{pmatrix} & \begin{pmatrix}4&1\end{pmatrix} & \begin{pmatrix}1\end{pmatrix}\end{pmatrix}$$

$$\mathsf{TLs}\begin{pmatrix}2&7&3\\8&4&1\end{pmatrix} = \begin{pmatrix}\begin{pmatrix}2\end{pmatrix} & \begin{pmatrix}2&7\end{pmatrix} & \begin{pmatrix}2&7&3\end{pmatrix}\\ \begin{pmatrix}2\\8\end{pmatrix} & \begin{pmatrix}2&7\\8&4\end{pmatrix} & \begin{pmatrix}2&7&3\\8&4&1\end{pmatrix}\end{pmatrix}$$

$$\mathsf{TRs}\begin{pmatrix}2&7&3\\8&4&1\end{pmatrix} = \begin{pmatrix}\begin{pmatrix}2&7&3\end{pmatrix} & \begin{pmatrix}7&3\end{pmatrix} & \begin{pmatrix}3\end{pmatrix}\\ \begin{pmatrix}2&7&3\\8&4&1\end{pmatrix} & \begin{pmatrix}7&3\\4&1\end{pmatrix} & \begin{pmatrix}3\\1\end{pmatrix}\end{pmatrix}$$

$$\mathsf{BLs}\begin{pmatrix}2&7&3\\8&4&1\end{pmatrix} = \begin{pmatrix}\begin{pmatrix}2\\8\end{pmatrix} & \begin{pmatrix}2&7\\8&4\end{pmatrix} & \begin{pmatrix}2&7&3\\8&4&1\end{pmatrix}\\ \begin{pmatrix}8\end{pmatrix} & \begin{pmatrix}8&4\end{pmatrix} & \begin{pmatrix}8&4&1\end{pmatrix}\end{pmatrix}$$

$$\mathsf{BRs}\begin{pmatrix}2&7&3\\8&4&1\end{pmatrix} = \begin{pmatrix}\begin{pmatrix}2&7&3\\8&4&1\end{pmatrix} & \begin{pmatrix}7&3\\4&1\end{pmatrix} & \begin{pmatrix}3\\1\end{pmatrix}\\ \begin{pmatrix}8&4&1\end{pmatrix} & \begin{pmatrix}4&1\end{pmatrix} & \begin{pmatrix}1\end{pmatrix}\end{pmatrix}$$

For each element of the input array, the first four functions generate rectangles consisting of elements on its top, left, bottom, and right, respectively. The rest for functions generate rectangles consisting of elements on its top left, top right, bottom left, and bottom right, respectively.

Those functions can be used to perform directed computations on arrays.

Example 5.21 (Top-left prefix sums)**.** Given a two-dimensional array, two associative, abiding binary operators, find sums for all top-left prefixes of the given array.

For example, the top-left prefix sums of $\begin{pmatrix}2&7&3\\8&4&1\end{pmatrix}$ with $+$ operators is given as follows.

$$\begin{aligned}
&\mathsf{map}\ (\mathsf{reduce}\ (+,+))\ \left(\mathsf{TLs}\begin{pmatrix}2&7&3\\8&4&1\end{pmatrix}\right)\\
=&\begin{pmatrix}\mathsf{reduce}\ (+,+)\ \begin{pmatrix}2\end{pmatrix} & \mathsf{reduce}\ (+,+)\ \begin{pmatrix}2&7\end{pmatrix} & \mathsf{reduce}\ (+,+)\ \begin{pmatrix}2&7&3\end{pmatrix}\\ \mathsf{reduce}\ (+,+)\ \begin{pmatrix}2\\8\end{pmatrix} & \mathsf{reduce}\ (+,+)\ \begin{pmatrix}2&7\\8&4\end{pmatrix} & \mathsf{reduce}\ (+,+)\ \begin{pmatrix}2&7&3\\8&4&1\end{pmatrix}\end{pmatrix}\\
=&\begin{pmatrix}2&9&12\\10&21&25\end{pmatrix}
\end{aligned}$$

Function *surrounds* to Generate Surrounding Rectangles

For each element in the given array, the function *surrounds* generates the all rectangles surrounding the element. This function corresponds to the simultaneous use of the previously-defined eight functions Ts, Ls, Bs, Rs, TLs, TRs, BLs, and BRs, except that each rectangle generated by *surrounds* does not contain the element itself.

The function *surrounds* is defined by the following two-phase computation: (1) computation of the parts of the northwest (i.e. c, n, w and nw) by scan, and (2) that of the other parts by scanr.

$$
\begin{aligned}
& \mathit{surrounds} = \mathsf{scanr}(\oplus_r, \otimes_r) \circ \mathsf{map}\ f_r \circ \mathsf{scan}(\oplus_f, \otimes_f)\ \circ \mathsf{map}\ f_f \\
& \ \textbf{where} \\
& f_f\ a = (a, \mathit{NIL}, \mathit{NIL}, \mathit{NIL}) \\
& (c_a, n_a, w_a, nw_a) \oplus_f (c_b, n_b, w_b, nw_b) = (\underbrace{c_b}_{c}, \underbrace{n_a \ominus |c_a| \ominus n_b}_{n}, \underbrace{w_b}_{w}, \underbrace{nw_a \ominus w_a \ominus nw_b}_{nw}) \\
& (c_a, n_a, w_a, nw_a) \otimes_f (c_b, n_b, w_b, nw_b) = (\underbrace{c_b}_{c}, \underbrace{n_b}_{n}, \underbrace{w_a \phi |c_a| \phi w_b}_{w}, \underbrace{nw_a \phi n_a \phi nw_b}_{nw}) \\
& f_r\ (c, n, w, nw) = (c, n, \mathit{NIL}, \mathit{NIL}, w, \mathit{NIL}, nw, \mathit{NIL}, \mathit{NIL}) \\
& (c_a, n_a, s_a, e_a, w_a, ne_a, nw_a, se_a, sw_a) \oplus_r (c_b, n_b, s_b, e_b, w_b, ne_b, nw_b, se_b, sw_b) \\
& \ = (\underbrace{c_a}_{c}, \underbrace{n_a}_{n}, \underbrace{s_a \ominus |c_b| \ominus s_b}_{s}, \underbrace{e_a}_{e}, \underbrace{w_a}_{w}, \underbrace{ne_a}_{ne}, \underbrace{nw_a}_{nw}, \underbrace{se_a \ominus e_b \ominus se_b}_{se}, \underbrace{sw_a \ominus w_b \ominus sw_b}_{sw}) \\
& (c_a, n_a, s_a, e_a, w_a, ne_a, nw_a, se_a, sw_a) \otimes_r (c_b, n_b, s_b, e_b, w_b, ne_b, nw_b, se_b, sw_b) \\
& \ = (\underbrace{c_a}_{c}, \underbrace{n_a}_{n}, \underbrace{s_a}_{s}, \underbrace{e_a \phi |c_b| \phi e_b}_{e}, \underbrace{w_a}_{w}, \underbrace{ne_a \phi n_b \phi ne_b}_{ne}, \underbrace{nw_a}_{nw}, \underbrace{se_a \phi s_b \phi se_b}_{se}, \underbrace{sw_a}_{sw})
\end{aligned}
$$

Here, NIL is a special value to indicate that there is no value, and we treat it as an identity of $\ominus$ and ϕ for simplification of the notation. Thus, $\mathit{NIL} \ominus x = x$, $x \ominus \mathit{NIL} = x$, $\mathit{NIL} \phi x = x$, and $x \phi \mathit{NIL} = x$. Each element of the resulting array is a tuple of nine elements. The meaning of each element of the tuple is as follows: c is the center element; s is an array of the elements on the south of the element; similarly n, e and w are arrays of the elements on the north, east and west respectively; ne, nw, se and sw are arrays of the elements on the northeast, northwest, southeast and southwest.

The function *surrounds* can be used to describe computations known as matrix-convolutions [Jai89, Rus06, GW06], in which each element in the resulting array depends on its surrounding elements. This set of computations includes important and fundamental problems such as image filters, difference methods, and the N-body problem.

Example 5.22 (Image filter)**.** An image filter by matrix-convolution is described

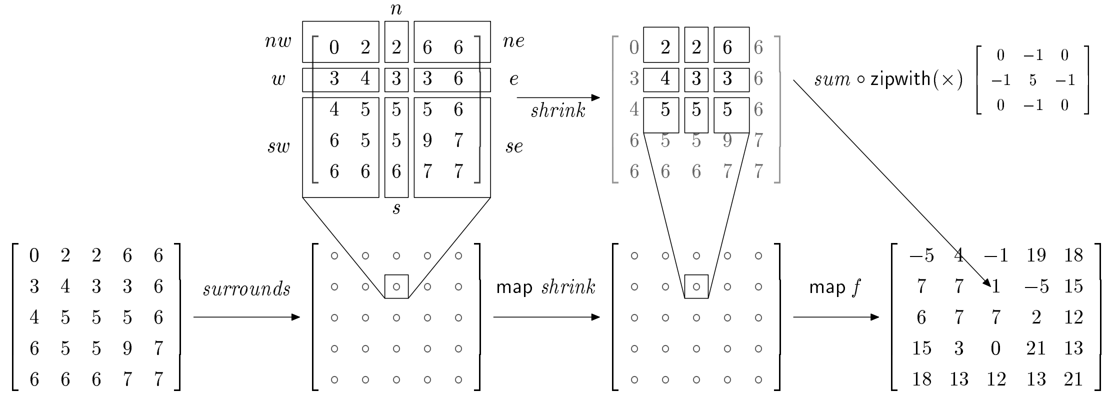


Figure 5.6. An image of the sharpen filter written with *surrounds*

with *surrounds* as follows.

$$
\begin{array}{l}
\mathit{imagefilter}\ \mathit{ker} \ = \ \mathsf{map}\ (\mathit{conv}\,\mathit{ker}) \circ \mathsf{map}\ \mathit{shrink}_1 \circ \mathit{surrounds} \\
\quad \textbf{where} \\
\quad \mathit{shrink}_1 = \mathit{id} \times B \times T \times L \times R \times BL \times BR \times TL \times TR \\
\quad B = (\![|\cdot|, \gg, \phi]\!), \quad T = (\![|\cdot|, \ll, \phi]\!), \quad L = (\![|\cdot|, \ominus, \ll]\!), \quad R = (\![|\cdot|, \ominus, \gg]\!), \\
\quad BL = (\![|\cdot|, \gg, \ll]\!), BR = (\![|\cdot|, \gg, \gg]\!), TL = (\![|\cdot|, \ll, \ll]\!), TR = (\![|\cdot|, \ll, \gg]\!)
\end{array}
$$

The function *imagefilter ker* is an image filter with the coefficient matrix *ker*, which is used to compute weighted sum of the surrounding pixels. The shrink_1 reduces each part of the gathered surrounding elements to the element closest to the center, and the function *conv ker* calculates the weighted sum of them. The functions B and T take the bottom row and the top row of the input array respectively. Similarly, each of L, R, BL, BR, TL and TR takes corresponding part of the input array. Figure 5.6 shows an image of execution of the sharpen-filter by the above general program.

The following example computes for each element the maximum of the column and the row in which the element belongs to.

Example 5.23 (Cross maximum)**.** Given a two-dimensional arrays, find for each element the maximum in the cross of the column and the row that the element belongs to.

The cross max can be computed by the following program. The shrink_{max} reduces each part of the gathered surrounding elements to the biggest element in the part. The function max_5 takes the maximum of the column and the row including the

center element.

$$\begin{aligned}
crossmax &= \mathsf{map}\ max_5 \circ \mathsf{map}\ shrink_{max} \circ surrounds \\
\textbf{where}\ shrink_{max} &= max \times \cdots \times max \\
max &= (\![id, \uparrow, \uparrow]\!) \\
max_5\ (c, n, s, e, w, _, _, _, _) &= c \uparrow n \uparrow s \uparrow e \uparrow w
\end{aligned}$$

Function $rects'$ to Generate All Rectangles

Finally, we will introduce the function $rects'$ to generate all rectangles in the given array.

$$rects' = flatten \circ \mathsf{map}\ \mathsf{TLs} \circ \mathsf{BRs}$$

For example, applying $rects'$ to $\begin{pmatrix} 2 & 7 & 3 \\ 8 & 4 & 1 \end{pmatrix}$, we get the following result.

$$\begin{aligned}
& rects' \begin{pmatrix} 2 & 7 & 3 \\ 8 & 4 & 1 \end{pmatrix} \\
= \;& (flatten \circ \mathsf{map}\ (\mathsf{TLs})) \begin{pmatrix} \begin{pmatrix} 2 & 7 & 3 \\ 8 & 4 & 1 \end{pmatrix} & \begin{pmatrix} 7 & 3 \\ 4 & 1 \end{pmatrix} & \begin{pmatrix} 3 \\ 1 \end{pmatrix} \\ \begin{pmatrix} 8 & 4 & 1 \end{pmatrix} & \begin{pmatrix} 4 & 1 \end{pmatrix} & \begin{pmatrix} 1 \end{pmatrix} \end{pmatrix} \\
= \;& flatten \begin{pmatrix} \begin{pmatrix} (2) & (2\ \ 7) & (2\ \ 7\ \ 3) \\ \begin{pmatrix} 2 \\ 8 \end{pmatrix} & \begin{pmatrix} 2 & 7 \\ 8 & 4 \end{pmatrix} & \begin{pmatrix} 2 & 7 & 3 \\ 8 & 4 & 1 \end{pmatrix} \\ (8) & (8\ \ 4) & (8\ \ 4\ \ 1) \end{pmatrix} & \begin{pmatrix} (7) & (7\ \ 3) \\ \begin{pmatrix} 7 \\ 4 \end{pmatrix} & \begin{pmatrix} 7 & 3 \\ 4 & 1 \end{pmatrix} \\ (4) & (4\ \ 1) \end{pmatrix} & \begin{pmatrix} (3) \\ \begin{pmatrix} 3 \\ 1 \end{pmatrix} \\ (1) \end{pmatrix} \end{pmatrix} \\
= \;& \begin{pmatrix} (2) & (2\ \ 7) & (2\ \ 7\ \ 3) & (7) & (7\ \ 3) & (3) \\ \begin{pmatrix} 2 \\ 8 \end{pmatrix} & \begin{pmatrix} 2 & 7 \\ 8 & 4 \end{pmatrix} & \begin{pmatrix} 2 & 7 & 3 \\ 8 & 4 & 1 \end{pmatrix} & \begin{pmatrix} 7 \\ 4 \end{pmatrix} & \begin{pmatrix} 7 & 3 \\ 4 & 1 \end{pmatrix} & \begin{pmatrix} 3 \\ 1 \end{pmatrix} \\ (8) & (8\ \ 4) & (8\ \ 4\ \ 1) & (4) & (4\ \ 1) & (1) \end{pmatrix}
\end{aligned}$$

The example application described with $rects'$ is the maximum rectangle problem, which has already been discussed in Section 4.3.4. This is a simplified problem of pattern matching in two-dimensional data structures.

Example 5.24 (Maximum rectangle sum)**.** Its statement is as follows: Given a two-dimensional array, find the maximum of sums of all the rectangle areas in the array. This problem was originated by Bentley [Ben84a, Ben84b] and improved by Takaoka [Tak02].

For example, the answer is 15 for the following data.

$$\begin{pmatrix} 3 & -1 & \mathbf{4} & \mathbf{-1} & -5 \\ 1 & -4 & \mathbf{-1} & \mathbf{5} & -3 \\ -4 & 1 & \mathbf{5} & \mathbf{3} & 1 \end{pmatrix}$$

Here, the sub-rectangle contributing the answer is denoted by bold numbers. The program to solve the problem is given as follows.

$$\mathsf{reduce}\ (\uparrow, \uparrow) \circ \mathsf{map}\ (\mathsf{reduce}\ (+, +)) \circ rects'$$

5.3.3 Useful Properties on Predicates

We will define some properties on predicates, which will be used in development of optimization theorems for each function shown in the previous section.

The following closure properties are defined on predicates for deriving efficient sequential implementation [Zan92, Jeu93]. Here, $\Rightarrow$ means the implication.

Definition 5.25 (Prefix-closed predicate)**.** Predicate p is said to be prefix-closed if the following equation holds for all x and y.

$$p\ (x +\!\!+\ y) \Rightarrow p\ (x)$$

□

Definition 5.26 (Suffix-closed predicate)**.** Predicate p is said to be suffix-closed if the following equation holds for all x and y.

$$p\ (x +\!\!+\ y) \Rightarrow p\ (y)$$

□

Definition 5.27 (Segment-closed predicate)**.** Predicate p is said to be segment-closed if p is prefix- and suffix closed, i.e. the following equation holds for all x, y and z.

$$p\ (x +\!\!+\ y +\!\!+\ z) \Rightarrow p\ (y)$$

□

Basically, those properties guarantee that we can know the result of the predicate for a long list from the result for smaller segments of the list. The following property is the almost converse of the segment-closedness.

Definition 5.28 (Overlap-closed predicate)**.** Predicate p is said to be overlap-closed if the following equation holds for all x, y and z.

$$p\ (x +\!\!+\ y) \wedge p\ (y +\!\!+\ z) \wedge y \neq [\,] \Rightarrow p\ (x +\!\!+\ y +\!\!+\ z)$$

□

Prefix-closed property plays the most important role in derivation of efficient *sequential* implementation of segment problems [Zan92, Jeu93]. Other properties are used as auxiliary tools to improve the derived implementation. However, in derivation of efficient *parallel* implementation, the pair of segment-closed property and overlap-closed property plays the most important role, which will be shown later.

A candidate of segment-closed and overlap-closed predicates is given by a relation [Zan92].

Definition 5.29 (Relational predicate)**.** Given a relation R, relational predicate p_R is defined as follows.

$$p_R\ (x) = \bigwedge\{aRb \ \mid\ [a,b] \in \mathit{segments}\ x\}$$

Here, *segments* x returns a set of segments (contiguous subsequences) of x, i.e., $\mathit{segments}\ x = \{\ y \mid u +\!\!+ y +\!\!+ v = x\ \}$. Note that $[a,b] \in \mathit{segments}\ x$ means that a and b are successive elements in x, since *segments* x generates all contiguous subsequences of x and $[a,b] \in \mathit{segments}\ x$ takes such subsequences of two elements. Relational predicate p_R is true for the given list x, if all successive elements a and b in x satisfy the relation R, i.e. $aRb = \mathsf{True}$. □

We can also show the converse, i.e., a segment-closed, overlap-closed predicate is a relational predicate. The following lemma shows the relation between relational predicates and segment-closed, overlap-closed predicates.

Lemma 5.30 (Relational predicate)**.** Given predicate p that is true for all singletons and empty list, the following statements are equivalent.

1. p is segment-closed and overlap-closed.

2. p is relational.

Proof. 2 ⇒ 1) Since p is relational, there exists a relation R and the following equation holds.

$$p\ (x) = \bigwedge\{aRb \mid [a,b] \in \mathit{segments}\ x\}$$

First, we show that p is segment-closed.

$$\begin{array}{ll}
 & p\ (x +\!\!+ y +\!\!+ z) \\
= & \{\ \text{unfolding } p\ \} \\
 & \bigwedge\{aRb \mid [a,b] \in \mathit{segments}\ (x +\!\!+ y +\!\!+ z)\} \\
\Rightarrow & \{\ \ \mathit{segments}\ y \subseteq \mathit{segments}\ (x +\!\!+ y +\!\!+ z)\ \} \\
 & \bigwedge\{aRb \mid [a,b] \in \mathit{segments}\ y\} \\
= & \{\ \text{folding } p\ \} \\
 & p\ (y)
\end{array}$$

Next, we show that p is overlap-closed.

$$\begin{array}{ll}
 & p\ (x +\!\!+ y) \wedge p\ (y +\!\!+ z) \wedge y \neq [\,] \\
= & \{\ \text{unfolding } p\ \} \\
 & \bigwedge\{aRb \mid [a,b] \in \mathit{segments}\ (x +\!\!+ y)\} \wedge \bigwedge\{aRb \mid [a,b] \in \mathit{segments}\ (y +\!\!+ z)\} \wedge y \neq [\,] \\
\Rightarrow & \left\{ \begin{array}{l} y \neq [\,] \Rightarrow \\ \quad \{[a,b] \mid [a,b] \in \mathit{segments}\ (x +\!\!+ y)\} \cup \{[a,b] \mid [a,b] \in \mathit{segments}\ (y +\!\!+ z)\} \\ \qquad = \{[a,b] \mid [a,b] \in \mathit{segments}\ (x +\!\!+ y +\!\!+ z)\} \end{array} \right\} \\
 & \bigwedge\{aRb \mid [a,b] \in \mathit{segments}\ (x +\!\!+ y +\!\!+ z)\} \\
= & \{\ \text{folding } p\ \} \\
 & p\ (x +\!\!+ y +\!\!+ z)
\end{array}$$

$1 \Rightarrow 2$) Letting $R = \{(a,b) \mid p\ ([a,b])\}$, we show $p = p_R$ by induction.
For base cases, we have $p\ ([\,]) = p\ ([a]) = p_R\ ([]) = p_R\ ([a]) = \mathsf{True}$ by assumption.
For induction case, we have $p\ ([a] +\!\!\!+\ x) = p_R\ ([a] +\!\!\!+\ x)$ by the following calculation.

$$\begin{array}{ll} & p\ ([a] +\!\!\!+\ x) \\ = & \{\ \text{segment-closed and overlap-closed}\ \} \\ & p\ (x) \wedge p\ ([a] +\!\!\!+\ [head\ x]) \\ = & \{\ \text{induction hypothesis and definition of } R\ \} \\ & p_R\ (x) \wedge aR(head\ x) \\ = & \{\ \text{definition of } p_R\ \} \\ & p_R\ ([a] +\!\!\!+\ x) \end{array}$$

Thus, $p = p_R$ and p is relational. □

In the above lemma, we assumed that the predicate p is true for all singletons and empty list for simplicity. However, we can remove this assumption by letting values of relational predicate p_R in the proof be those of the given predicate p.

The following predicates are examples of relational predicates.

$$\begin{array}{lcl} ascending\ (x) & = & p_<\ (x) \\ descending\ (x) & = & p_>\ (x) \\ flat\ (x) & = & p_=\ (x) \\ smooth_c\ (x) & = & p_{R_c}\ (x) \quad \textbf{where}\ \ aR_cb = |a-b| \leq c \end{array}$$

Predicate *ascending* is true when the given list is ascendingly sorted, while *descending* is true for descendingly sorted lists. Predicate *flat* is true if the all elements in the given list are the same. Predicate $smooth_c$ is true if the maximum of differences of successive elements is less than or equal to c. Especially, $flat = smooth_0$.

It is worth mentioning about composition of predicates [Zan92]. Each closure property of prefix-closed, suffix-closed, segment-closed and overlap-closed is closed under disjunction. Each closure property of prefix-closed, suffix-closed, and segment-closed is closed also under conjunction, but overlap-closed property is not closed under conjunction. For example, *ascending* and *descending* are both overlap-closed, but $ascending \vee descending$ is not overlap-closed since we can make a counterexample: $ascending(x +\!\!\!+\ y) \wedge descending(y +\!\!\!+\ z) \wedge y \neq [\,]$ implies neither $ascending(x +\!\!\!+\ y +\!\!\!+\ z)$ nor $descending(x +\!\!\!+\ y +\!\!\!+\ z)$.

5.3.4 Optimization Theorems for the Nested Reductions

In this section, we will give optimization theorems for nested reductions described with the generation functions discussed so far. We will develop a theory of optimizations for each generation function one by one, because in our theory the data structure and the computation structure are closely related to each other, and the structures of nested data are determined by the generation functions.

First, we will give theories for nested reductions on lists. Then, we will proceed to theories for two-dimensional arrays.

Optimization Theory of inits

In this section, we will give a theory for optimization of nested reductions described with inits.

The following two lemmas are well-known lemmas of inits [Bir87].

Lemma 5.31 (Inits-map promotion)**.** For function f, the following equation holds.

$$\mathsf{map}\,(\mathsf{map}\,f) \circ \mathsf{inits} = \mathsf{inits} \circ \mathsf{map}\,f$$

This lemma gives us a way to promote the application of function f through inits. The number of applications of function f on the left hand side is $n(n+1)/2$, while that on the right hand side is n. So, transformation from the left hand side to the right hand side improves the efficiency. Also, the lemma enables us to ignore a function g in the generate-and-test specification $(\![f, \oplus]\!) \circ \mathsf{map}\,(\![g, \otimes]\!) \circ \mathsf{filter}\,p \circ \mathsf{inits}$, since we can replace g with the identity function id and apply g to the input of inits beforehand (of course, we need some tricks to move through $\mathsf{filter}\,p$).

Lemma 5.32 (Scan)**.** For any associative binary operator $\oplus$, the following equation holds.

$$\mathsf{scan}\ (\oplus) = \mathsf{map}\,(\mathsf{reduce}\,(\oplus)) \circ \mathsf{inits}$$

This lemma gives us the relation between computations of prefix sums by scan and by nested reductions with inits. It means that we can use scan to compute nested reductions if the outer reduction is $id = (\![[\cdot], +\!\!+]\!)$, which results in the smaller number of uses of the operator $\oplus$.

The following theorem gives efficient parallel implementation of nested reductions for inits when two reductions have distributivity. One of the most famous problems for which this theorem is applicable is maximum initial-segment sum problem (also known as maximum prefix sum problem), which is an instance of maximum marking problems [SHTO00, Bir01].

Theorem 5.33 (Maximum initial-segment sum)**.** Provided that $\oplus$ is associative, and $\otimes$ is associative and left-distributive over $\oplus$, the following equation holds.

$$\begin{array}{l} \mathsf{reduce}\,(\oplus) \circ \mathsf{map}\,(\mathsf{reduce}\,(\otimes)) \circ \mathsf{inits} \;=\; \pi_1 \circ (\![pair, \odot]\!) \\ \textbf{where}\quad (i_1, s_1) \odot (i_2, s_2) = (i_1 \oplus (s_1 \otimes i_2), s_1 \otimes s_2) \\ \qquad\qquad pair\ a = (a, a) \end{array}$$

Proof. We show the theorem by induction.

For base case, we have $LHS\ [a] = RHS\ [a]$ by the following calculation.

$$
\begin{array}{cl}
 & LHS\ [a] \\
= & \{\ \text{LHS}\ \} \\
 & (\mathsf{reduce}\,(\oplus) \circ \mathsf{map}\,(\mathsf{reduce}\,(\otimes)) \circ \mathsf{inits})\ [a] \\
= & \{\ \text{definition of } \mathsf{inits},\ \mathsf{map},\ \text{and } \mathsf{reduce}\ \} \\
 & a \\
= & \{\ \text{definition of } \pi_1, \text{ homomorphism, and } pair\ \} \\
 & (\pi_1 \circ (\![pair, \odot]\!))\ [a] \\
= & \{\ \text{RHS}\ \} \\
 & RHS\ [a]
\end{array}
$$

For induction case, we have $LHS\ (x +\!\!+\, y) = RHS\ (x +\!\!+\, y)$ by the following calculation.

$$
\begin{array}{cl}
 & LHS\ (x +\!\!+\, y) \\
= & \{\ \text{LHS}\ \} \\
 & (\mathsf{reduce}\,(\oplus) \circ \mathsf{map}\,(\mathsf{reduce}\,(\otimes)) \circ \mathsf{inits})\ (x +\!\!+\, y) \\
= & \{\ \text{definition of function composition, unfolding } \mathsf{inits},\ last\ (\mathsf{inits}\ x) = x\ \} \\
 & (\mathsf{reduce}\,(\oplus) \circ \mathsf{map}\,(\mathsf{reduce}\,(\otimes)))\ (\mathsf{inits}\ x +\!\!+\, \mathsf{map}\,(x +\!\!+)\,(\mathsf{inits}\ y)) \\
= & \{\ \text{definition of function composition, } \mathsf{map} \text{ and } \mathsf{reduce}\ \} \\
 & \mathsf{reduce}\,(\oplus)\,(\mathsf{map}\,(\mathsf{reduce}\,(\otimes))\,(\mathsf{inits}\ x)) \\
 & \qquad \oplus \mathsf{reduce}\,(\oplus)\,(\mathsf{map}\,((\mathsf{reduce}\,(\otimes))\ x \otimes)(\mathsf{map}\,(\mathsf{reduce}\,(\otimes))\,(\mathsf{inits}\ y))) \\
= & \{\ \text{distributivity of } \otimes\ \} \\
 & \mathsf{reduce}\,(\oplus)\,(\mathsf{map}\,(\mathsf{reduce}\,(\otimes))\,(\mathsf{inits}\ x)) \\
 & \qquad \oplus (\mathsf{reduce}\,(\otimes)\ x \otimes \mathsf{reduce}\,(\oplus)\,(\mathsf{map}\,(\mathsf{reduce}\,(\otimes))\,(\mathsf{inits}\ y))) \\
= & \{\ \text{induction hypothesis}\ \} \\
 & (\pi_1 \circ (\![pair, \odot]\!))\ x \oplus (\mathsf{reduce}\,(\otimes)\ x \otimes (\pi_1 \circ (\![pair, \odot]\!))\ y) \\
= & \{\ \mathsf{reduce}\,(\otimes)\ x = \pi_2\ ((\![pair, \odot]\!)\ x)\ \text{(shown below)}\ \} \\
 & (\pi_1 \circ (\![pair, \odot]\!))\ x \oplus ((\pi_2\ ((\![pair, \odot]\!)\ x)) \otimes (\pi_1 \circ (\![pair, \odot]\!))\ y) \\
= & \{\ \text{definition of } \odot\ \} \\
 & \pi_1\ ((\![pair, \odot]\!)\ x \odot (\![pair, \odot]\!)\ y) \\
= & \{\ \text{definition of homomorphism, and function composition}\ \} \\
 & (\pi_1 \circ (\![pair, \odot]\!))\ (x +\!\!+\, y) \\
= & \{\ \text{RHS}\ \} \\
 & RHS\ (x +\!\!+\, y)
\end{array}
$$

Finally, we show that $\mathsf{reduce}\,(\otimes)\,x = \pi_2\ ((\![pair, \odot]\!)\ x)$. For base case, we have

$$
\begin{array}{ll}
 & LHS\ [a] \\
= & \{\ \text{LHS}\ \} \\
 & \mathsf{reduce}\,(\otimes)[a] \\
= & \{\ \text{definition of } \mathsf{reduce}\ \} \\
 & a \\
= & \{\ \text{definition of } \pi_2\text{, homomorphism, and } pair\ \} \\
 & \pi_2\ ((\![pair, \odot]\!)\ [a]) \\
= & \{\ \text{RHS}\ \} \\
 & RHS\ [a]
\end{array}
$$

For induction case, we have

$$
\begin{array}{ll}
 & LHS\ (x +\!\!+\ y) \\
= & \{\ \text{LHS}\ \} \\
 & \mathsf{reduce}\,(\otimes)\,(x +\!\!+\ y) \\
= & \{\ \text{definition of } \mathsf{reduce}\ \} \\
 & \mathsf{reduce}\,(\otimes)\,x \otimes \mathsf{reduce}\,(\otimes)\,y \\
= & \{\ \text{induction hypothesis}\ \} \\
 & \pi_2\ ((\![pair, \odot]\!)\ x) \otimes \pi_2\ ((\![pair, \odot]\!)\ y) \\
= & \{\ \text{definition of } \odot\ \} \\
 & \pi_2\ ((\![pair, \odot]\!)\ x \odot (\![pair, \odot]\!)\ y) \\
= & \{\ \text{definition of homomorphism}\ \} \\
 & \pi_2\ ((\![pair, \odot]\!)\ (x +\!\!+\ y)) \\
= & \{\ \text{RHS}\ \} \\
 & RHS\ (x +\!\!+\ y)
\end{array}
$$

□

The resulting program (the right hand side) of the theorem uses only one reduction with the new operator $\odot$, while the original program (the left hand side) uses two nested reductions with inits. The new operator $\odot$ is applied to tuples. The first element of a tuple is equal to the result of the original program. The second element of the tuple is equal to the reduction of the same input with operator $\otimes$, which is used to improve efficiency of the program by reusing the partial results effectively.

The resulting program (the right hand side of the equation) is more efficient than the original program. This is because the cost of the new operator is proportional to the cost of operators in the original reductions. Also, the resulting program has no intermediate data structures.

The following theorem extends Theorem 5.33, which allows filtering with the given predicate. It gives efficient parallel implementation of nested reductions for inits when two reductions have distributivity and the predicate is relational.

Theorem 5.34 (Maximum p-initial-segment sum)**.** Provided that $\oplus$ is associative, $\otimes$ is associative and left-distributive over $\oplus$, and predicate p is relational, the following equation holds.

$$
\begin{array}{l}
\mathsf{reduce}\,(\oplus) \circ \mathsf{map}\,(\mathsf{reduce}\,(\otimes)) \circ \mathsf{filter}\,p \circ \mathsf{inits} = \pi_1 \circ (\![\mathit{pentuple}, \boxdot]\!) \\
\quad \textbf{where} \\
\qquad (i_1, s_1, h_1, l_1, p_1) \boxdot (i_2, s_2, h_2, l_2, p_2) = \\
\qquad\qquad (i, s_1 \otimes s_2, h_1 \ll h_2, l_1 \gg l_2, p_1 \wedge p_2 \wedge p\ ([l_1, h_2])) \\
\qquad\quad \textbf{where}\ \ i = i_1 \oplus \textbf{if}\ p_1 \wedge p\ ([l_1, h_2])\ \textbf{then}\ s_1 \otimes i_2\ \textbf{else}\ \imath_{\oplus} \\
\qquad \mathit{pentuple}\ a = (a, a, a, a, T)
\end{array}
$$

Proof. We show the theorem by induction.

For base case, we have

$$
\begin{array}{cl}
 & \mathit{LHS}\ [a] \\
= & \{\ \ \mathrm{LHS}\ \ \} \\
 & (\mathsf{reduce}\,(\oplus) \circ \mathsf{map}\,(\mathsf{reduce}\,(\otimes)) \circ \mathsf{filter}\,p \circ \mathsf{inits})\ [a] \\
= & \{\ \ \text{definition of } \mathsf{inits}, \mathsf{map}\,, \mathsf{reduce} \text{ and } \mathsf{filter}\,, p \text{ is true for singleton}\ \} \\
 & a \\
= & \{\ \ \text{definition of } \pi_1, \text{ homomorphism, and } \mathit{pentuple}\ \} \\
 & (\pi_1 \circ (\![\mathit{pentuple}, \boxdot]\!))\ [a] \\
= & \{\ \ \mathrm{RHS}\ \ \} \\
 & \mathit{RHS}\ [a]
\end{array}
$$

For induction case, we have the following calculation.

$$
\begin{array}{cl}
 & \mathit{LHS}\ (x +\!\!+\, y) \\
= & \{\ \ \mathrm{LHS}\ \ \} \\
 & (\mathsf{reduce}\,(\oplus) \circ \mathsf{map}\,(\mathsf{reduce}\,(\otimes)) \circ \mathsf{filter}\,p \circ \mathsf{inits})\ (x +\!\!+\, y) \\
= & \{\ \ \text{definition of } \mathsf{inits}\ \ \} \\
 & (\mathsf{reduce}\,(\oplus) \circ \mathsf{map}\,(\mathsf{reduce}\,(\otimes)) \circ \mathsf{filter}\,p)\ (\mathsf{inits}\ x +\!\!+ (\mathsf{map}\,(x +\!\!+)\,(\mathsf{inits}\ y))) \\
= & \{\ \ \text{definition of } \mathsf{filter}\,, \mathsf{reduce}, \text{ and } \mathsf{map}\ \ \} \\
 & (\mathsf{reduce}\,(\oplus) \circ \mathsf{map}\,(\mathsf{reduce}\,(\otimes)) \circ \mathsf{filter}\,p \circ \mathsf{inits})\ x \\
 & \quad \oplus (\mathsf{reduce}\,(\oplus) \circ \mathsf{map}\,(\mathsf{reduce}\,(\otimes)) \circ \mathsf{filter}\,p)\ (\mathsf{map}\,(x +\!\!+)\,(\mathsf{inits}\ y))
\end{array}
$$

To proceed more, we calculate a part of the above equation $\mathsf{filter}\,p\,(\mathsf{map}\,(x +\!\!+)\,(\mathsf{inits}\ y))$ as follows.

$$
\begin{array}{cl}
 & \mathsf{filter}\,p\,(\mathsf{map}\,(x +\!\!+)\,(\mathsf{inits}\ y)) \\
= & \{\ \ p \text{ is relational: } p\ (x +\!\!+\, y) \Rightarrow p\ (y)\ \ \} \\
 & \mathsf{filter}\,p\,(\mathsf{map}\,(x +\!\!+)\,(\mathsf{filter}\,p\,(\mathsf{inits}\ y))) \\
= & \{\ \ p \text{ is relational: } p\ (x +\!\!+\, y) = p\ (y) \wedge p\ (x) \wedge p\ ([\mathit{last}\ x, \mathit{head}\ y])\ \ \} \\
 & \textbf{if}\ p\ (x) \wedge p\ ([\mathit{last}\ x, \mathit{head}\ y])\ \textbf{then}\ \mathsf{map}\,(x +\!\!+)\,(\mathsf{filter}\,p\,(\mathsf{inits}\ y))\ \textbf{else}\ [\,]
\end{array}
$$

Using this result, we proceed a wider part of the above equation as follows.

$$
\begin{array}{ll}
& (\mathsf{reduce}\,(\oplus) \circ \mathsf{map}\,(\mathsf{reduce}\,(\otimes)) \circ \mathsf{filter}\,p)\ (\mathsf{map}\,(x{+\!\!+})\,(\mathsf{inits}\ y)) \\
= & \{\ \text{above calculation}\ \} \\
& (\mathsf{reduce}\,(\oplus) \circ \mathsf{map}\,(\mathsf{reduce}\,(\otimes))) \\
& \quad (\mathbf{if}\ p\ (x) \wedge p\ ([\mathit{last}\ x, \mathit{head}\ y])\ \mathbf{then}\ \mathsf{map}\,(x{+\!\!+})\,(\mathsf{filter}\,p\,(\mathsf{inits}\ y))\ \mathbf{else}\ [\,]) \\
= & \{\ \text{distributivity of if-then-else, definition of } \mathsf{map} \text{ and } \mathsf{reduce}\ \} \\
& \mathbf{if}\ p\ (x) \wedge p\ ([\mathit{last}\ x, \mathit{head}\ y]) \\
& \quad \mathbf{then}\ \mathsf{reduce}\,(\oplus)\,(\mathsf{map}\,((\mathsf{reduce}\,(\otimes))\,x\otimes)\,(\mathsf{map}\,(\mathsf{reduce}\,(\otimes))\,(\mathsf{filter}\,p\,(\mathsf{inits}\ y)))) \\
& \quad \mathbf{else}\ \imath_\oplus \\
= & \{\ \text{distributivity of } \otimes\ \} \\
& \mathbf{if}\ p\ (x) \wedge p\ ([\mathit{last}\ x, \mathit{head}\ y]) \\
& \quad \mathbf{then}\ \mathsf{reduce}\,(\otimes)\,x \otimes \mathsf{reduce}\,(\oplus)\,(\mathsf{map}\,(\mathsf{reduce}\,(\otimes))\,(\mathsf{filter}\,p\,(\mathsf{inits}\ y)))\ \mathbf{else}\ \imath_\oplus
\end{array}
$$

Now, we resume the suspended calculation for induction case.

$$
\begin{array}{l}
\quad \mathit{LHS}\ (x +\!\!+ y) \\
= \{\ \text{resume}\ \} \\
\quad (\mathsf{reduce}\,(\oplus) \circ \mathsf{map}\,(\mathsf{reduce}\,(\otimes)) \circ \mathsf{filter}\,p \circ \mathsf{inits})\ x \\
\qquad \oplus (\mathsf{reduce}\,(\oplus) \circ \mathsf{map}\,(\mathsf{reduce}\,(\otimes)) \circ \mathsf{filter}\,p)\ (\mathsf{map}\,(x{+\!\!+})\,(\mathsf{inits}\ y)) \\
= \{\ \text{above calculation}\ \} \\
\quad (\mathsf{reduce}\,(\oplus) \circ \mathsf{map}\,(\mathsf{reduce}\,(\otimes)) \circ \mathsf{filter}\,p \circ \mathsf{inits})\ x \\
\qquad \otimes\,\mathbf{if}\ p\ (x) \wedge p\ ([\mathit{last}\ x, \mathit{head}\ y]) \\
\qquad\quad \mathbf{then}\ \mathsf{reduce}\,(\otimes)\,x \otimes \mathsf{reduce}\,(\oplus)\,(\mathsf{map}\,(\mathsf{reduce}\,(\otimes))\,(\mathsf{filter}\,p\,(\mathsf{inits}\ y))) \\
\qquad\quad \mathbf{else}\ \imath_\oplus \\
= \{\ \text{induction hypothesis}\ \} \\
\quad (\pi_1 \circ (\![\mathit{pentuple}, \boxdot]\!))\ x \\
\qquad \otimes\,\mathbf{if}\,p\,(x) \wedge p\,([\mathit{last}\,x, \mathit{head}\,y])\,\mathbf{then}\ \mathsf{reduce}\,(\otimes)\,x \otimes (\pi_1 \circ (\![\mathit{pentuple}, \boxdot]\!))\,y\,\mathbf{else}\,\imath_\oplus \\
= \{\ (_, \mathsf{reduce}\,(\otimes)\,x, \mathit{head}\ x, \mathit{last}\ x, p\ (x)) = (\![\mathit{pentuple}, \boxdot]\!)\,x\ \text{(shown below)}\ \} \\
\quad (\pi_1 \circ (\![\mathit{pentuple}, \boxdot]\!))\ x \\
\qquad \otimes\,\mathbf{if}\ (\pi_5 \circ (\![\mathit{pentuple}, \boxdot]\!))\ x \wedge p\ ([(\pi_4 \circ (\![\mathit{pentuple}, \boxdot]\!))\ x, (\pi_3 \circ (\![\mathit{pentuple}, \boxdot]\!))\ y]) \\
\qquad\ \mathbf{then}\ (\pi_2 \circ (\![\mathit{pentuple}, \boxdot]\!))\ x \otimes (\pi_1 \circ (\![\mathit{pentuple}, \boxdot]\!))\ y\ \mathbf{else}\ \imath_\oplus \\
= \{\ \text{definition of } \boxdot\ \} \\
\quad (\pi_1 \circ (\![\mathit{pentuple}, \boxdot]\!))\ (x +\!\!+ y) \\
= \{\ \text{RHS}\ \} \\
\quad \mathit{RHS}\ (x +\!\!+ y)
\end{array}
$$

Finally, we show $(_, \mathsf{reduce}\,(\otimes)\,x, \mathit{head}\ x, \mathit{last}\ x, p\ (x)) = (\![\mathit{pentuple}, \boxdot]\!)\,x$. For base case, we have

$$
\begin{array}{l}
\quad (_, \mathsf{reduce}\,(\otimes)\,[a], \mathit{head}\ [a], \mathit{last}\ [a], p\ ([a])) \\
= \{\ \text{definition of each function, } p \text{ is relational}\ \} \\
\quad (_, a, a, a, T) \\
= \{\ \text{definition of homomorphism and } \mathit{pentuple}\ \} \\
\quad (\![\mathit{pentuple}, \boxdot]\!)\ [a]
\end{array}
$$

For induction case, we have

$$
\begin{aligned}
& (_, \mathsf{reduce}\,(\otimes)\,(x +\!\!+\, y), head\ (x +\!\!+\, y), last\ (x +\!\!+\, y), p\ (x +\!\!+\, y)) \\
=\ & \{\ \text{definition of each function, } p \text{ is relational}\ \} \\
& (_, \mathsf{reduce}\,(\otimes)\,x \otimes \mathsf{reduce}\,(\otimes)\,y, \\
& \quad head\ x \ll head\ y, last\ x \gg head\ y, p\ x \wedge p\ y \wedge p\ ([head\ x, last\ y])) \\
=\ & \{\ \text{definition of } \boxdot\ \} \\
& (_, \mathsf{reduce}\,(\otimes)\,x, head\ x, last\ x, p\ x) \boxdot (_, \mathsf{reduce}\,(\otimes)\,y, head\ y, last\ y, p\ y) \\
=\ & \{\ \text{induction hypothesis}\ \} \\
& (\![pentuple, \boxdot]\!)\,x \boxdot (\![pentuple, \boxdot]\!)\,y \\
=\ & \{\ \text{definition of homomorphism}\ \} \\
& (\![pentuple, \boxdot]\!)\,(x +\!\!+\, y)
\end{aligned}
$$

Thus, we have $(_, \mathsf{reduce}\,(\otimes)\,x, head\ x, last\ x, p\ (x)) = (\![pentuple, \boxdot]\!)\,x$. □

Similar to the previous theorem, the resulting program (the right hand side) of the theorem uses only one reduction with the new operator $\boxdot$, while the original program (the left hand side) uses two nested reductions with inits. The cost of the new operator is proportional to the cost of operators in the original reductions and the application of p. So, the resulting program is more efficient than the original program.

The new operator $\boxdot$ is applied to pentuples. The first element of a pentuple is equal to the result of the original program. The second element of the pentuple is equal to the reduction of the same input with operator $\otimes$, which is used to improve efficiency of the program by reusing the partial results effectively. The third and forth elements are the edge elements of the input. Those edge elements are used to check whether results from two recursions in divide-and-conquer computation can be connected to make a better solution. Since the predicate p is relational, we can check the connectability by using only elements on the edge. The fifth element is a Boolean value that is the result of p applied to the input.

Here, we can reduce the size of pentuples by eliminating the fifth element (it corresponds to $p\ (x)$) for simplicity, when we introduce an assumption that $\imath_\otimes$ is the zero of $\otimes$, i.e. $\imath_\oplus \otimes a = \imath_\oplus$.

Corollary 5.35 (Maximum p-initial-segment sum (simplified)). Provided that $\oplus$ is associative, $\otimes$ is associative and left-distributive over $\oplus$, the identity $\imath_\oplus$ is the zero of $\otimes$, and predicate p is relational, the following equation holds.

$$
\begin{aligned}
& \mathsf{reduce}\,(\oplus) \circ \mathsf{map}\,(\mathsf{reduce}\,(\otimes)) \circ \mathsf{filter}\,p \circ \mathsf{inits} = \pi_1 \circ (\![quadruple, \boxtimes]\!) \\
& \quad \textbf{where} \\
& \quad (i_1, s_1, h_1, l_1) \boxtimes (i_2, s_2, h_2, l_2) = (i_1 \oplus (s_1 \otimes i_2)_{l_1,h_2}, (s_1 \otimes s_2)_{l_1,h_2}, h_1 \ll h_2, l_1 \gg l_2) \\
& \quad quadruple\ a = (a, a, a, a) \\
& \quad (a)_{l,h} = \textbf{if}\ p\ ([l, h])\ \textbf{then}\ a\ \textbf{else}\ \imath_\oplus
\end{aligned}
$$

Proof. To simplify the result of the theorem, we add an invariant to the result of Theorem 5.34. The invariant added to the pentuple (i, s, h, t, p) is $\neg p \Rightarrow s = \imath_\oplus$. In the following calculation, we derive a operator slightly changed from that of Theorem 5.34.

$$
\begin{array}{ll}
& i \\
= & \{ \text{ definition } \} \\
& i_1 \oplus \textbf{if } p_1 \wedge p\ ([l_1, h_2]) \textbf{ then } s_1 \otimes i_2 \textbf{ else } \imath_\oplus \\
= & \{ \text{ splitting condition } \} \\
& i_1 \oplus \textbf{if } p\ ([l_1, h_2]) \textbf{ then } (\textbf{if } p_1 \textbf{ then } (s_1 \otimes i_2) \textbf{ else } \imath_\oplus) \textbf{ else } \imath_\oplus \\
= & \{ \text{ assumption: } \imath_\oplus \text{ is the zero } \} \\
& i_1 \oplus \textbf{if } p\ ([l_1, h_2]) \textbf{ then } ((\textbf{if } p_1 \textbf{ then } s_1 \textbf{ else } \imath_\oplus) \otimes i_2) \textbf{ else } \imath_\oplus \\
= & \{ \ s'_1 = \textbf{if } p_1 \textbf{ then } s_1 \textbf{ else } \imath_\oplus \ \} \\
& i_1 \oplus \textbf{if } p\ ([l_1, h_2]) \textbf{ then } (s'_1 \otimes i_2) \textbf{ else } \imath_\oplus \\
= & \{ \text{ definition of } (a)_{t,h} \} \\
& i_1 \oplus (s'_1 \otimes i_2)_{l_1, h_2}
\end{array}
$$

Computation of $s' = \textbf{if } p \textbf{ then } s \textbf{ else } \imath_\oplus$ is as follows.

$$
\begin{array}{ll}
& s' \\
= & \{ \text{ definition } \} \\
& \textbf{if } p \textbf{ then } s \textbf{ else } \imath_\oplus \\
= & \{ \text{ computation of } s \text{ and } p \ \} \\
& \textbf{if } p_1 \wedge p_2 \wedge p\ ([l_1, h_2]) \textbf{ then } s_1 \otimes s_2 \textbf{ else } \imath_\oplus \\
= & \{ \text{ splitting condition } \} \\
& \textbf{if } p\ ([l_1, h_2]) \textbf{ then } (\textbf{if } p_1 \textbf{ then } s_1 \textbf{ else } \imath_\oplus) \otimes (\textbf{if } p_1 \textbf{ then } s_2 \textbf{ else } \imath_\oplus) \textbf{ else } \imath_\oplus \\
= & \{ \text{ definition of s' } \} \\
& \textbf{if } p\ ([l_1, h_2]) \textbf{ then } s'_1 \otimes s'_2 \textbf{ else } \imath_\oplus \\
= & \{ \text{ definition of } (a)_{t,h} \} \\
& (s'_1 \otimes s'_2)_{l_1, h_2}
\end{array}
$$

Now, we can use s' instead of s and p in the pentuple (i, s, h, t, p), since p and s are not used by computation of other parts. Thus, replacing s and p in the pentuple with s' and rename s' as s, we get the reduced operator $\boxtimes$. □

The resulting program (the right hand side) uses the new reduction operator $\boxtimes$ that is applied on quadruples. The difference from the result of Theorem 5.34 is that the second element becomes the identity $\imath_\oplus$ of $\oplus$ when the input does not satisfy the predicate p.

Optimization Theory of tails

In this section, we will give a theory for optimization of nested reductions described with tails, which is similar to that of inits.

The following two lemmas are well-known lemmas of tails [Bir87].

Lemma 5.36 (Tails-map promotion)**.** For function f, the following equation holds.

$$\mathsf{map}\,(\mathsf{map}\,f) \circ \mathsf{tails} = \mathsf{tails} \circ \mathsf{map}\,f$$

This lemma gives us a way to promote the application of function f through tails. The number of applications of function f on the left hand side is $n(n+1)/2$, while that on the right hand side is n. So, transformation from the left hand side to the right hand side improves the efficiency. Also, the lemma enables us to ignore a function g in the generate-and-test specification $(\![f, \oplus]\!) \circ \mathsf{map}\,(\![g, \otimes]\!) \circ \mathsf{filter}\,p \circ \mathsf{tails}$, since we can replace g with the identity function id and apply g to the input of tails beforehand (of course, we need some tricks to through $\mathsf{filter}\,p$).

Lemma 5.37 (Scanr)**.** For any associative binary operator $\oplus$, the following equation holds.

$$\mathsf{scanr}\ (\oplus) = \mathsf{map}\,(\mathsf{reduce}\,(\oplus)) \circ \mathsf{tails}$$

This lemma gives us the relation between computations of suffix sums by scanr and by nested reductions with tails. It means that we can use scanr to compute nested reductions if the outer reduction is $id = (\![[\cdot], +\!\!+]\!)$, which results in the smaller number of uses of the operator $\oplus$.

The following theorem gives efficient parallel implementation of nested reductions for tails when two reductions have distributivity. One of the most famous problems for which this theorem is applicable is maximum tail-segment sum problem (also known as maximum suffix sum problem), which is an instance of maximum marking problems [SHTO00, Bir01].

Theorem 5.38 (Maximum tail-segment sum)**.** Provided that $\oplus$ is associative, and $\otimes$ is associative and right-distributive over $\oplus$, the following equation holds.

$$\begin{aligned} \mathsf{reduce}\,(\oplus) \circ \mathsf{map}\,(\mathsf{reduce}\,(\otimes)) \circ \mathsf{tails} \quad &= \quad \pi_1 \circ (\![pair, \circledast]\!) \\ \textbf{where} \quad (t_1, s_1) \circledast (t_2, s_2) &= ((t_1 \otimes s_2) \oplus t_2, s_1 \otimes s_2) \\ pair\ a &= (a, a) \end{aligned}$$

Proof. Similar to the proof of Theorem 5.33. □

Similar to the previous theorem, the resulting program (the right hand side) of the theorem uses only one reduction with the new operator $\circledast$, while the original program (the left hand side) uses two nested reductions with tails. The cost of the new operator is proportional to the cost of operators in the original reductions. So, the resulting program is more efficient than the original program.

The new operator $\circledast$ is applied to tuples. The first element of a tuple is equal to the result of the original program. The second element of the tuple is equal to the reduction of the same input with operator $\otimes$, which is used to improve efficiency of the program by reusing the partial results effectively.

The following theorem extends Theorem 5.38, which allows filtering with the given predicate. It gives efficient parallel implementation of nested reductions for tails when two reductions have distributivity and the predicate is relational.

Theorem 5.39 (Maximum p-tail-segment sum)**.** Provided that $\oplus$ is associative, $\otimes$ is associative and right-distributive over $\oplus$, and predicate p is relational, the following equation holds.

$$
\begin{array}{l}
\mathsf{reduce}\,(\oplus) \circ \mathsf{map}\,(\mathsf{reduce}\,(\otimes)) \circ \mathsf{filter}\, p \circ \mathsf{tails} = \pi_1 \circ (\![pentuple, \boxast]\!) \\
\quad \textbf{where} \\
\qquad (t_1, s_1, h_1, l_1, p_1) \boxast (t_2, s_2, h_2, l_2, p_2) = \\
\qquad\qquad (t, s_1 \otimes s_2, h_1 \ll h_2, l_1 \gg l_2, p_1 \wedge p_2 \wedge p\ ([l_1, h_2])) \\
\qquad\quad \textbf{where}\ \ i = (\textbf{if}\ p_2 \wedge p\ ([l_1, h_2])\ \textbf{then}\ t_1 \otimes s_2\ \textbf{else}\ \imath_\oplus) \oplus t_2 \\
\qquad pentuple\ a = (a, a, a, a, T)
\end{array}
$$

Proof. Similar to the proof of Theorem 5.34. □

The resulting program (the right hand side) of the theorem uses only one reduction with the new operator $\boxast$, while the original program (the left hand side) uses two nested reductions with tails. The cost of the new operator is proportional to the cost of operators in the original reductions and the application of p. So, the resulting program is more efficient than the original program.

The new operator $\boxast$ is applied to pentuples. The first element of a pentuple is equal to the result of the original program. The second element of the pentuple is equal to the reduction of the same input with operator $\otimes$, which is used to improve efficiency of the program by reusing the partial results effectively. The third and forth elements are the edge elements of the input. Those edge elements are used to check whether results from two recursions in divide-and-conquer computation can be connected to make a better solution. Since the predicate p is relational, we can check the connectability by using only elements on the edge. The fifth element is a Boolean value that is the result of p applied to the input.

Here, we can reduce the size of pentuples by eliminating the fifth element (it corresponds to $p\ (x)$) for simplicity, when we3 introduce an assumption that $\imath_\otimes$ is the zero of $\otimes$, i.e. $a \otimes \imath_\oplus = \imath_\oplus$.

Corollary 5.40 (Maximum p-tail-segment sum (simplified))**.** Provided that $\oplus$ is associative, $\otimes$ is associative and right-distributive over $\oplus$, the identity $\imath_\oplus$ is the zero of $\otimes$, and predicate p is relational, the following equation holds.

$$
\begin{array}{l}
\mathsf{reduce}\,(\oplus) \circ \mathsf{map}\,(\mathsf{reduce}\,(\otimes)) \circ \mathsf{filter}\, p \circ \mathsf{tails} = \pi_1 \circ (\![quadruple, \boxtimes]\!) \\
\quad \textbf{where} \\
\quad (t_1, s_1, h_1, l_1) \boxtimes (t_2, s_2, h_2, l_2) = ((t_1 \otimes s_2)_{l_1, h_2} \oplus t_2, (s_1 \otimes s_2)_{l_1, h_2}, h_1 \ll h_2, l_1 \gg l_2) \\
\quad quadruple\ a = (a, a, a, a) \\
\quad (a)_{l,h} = \textbf{if}\ p\ ([l, h])\ \textbf{then}\ a\ \textbf{else}\ \imath_\oplus
\end{array}
$$

Proof. Similar to the proof of Corollary 5.35. □

The resulting program (the right hand side) uses the new reduction operator $\boxtimes$ that is applied on quadruples. The difference from the result of Theorem 5.39 is that

the second element becomes the identity $\imath_\oplus$ of $\oplus$ when the input does not satisfy the predicate p.

A sufficient condition of the assumption that $\imath_\oplus$ is the zero of $\otimes$ is that $\otimes$ distributes over $\oplus$. So, we will use this condition in implementation of the library.

Optimization Theory of segs

In this section, we will give a theory for optimization of nested reductions described with segs.

The following lemma is well-known lemma of segs [Bir87].

Lemma 5.41 (Segs-map promotion)**.** For function f, the following equation holds.

$$\mathsf{map}\,(\mathsf{map}\,f) \circ \mathsf{segs} = \mathsf{segs} \circ \mathsf{map}\,f$$

The lemma gives us a way to promote the application of function f through segs. The number of applications of function f on the left hand side is $\mathrm{O}(n^3)$, while that on the right hand side is n. So, transformation from the left hand side to the right hand side improves the efficiency. Also, the lemma enables us to ignore a function g in the generate-and-test specification $(\![f, \oplus]\!) \circ \mathsf{map}\,(\![g, \otimes]\!) \circ \mathsf{filter}\,p \circ \mathsf{segs}$, since we can replace g with the identity function id and apply g to the input of segs beforehand (of course, we need some tricks to through $\mathsf{filter}\,p$).

The following theorem gives efficient parallel implementation of nested reductions for segs when two reductions have distributivity. One of the most famous problems for which this theorem is applicable is maximum segment sum problem [Bir87], which is an instance of maximum marking problems [SHTO00, Bir01].

Theorem 5.42 (Maximum segment sum)**.** Provided that $\oplus$ is associative and commutative, and $\otimes$ is associative and distributive over $\oplus$, the following equation holds.

$$\begin{array}{l}
\mathsf{reduce}\,(\oplus) \circ \mathsf{map}\,(\mathsf{reduce}\,(\otimes)) \circ \mathsf{segs} = \pi_1 \circ (\![\mathit{quadruple}, \circledcirc]\!) \\
\quad \textbf{where} \\
\quad\; (m_1, t_1, i_1, s_1) \circledcirc (m_2, t_2, i_2, s_2) = \\
\quad\quad (m_1 \oplus m_2 \oplus (t_1 \otimes i_2), (t_1 \otimes s_2) \oplus t_2, i_1 \oplus (s_1 \otimes i_2), s_1 \otimes s_2) \\
\quad\; \mathit{quadruple}\ a = (a, a, a, a)
\end{array}$$

Proof. We can prove the theorem by the following calculation.

$$\begin{array}{cl}
 & \mathit{LHS} \\
= & \{\ \text{LHS}\ \} \\
 & \mathsf{reduce}\,(\oplus) \circ \mathsf{map}\,(\mathsf{reduce}\,(\otimes)) \circ \mathsf{segs} \\
= & \{\ \text{definition of } \mathsf{segs}\ \} \\
 & \mathsf{reduce}\,(\oplus) \circ \mathsf{map}\,(\mathsf{reduce}\,(\otimes)) \circ \mathsf{reduce}\,(+\!\!+) \circ \mathsf{map\,inits} \circ \mathsf{tails} \\
= & \{\ \text{promotion of } \mathsf{map}\ \} \\
 & \mathsf{reduce}\,(\oplus) \circ \mathsf{reduce}\,(+\!\!+) \circ \mathsf{map}\,(\mathsf{map}\,(\mathsf{reduce}\,(\otimes))) \circ \mathsf{map\,inits} \circ \mathsf{tails} \\
= & \{\ \text{promotion of } \mathsf{reduce}\ \} \\
 & \mathsf{reduce}\,(\oplus) \circ \mathsf{map}\,(\mathsf{reduce}\,(\oplus)) \circ \mathsf{map}\,(\mathsf{map}\,(\mathsf{reduce}\,(\otimes))) \circ \mathsf{map\,inits} \circ \mathsf{tails}
\end{array}$$

$$
\begin{array}{ll}
= & \{\ \text{distributivity of } \mathsf{map}\ \} \\
& \mathsf{reduce}\,(\oplus) \circ \mathsf{map}\,((\mathsf{reduce}\,(\oplus)) \circ (\mathsf{map}\,(\mathsf{reduce}\,(\otimes))) \circ \mathsf{inits}) \circ \mathsf{tails} \\
= & \{\ \text{Theorem 5.33}\ \} \\
& \mathsf{reduce}\,(\oplus) \circ \mathsf{map}\,(\pi_1 \circ (\![pair, \odot]\!)) \circ \mathsf{tails} \\
= & \{\ \text{making } \oplus' \text{ so that } (a_1, b_1) \oplus' (a_2, b_2) = (a_1 \oplus a_2, b_1 \oplus b_2)\text{, Lemma 5.36}\ \} \\
& \pi_1 \circ \mathsf{reduce}\,(\oplus') \circ \mathsf{map}\,(\mathsf{reduce}\,(\odot)) \circ \mathsf{tails} \circ \mathsf{map}\ pair \\
= & \{\ \text{Theorem 5.38 (commutativity of } \oplus \text{ guarantees distributivity of } \odot \text{ over } \oplus')\} \\
& \pi_1 \circ \pi_1 \circ (\![pair, \circledast]\!) \circ \mathsf{map}\ pair \\
= & \{\ \text{fusing two } \pi_1\text{s, and two } pair\text{s}\ \} \\
& \pi_1 \circ (\![quadruple, \circledcirc]\!) \\
= & \{\ \text{RHS}\ \} \\
& RHS
\end{array}
$$

Definition of $\circledast$ in the above calculation is given as follows.

$$
\begin{array}{l}
\quad ((m_1, t_1), (i_1, s_1)) \circledast ((m_2, t_2), (i_2, s_2)) \\
= \{\ \text{definition of } \circledast \text{ in Theorem 5.38}\ \} \\
\quad (((m_1, t_1) \odot (i_2, s_2)) \oplus' (m_2, t_2), (i_1, s_1) \odot (i_2, s_2)) \\
= \{\ \text{definition of } \odot \text{ in Theorem 5.33}\ \} \\
\quad ((m_1 \oplus (t_1 \otimes i_2), t_1 \otimes s_2) \oplus' (m_2, t_2), (i_1 \oplus (s_1 \otimes i - 2), s_1 \otimes s_2)) \\
= \{\ \text{definition of } \oplus' \text{ shown above}\ \} \\
\quad ((m_1 \oplus (t_1 \otimes i_2) \oplus m_2, (t_1 \otimes s_2) \oplus t_2), (i_1 \oplus (s_1 \otimes i - 2), s_1 \otimes s_2))
\end{array}
$$

Flattening the nested pair into quadruple, we get the definition of $\circledcirc$. □

The resulting program (the right hand side) of the theorem uses only one reduction with the new operator $\circledcirc$, while the original program (the left hand side) uses two nested reductions with segs. The cost of the new operator is proportional to the cost of operators in the original reductions. So, the resulting program is more efficient than the original program.

The new operator $\circledcirc$ is applied to quadruples. The first element of a tuple is equal to the result of the original program. The second and the third elements are results of the original nested reductions with tails and inits. The last element is equal to the reduction of the same input with operator $\otimes$. These extra elements are used to improve efficiency of the program by reusing the partial results effectively.

The following theorem extends Theorem 5.42, which allows filtering with the given predicate. It gives efficient parallel implementation of nested reductions for segs when two reductions have distributivity and the predicate is relational. Derivation of efficient sequential implementations for those programs with filtering is shown in [Zan92, Jeu93].

Theorem 5.43 (Maximum p-segment sum)**.** Provided that $\oplus$ is associative and commutative, $\otimes$ is associative and distributive over $\oplus$, the identity $\imath_\oplus$ is the zero of $\otimes$, and predicate p is relational, the following equation holds.

$$
\begin{array}{l}
\mathsf{reduce}\,(\oplus) \circ \mathsf{map}\,(\mathsf{reduce}\,(\otimes)) \circ \mathsf{filter}\,p \circ \mathsf{segs} = \pi_1 \circ (\![\mathit{hextuple}, \boxdot]\!) \\
\quad \textbf{where} \\
\quad (m_1, t_1, i_1, s_1, h_1, l_1) \boxdot (m_2, t_2, i_2, s_2, h_2, l_2) \\
\qquad = (\; m_1 \oplus m_2 \oplus (t_1 \otimes i_2)_{l_1,h_2}, (t_1 \otimes s_2)_{l_1,h_2} \oplus t_2, i_1 \oplus (s_1 \otimes i_2)_{l_1,h_2}, \\
\qquad\qquad (s_1 \otimes s_2)_{l_1,h_2}, h_1 \ll h_2, l_1 \gg l_2) \\
\quad \mathit{hextuple}\ a = (a, a, a, a, a, a) \\
\quad (a)_{l,h} = \textbf{if}\ p\ ([l, h])\ \textbf{then}\ a\ \textbf{else}\ \imath_\oplus
\end{array}
$$

Proof. We can prove the theorem by the following calculation.

$$
\begin{array}{cl}
 & \mathit{LHS} \\
= & \{\ \text{LHS}\ \} \\
 & \mathsf{reduce}\,(\oplus) \circ \mathsf{map}\,(\mathsf{reduce}\,(\otimes)) \circ \mathsf{filter}\,p \circ \mathsf{segs} \\
= & \{\ \text{definition of } \mathsf{segs}\ \} \\
 & \mathsf{reduce}\,(\oplus) \circ \mathsf{map}\,(\mathsf{reduce}\,(\otimes)) \circ \mathsf{filter}\,p \circ \mathsf{reduce}\,(+\!\!+) \circ \mathsf{map\,inits} \circ \mathsf{tails} \\
= & \{\ \text{promotion of } \mathsf{filter},\ \mathsf{map} \text{ and } \mathsf{reduce}\ \} \\
 & \mathsf{reduce}\,(\oplus) \circ \mathsf{map}\,(\mathsf{reduce}\,(\oplus)) \circ \mathsf{map}\,(\mathsf{reduce}\,(\otimes)) \circ \mathsf{map}\,(\mathsf{filter}\,p) \circ \mathsf{map\,inits} \circ \mathsf{tails} \\
= & \{\ \mathsf{map} \text{ distributivity}\ \} \\
 & \mathsf{reduce}\,(\oplus) \circ \mathsf{map}\,((\mathsf{reduce}\,(\oplus)) \circ (\mathsf{reduce}\,(\otimes)) \circ (\mathsf{filter}\,p) \circ \mathsf{inits}) \circ \mathsf{tails} \\
= & \{\ \text{Collorary 5.35}\ \} \\
 & \mathsf{reduce}\,(\oplus) \circ \mathsf{map}\,(\pi_1 \circ (\![\mathit{quadruple}, \boxtimes]\!)) \circ \mathsf{tails} \\
= & \left\{ \begin{array}{l} \text{making } \oplus' \text{ such that} \\ (i_1, s_1, h_1, l_1) \oplus' (i_2, s_2, h_2, l_2) = (i_1 \oplus i_2, s_1 \oplus s_2, h_1 \ll h_2, l_1 \gg l_2) \end{array} \right\} \\
 & \pi_1 \circ \mathsf{reduce}\,(\oplus') \circ \mathsf{map}\,(\mathsf{reduce}\,(\boxtimes)) \circ \mathsf{tails} \circ \mathsf{map}\ \mathit{quadruple} \\
= & \{\ \text{Theorem 5.38 } (\boxtimes \text{ distributes over } \oplus')\ \} \\
 & \pi_1 \circ \pi_1 \circ (\![\mathit{pair}, \circledast]\!) \circ \mathsf{map}\ \mathit{quadruple} \\
= & \{\ \text{fusing two } \pi_1\text{s, fusing } \mathit{pair} \text{ and } \mathit{quadruple}\text{, removing duplicated parts}\ \} \\
 & \pi_1 \circ (\![\mathit{hextuple}, \boxdot]\!) \\
= & \{\ \text{RHS}\ \} \\
 & \mathit{RHS}
\end{array}
$$

Note that distributivity of $\boxtimes$ over $\oplus'$ is guaranteed for quadruples (i_1, s_1, h_1, l_1) and (i_2, s_2, h_2, l_2) of operands of $\oplus'$ when $l_1 = l_2$. Condition $l_1 = l_2$ holds in the above calculation since l_1 and l_2 are the last element of tail-segments of the same list.

Definition of $\circledast$ in the above calculation is given as follows.

$$
\begin{aligned}
& ((m_1,t_1,h_1,l_1),(i_1,s_1,k_1,n_1)) \circledast ((m_2,t_2,h_2,l_2),(i_2,s_2,k_2,n_2)) \\
= & \{ \text{ definition of } \circledast \text{ in Theorem 5.38 } \} \\
& (((m_1,t_1,h_1,l_1) \boxtimes (i_2,s_2,k_2,n_2)) \oplus' (m_2,t_2,h_2,l_2),(i_1,s_1,k_1,n_1) \boxtimes (i_2,s_2,k_2,n_2)) \\
= & \{ \text{ definition of } \boxtimes \text{ in Theorem 5.35 } \} \\
& ((m_1 \oplus (t_1 \otimes i_2)_{l_1,k_2},(t_1 \otimes s_2)_{l_1,k_2},h_1 \ll k_2,l_1 \gg n_2) \oplus' (m_2,t_2,h_2,l_2), \\
& \quad (i_1 \oplus (s_1 \otimes i_2)_{n_1,k_2},(s_1 \otimes s_2)_{n_1,k_2},k_1 \ll k_2,n_1 \gg n_2) \\
= & \{ \text{ definition of } \oplus' \text{ shown above } \} \\
& ((m_1 \oplus m_2 \oplus (t_1 \otimes i_2)_{l_1,k_2},(t_1 \otimes s_2)_{l_1,k_2} \oplus t_2,h_1 \ll k_2 \ll h_2,l_1 \gg n_2 \gg l_2), \\
& \quad (i_1 \oplus (s_1 \otimes i_2)_{n_1,k_2},(s_1 \otimes s_2)_{n_1,k_2},k_1 \ll k_2,n_1 \gg n_2)
\end{aligned}
$$

If $h_1 = k_1$, $l_1 = n_1$, $h_2 = k_2$ and $l_2 = n_2$, then we have $h_1 \ll k_2 \ll h_2 = k_1 \ll k_2$ and $l_1 \gg n_2 \gg l_2 = n_1 \gg n_2$. So, for octuple $((m,t,h,l),(i,s,k,n))$, we have invariant $h = k$ and $l = n$. Using this invariant, we can eliminate k and n from octuples and we get computation using hextuples. Finally, flattening the hextuples, we get the definition of $\circledcirc$. □

The resulting program (the right hand side) of the theorem uses only one reduction with the new operator $\boxdot$, while the original program (the left hand side) uses two nested reductions with segs. The cost of the new operator is proportional to the cost of operators in the original reductions. So, the resulting program is more efficient than the original program.

The new operator $\boxdot$ is applied to hextuples. The first element of a tuple is equal to the result of the original program. The second and the third elements are results of the original program in which segs is replaced with tails and inits. The fourth element is equal to the reduction of the same input with operator $\otimes$. The fifth and the sixth elements are the edge elements of the input. Those edge elements are used to check whether results from two recursions in divide-and-conquer computation can be connected to make a better solution. Since the predicate p is relational, we can check the connectability by using only elements on the edge. These extra elements are used to improve efficiency of the program by reusing the partial results effectively.

Optimization Theory of *neighbors*

In this section, we will give a theory for optimization of nested reductions described with *neighbors*.

The following theorem links the result of domain specific fusion optimization in Section 5.2 to the world of nested reductions.

Theorem 5.44 (Finite-window neighbors)**.** The following equation holds. Also, we can apply the domain-specific fusion optimization in Section 5.2 to the program on the right hand side.

$$
\begin{array}{l}
\mathsf{map}\ f \circ \mathit{neighbors}'\ \mathit{ls}\ \mathit{rs} = \mathsf{map}'\ f' \circ \mathit{zipshift}\ \mathit{ls}\ \mathit{rs} \\
\quad \textbf{where}\ \ f'(a_{-m}, \ldots, a_{-1}, a_0, a_1, \ldots, a_n) = f([a_{-m}, \ldots, a_{-1}], a_0, [a_1, \ldots, a_n]) \\
\qquad\qquad \mathit{zipshift}\ \mathit{ls}\ \mathit{rs}\ x = \\
\qquad\qquad\quad \begin{array}{lll}
\textbf{let}\ z_0 & = & x \\
\quad\ z_1 & = & \mathsf{shift}_{\ll}\ r_1\ z_0 \\
\quad\ z_2 & = & \mathsf{shift}_{\ll}\ r_2\ z_1 \\
 & \vdots & \\
\quad\ z_n & = & \mathsf{shift}_{\ll}\ r_n\ z_{n-1} \\
\quad\ z_{-1} & = & \mathsf{shift}_{\gg}\ l_1\ z_0 \\
 & \vdots & \\
\quad\ z_{-m} & = & \mathsf{shift}_{\ll}\ l_m\ z_{-(m-1)} \\
\quad\ [l_m, \ldots, l_1] & = & \mathit{ls} \\
\quad\ [r_1, \ldots, r_n] & = & \mathit{rs} \\
\end{array} \\
\qquad\qquad\quad \textbf{in}\ \mathsf{zip}\ z_{-m}\ (\mathsf{zip}\ z_{-(m-1)}\ (\ldots(\mathsf{zip}\ z_0\ (\ldots(\mathsf{zip}\ z_{n-1}\ z_n)\cdots))\cdots))
\end{array}
$$

Proof. Skeleton compositions in the program on the right hand side has the target pattern of the domain-specific fusion optimization in Section 5.2. Thus, we can apply the optimization to it.

The rest of the proof is to show the equation. We will show it by induction on the input. We will use rules shown in Appendix C in the following proof.

The base case is trivial. Supposing the input is $[a]$, we get $z_0 = [a]$, $z_i = r_i (i \geq 0)$, and $z_{-i} = l_i (i \geq 0)$, and thus the right hand side is $f([l_m, \ldots, l_1], a, [r_1, \ldots, r_n]$, which is equal to the left hand side.

Next, we will show The induction case. To this end, we will first do some small calculations, whose results will be used later.

$$
\begin{array}{cl}
 & \mathsf{map\ tail}\ (\mathsf{tails}\ (x +\!\!+\ y)) \\
= & \{\ \text{definition of}\ \mathsf{tails}\ \} \\
 & \mathsf{map\ tail}\ (\mathsf{map}\ (+\!\!+ \mathit{head}\ (\mathsf{tails}\ y))\ (\mathsf{tails}\ x) +\!\!+\ \mathsf{tails}\ y) \\
= & \{\ \mathit{head} \circ \mathsf{tails} = \mathit{id}\ \text{by homomorphism fusion}\ \} \\
 & \mathsf{map\ tail}\ (\mathsf{map}\ (+\!\!+ y)\ (\mathsf{tails}\ x) +\!\!+\ \mathsf{tails}\ y) \\
= & \{\ \text{definition of}\ \mathsf{map}\text{, and rule IV}\ \} \\
 & \mathsf{map}\ (+\!\!+ y)\ (\mathsf{map\ tail}\ (\mathsf{tails}\ x)) +\!\!+\ \mathsf{map\ tail}\ (\mathsf{tails}\ y)
\end{array}
$$

Similarly, we have the following equation.

$$
\mathsf{map\ init}\ (\mathsf{inits}\ (x +\!\!+\ y)) = \mathsf{map\ init}\ (\mathsf{inits}\ x) +\!\!+\ \mathsf{map}\ (x +\!\!+)\ \mathsf{map\ init}\ (\mathsf{inits}\ y)
$$

Using the above calculation, we have the following result for the left hand side. Some

where-clauses are omitted for readability.

$$
\begin{array}{ll}
 & (\mathsf{map}\ f \circ \mathit{neighbors}'\ \mathit{ls}\ \mathit{rs})\ (x +\!\!+ y) \\
= & \{\ \text{definition of } \mathit{neighbors}' \text{ and } \mathit{neighbors}\ \} \\
 & (\mathsf{map}\ f \circ \mathsf{map}\ (\mathit{taker}\ m \circ (\mathit{ls} +\!\!+) \times \mathit{id} \times \mathit{take}\ n \circ (+\!\!+ \mathit{rs})) \\
 & \quad \circ \mathit{zipP}_3 \circ ((\mathsf{map\ init} \circ \mathsf{inits}) \triangle \mathit{id} \triangle (\mathsf{map\ tail} \circ \mathsf{tails})))\ (x +\!\!+ y) \\
 & \mathbf{where}\ \ \mathit{zipP}_3\ (x, y, z) = \mathsf{zip}\ x\ y\ z m = \mathit{length}\ \mathit{ls} \\
 & \qquad\quad n = \mathit{length}\ \mathit{rs} \\
= & \{\ \text{the above calculation, and the definition of } \mathsf{map}\ \} \\
 & (\mathsf{map}\ f \circ \mathsf{map}\ (\mathit{taker}\ m \circ (\mathit{ls} +\!\!+) \times \mathit{id} \times \mathit{take}\ n \circ (+\!\!+ \mathit{rs})) \\
 & \ \circ \mathit{zipP}_3 \circ \mathsf{map}\ (\mathit{id} \times \mathit{id} \times (+\!\!+ y)) \circ ((\mathsf{map\ init} \circ \mathsf{inits}) \triangle \mathit{id} \triangle (\mathsf{map\ tail} \circ \mathsf{tails})))\ x \\
 & \ \ +\!\!+ (\mathsf{map}\ f \circ \mathsf{map}\ (\mathit{taker}\ m \circ (\mathit{ls} +\!\!+) \times \mathit{id} \times \mathit{take}\ n \circ (+\!\!+ \mathit{rs})) \\
 & \ \ \ \circ \mathit{zipP}_3 \circ \mathsf{map}\ ((x +\!\!+) \times \mathit{id} \times \mathit{id}) \circ ((\mathsf{map\ init} \circ \mathsf{inits}) \triangle \mathit{id} \triangle (\mathsf{map\ tail} \circ \mathsf{tails})))\ y \\
= & \{\ \text{fusion of } \mathit{zipPL}_3 \text{ and } \mathsf{map}, \text{ and associativity of } +\!\!+\ \} \\
 & (\mathsf{map}\ f \circ \mathsf{map}\ (\mathit{taker}\ m \circ (\mathit{ls} +\!\!+) \times \mathit{id} \times \mathit{take}\ n \circ (+\!\!+ (y +\!\!+ \mathit{rs}))) \\
 & \ \circ \mathit{zipP}_3 \circ ((\mathsf{map\ init} \circ \mathsf{inits}) \triangle \mathit{id} \triangle (\mathsf{map\ tail} \circ \mathsf{tails})))\ x \\
 & \ \ +\!\!+ (\mathsf{map}\ f \circ \mathsf{map}\ (\mathit{taker}\ m \circ ((\mathit{ls} +\!\!+ x) +\!\!+) \times \mathit{id} \times \mathit{take}\ n \circ (+\!\!+ \mathit{rs})) \\
 & \ \ \ \circ \mathit{zipP}_3 \circ ((\mathsf{map\ init} \circ \mathsf{inits}) \triangle \mathit{id} \triangle (\mathsf{map\ tail} \circ \mathsf{tails})))\ y \\
= & \{\ \mathit{take}\ n \circ \mathit{take}\ n = \mathit{take}\ n,\ \mathit{taker}\ m \circ \mathit{taker}\ m = \mathit{taker}\ m, \text{ and folding}\ \} \\
 & (\mathsf{map}\ f \circ \mathit{neighbors}'\ \mathit{ls}\ (\mathit{take}\ n\ (y +\!\!+ \mathit{rs})))\ x \\
 & \ +\!\!+ (\mathsf{map}\ f \circ \mathit{neighbors}'\ (\mathit{taker}\ m\ (\mathit{ls} +\!\!+ x))\ \mathit{rs}))\ y
\end{array}
$$

Similarly, we will calculate the right hand side. To this end, we first split the z_is in the right hand side into z_i^xs and z_i^ys. We use rule III to split them.

$$
\begin{array}{l}
z_0 = z_0^x +\!\!+ z_0^y = x +\!\!+ y \\
z_1 = z_1^x +\!\!+ z_1^y = \mathsf{shift}_{\ll}\ r_1\ (z_0^x +\!\!+ z_0^y) = \mathsf{shift}_{\ll}\ (\mathit{head}\ z_0^y)\ z_0^x +\!\!+ \mathsf{shift}_{\ll}\ r_1\ z_0^y \\
\vdots \\
z_n = z_n^x +\!\!+ z_n^y = \mathsf{shift}_{\ll}\ (\mathit{head}\ z_{n-1}^y)\ z_{n-1}^x +\!\!+ \mathsf{shift}_{\ll}\ r_n\ z_{n-1}^y \\
z_{-1} = z_{-1}^x +\!\!+ z_{-1}^y = \mathsf{shift}_{\gg}\ l_1\ z_0^x +\!\!+ \mathsf{shift}_{\gg}\ (\mathit{last}\ z_0^x)\ z_0^y \\
\vdots \\
z_{-m} = z_{-m}^x +\!\!+ z_{-m}^y = \mathsf{shift}_{\gg}\ l_m\ z_{-(m-1)}^x +\!\!+ \mathsf{shift}_{\gg}\ (\mathit{last}\ z_{-(m-1)}^x)\ z_{-(m-1)}^y
\end{array}
$$

Then, we show the following equation for $i \geq 0$ by induction on i.

$$z_i^y = \mathit{drop}\ i\ (y +\!\!+ [r_1, \ldots, r_i])$$

The base case $i = 0$ is trivial. The induction case is as follows.

$$
\begin{array}{ll}
 & z_i^y \\
= & \{ \text{ definition } \} \\
 & \mathsf{shift}_{\ll}\ r_i\ z_{i-1}^y \\
= & \{ \text{ induction hypothesis } \} \\
 & \mathsf{shift}_{\ll}\ r_i\ (drop\ (i-1)\ (y +\!\!+ [r_1, \ldots, r_{i-1}])) \\
= & \{ \text{ rule VI } \} \\
 & \mathsf{tail}\ (drop\ (i-1)\ (y +\!\!+ [r_1, \ldots, r_{i-1}])) +\!\!+ [r_i] \\
= & \{ \text{ definition of } drop \ \} \\
 & drop\ i\ (y +\!\!+ [r_1, \ldots, r_{i-1}]) +\!\!+ [r_i] \\
= & \{\ y +\!\!+ [r_1, \ldots, r_{i-1}] \text{ has at least } i \text{ elements } \} \\
 & drop\ i\ (y +\!\!+ [r_1, \ldots, r_{i-1}, r_i])
\end{array}
$$

Using the result we have the following result about z_i^x.

$$
\begin{array}{ll}
 & z_i^x \\
= & \{ \text{ definition } \} \\
 & \mathsf{shift}_{\ll}\ (head\ z_{i-1}^y)\ z_{i-1}^x \\
= & \{ \text{ the above result } \} \\
 & \mathsf{shift}_{\ll}\ (head\ (drop\ (i-1)\ (y +\!\!+ [r_1, \ldots, r_{i-1}])))\ z_{i-1}^x \\
= & \{ \text{ definition of } head\text{, and } (y +\!\!+ [r_1, \ldots, r_{i-1}]) \text{ has at least } i \text{ elements. } \} \\
 & \mathsf{shift}_{\ll}\ (head\ (drop\ (i-1)\ (y +\!\!+ rs)))\ z_{i-1}^x \\
= & \{\ i \text{ is smaller than or equal to } n\ \} \\
 & \mathsf{shift}_{\ll}\ (head\ (drop\ (i-1)\ (take\ n\ (y +\!\!+ rs))))\ z_{i-1}^x
\end{array}
$$

s

We can show similar results on z_{-i}^xs and z_{-i}^ys.

Summarizing above calculation, we have the following result on the right hand side.

$$
\begin{array}{l}
(\mathsf{map}'\ f' \circ zipshift\ ls\ rs)\ (x +\!\!+ y) \\
\quad = (\mathsf{map}'\ f' \circ zipshift\ ls\ (take\ n\ (y +\!\!+ rs)))\ x \\
\quad\quad +\!\!+ (\mathsf{map}'\ f' \circ zipshift\ (taker\ m\ (ls +\!\!+ x))\ rs)\ y
\end{array}
$$

Therefore, using the induction hypothesis, we have shown the induction case for our first equation. □

Optimization Theory of Ts, Ls, Bs, Rs, TLs, TRs, BLs, and BRs

Similar to the functions inits and tails, those functions have promotion lemmas for map.

Lemma 5.45 (Map promotions for Ts, Ls, Bs, Rs, TLs, TRs, BLs, and BRs)**.** For

any function f, the following equations hold.

$$\begin{array}{lcl}
\mathsf{map}\ (\mathsf{map}\ f) \circ \mathsf{Ts} &=& \mathsf{Ts} \circ \mathsf{map}\ f \\
\mathsf{map}\ (\mathsf{map}\ f) \circ \mathsf{Ls} &=& \mathsf{Ls} \circ \mathsf{map}\ f \\
\mathsf{map}\ (\mathsf{map}\ f) \circ \mathsf{Bs} &=& \mathsf{Bs} \circ \mathsf{map}\ f \\
\mathsf{map}\ (\mathsf{map}\ f) \circ \mathsf{Rs} &=& \mathsf{Rs} \circ \mathsf{map}\ f \\
\mathsf{map}\ (\mathsf{map}\ f) \circ \mathsf{TLs} &=& \mathsf{TLs} \circ \mathsf{map}\ f \\
\mathsf{map}\ (\mathsf{map}\ f) \circ \mathsf{TRs} &=& \mathsf{TRs} \circ \mathsf{map}\ f \\
\mathsf{map}\ (\mathsf{map}\ f) \circ \mathsf{BLs} &=& \mathsf{BLs} \circ \mathsf{map}\ f \\
\mathsf{map}\ (\mathsf{map}\ f) \circ \mathsf{BRs} &=& \mathsf{BRs} \circ \mathsf{map}\ f
\end{array}$$

Proof. We show the proof for Ts. The others can be proved similarly.

$$\begin{array}{ll}
& \mathsf{map}\ (\mathsf{map}\ f) \circ \mathsf{Ts} \\
= & \{ \text{ definition of } \mathsf{Ts} \ \} \\
& \mathsf{map}\ (\mathsf{map}\ f) \circ \mathsf{scan}\ (\ominus, \gg) \circ \mathsf{map}\ |\cdot| \\
= & \{ \text{ definition of } \mathsf{scan} \ \} \\
& \mathsf{map}\ (\mathsf{map}\ f) \circ (\!|\, |\cdot|, \oplus', \otimes' |\!) \circ \mathsf{map}\ |\cdot| \\
& \quad \textbf{where}\ sx \oplus' sy \ = \ sx \ominus \mathsf{map}_r\ (\mathsf{zipwith}\ (\ominus)\ (\mathit{bottom}\ sx))\ sy \\
& \qquad\qquad\ sx \otimes' sy \ = \ sx \varphi\ \mathsf{map}_c\ (\mathsf{zipwith}\ (\gg)\ (\mathit{right}\ sx))\ sy \\
= & \{ \text{ definition of } \mathsf{map}\text{, and fusion of homomorphism (Theorem 4.5) } \} \\
& (\!|\, \mathsf{map}\ (\mathsf{map}\ f) \circ |\cdot| \circ |\cdot|, \oplus', \otimes' |\!) \\
& \quad \textbf{where}\ sx \oplus' sy \ = \ sx \ominus \mathsf{map}_r\ (\mathsf{zipwith}\ (\ominus)\ (\mathit{bottom}\ sx))\ sy \\
& \qquad\qquad\ sx \otimes' sy \ = \ sx \varphi\ \mathsf{map}_c\ (\mathsf{zipwith}\ (\gg)\ (\mathit{right}\ sx))\ sy \\
= & \{ \ \mathsf{map}\ (\mathsf{map}\ f) \circ |\cdot| \circ |\cdot| = |\cdot| \circ |\cdot| \circ f \ \} \\
& (\!|\, |\cdot| \circ |\cdot| \circ f, \oplus', \otimes' |\!) \\
& \quad \textbf{where}\ sx \oplus' sy \ = \ sx \ominus \mathsf{map}_r\ (\mathsf{zipwith}\ (\ominus)\ (\mathit{bottom}\ sx))\ sy \\
& \qquad\qquad\ sx \otimes' sy \ = \ sx \varphi\ \mathsf{map}_c\ (\mathsf{zipwith}\ (\gg)\ (\mathit{right}\ sx))\ sy \\
= & \{ \ \text{definition of } \mathsf{map} \text{ and } \mathsf{scan} \ \} \\
& \mathsf{scan}\ (\ominus, \gg) \circ \mathsf{map}\ |\cdot| \circ \mathsf{map}\ f \\
= & \{ \ \text{definition of } \mathsf{Ts} \ \} \\
& \mathsf{Ts} \circ \mathsf{map}\ f
\end{array}$$

□

This lemma gives us a way to promote the application of function f through the generation functions. The number of applications of function f in the right hand side is smaller than that in the left hand side.

The combination of the above lemma and the following lemma gives as a way to promote homomorphism through the generation functions.

Lemma 5.46 (Reducue promotions for Ts, Ls, Bs, Rs, TLs, TRs, BLs, and BRs)**.** For any binary operators which are associative and have the abide property, the

following equations hold.

$$
\begin{array}{lcl}
\mathsf{map}\ (\mathsf{reduce}\ (\ominus, \odot)) \circ \mathsf{Ts} & = & \mathsf{scan}\ (\ominus, \gg) \\
\mathsf{map}\ (\mathsf{reduce}\ (\ominus, \odot)) \circ \mathsf{Ls} & = & \mathsf{scan}\ (\gg, \odot) \\
\mathsf{map}\ (\mathsf{reduce}\ (\ominus, \odot)) \circ \mathsf{Bs} & = & \mathsf{scanr}\ (\ominus, \ll) \\
\mathsf{map}\ (\mathsf{reduce}\ (\ominus, \odot)) \circ \mathsf{Rs} & = & \mathsf{scanr}\ (\ll, \odot) \\
\mathsf{map}\ (\mathsf{reduce}\ (\ominus, \odot)) \circ \mathsf{TLs} & = & \mathsf{scan}\ (\ominus, \odot) \\
\mathsf{map}\ (\mathsf{reduce}\ (\ominus, \odot)) \circ \mathsf{TRs} & = & \mathsf{scanr}\ (\ll, \odot) \circ \mathsf{scan}\ (\ominus, \gg) \\
\mathsf{map}\ (\mathsf{reduce}\ (\ominus, \odot)) \circ \mathsf{BLs} & = & \mathsf{scanr}\ (\ominus, \ll) \circ \mathsf{scan}\ (\gg, \odot) \\
\mathsf{map}\ (\mathsf{reduce}\ (\ominus, \odot)) \circ \mathsf{BRs} & = & \mathsf{scanr}\ (\ominus, \odot)
\end{array}
$$

Proof. We show the proof for Ts. The others can be proved similarly.

$$
\begin{array}{cl}
 & \mathsf{map}\ (\mathsf{reduce}\ (\ominus, \odot)) \circ \mathsf{Ts} \\
= & \{ \text{ definition of } \mathsf{Ts} \ \} \\
 & \mathsf{map}\ (\mathsf{reduce}\ (\ominus, \odot)) \circ \mathsf{scan}\ (\Theta, \gg) \circ \mathsf{map}\ |\cdot| \\
= & \{ \text{ definition of } \mathsf{scan} \ \} \\
 & \mathsf{map}\ (\mathsf{reduce}\ (\ominus, \odot)) \circ (\![|\cdot|, \oplus', \otimes']\!) \circ \mathsf{map}\ |\cdot| \\
 & \quad \mathbf{where}\ \ sx \oplus' sy \ = \ sx \,\Theta\, \mathsf{map}_r\ (\mathsf{zipwith}\ (\Theta)\ (bottom\ sx))\, sy \\
 & \qquad\qquad\ \ sx \otimes' sy \ = \ sx \,\Phi\, \mathsf{map}_c\ (\mathsf{zipwith}\ (\gg)\ (right\ sx))\, sy \\
= & \{ \text{ definition of } \mathsf{map}\text{, and fusion of homomorphism (Theorem 4.5) } \} \\
 & (\![\mathsf{map}\ (\mathsf{reduce}\ (\ominus, \odot)) \circ |\cdot| \circ |\cdot|, \oplus'', \otimes'']\!) \\
 & \quad \mathbf{where}\ \ sx \oplus'' sy \ = \ sx \,\Theta\, \mathsf{map}_r\ (\mathsf{zipwith}\ (\ominus)\ (bottom\ sx))\, sy \\
 & \qquad\qquad\ \ sx \otimes'' sy \ = \ sx \,\Phi\, \mathsf{map}_c\ (\mathsf{zipwith}\ (\gg)\ (right\ sx))\, sy \\
= & \{ \ \mathsf{map}\ (\mathsf{reduce}\ (\ominus, \odot)) \circ |\cdot| \circ |\cdot| = |\cdot| \ \} \\
 & (\![|\cdot|, \oplus'', \otimes'']\!) \\
 & \quad \mathbf{where}\ \ sx \oplus'' sy \ = \ sx \,\Theta\, \mathsf{map}_r\ (\mathsf{zipwith}\ (\ominus)\ (bottom\ sx))\, sy \\
 & \qquad\qquad\ \ sx \otimes'' sy \ = \ sx \,\Phi\, \mathsf{map}_c\ (\mathsf{zipwith}\ (\gg)\ (right\ sx))\, sy \\
= & \{ \text{ definition of } \mathsf{scan} \ \} \\
 & \mathsf{scan}\ (\ominus, \gg)
\end{array}
$$

□

Optimization Theory of *surrounds*

In this section, we will give a theory for optimization of nested reductions described with *surrounds*.

The main result of optimization on *surrounds* is summarized as follows.

Theorem 5.47 (Surrounding)**.** Let the function *shrink* be defined by homomorphisms as follows.

$$
\begin{array}{l}
shrink \ = \ g_c \times h_n \times h_s \times h_e \times h_w \times h_{ne} \times h_{nw} \times h_{se} \times h_{sw} \\
\quad \mathbf{where} \\
\quad h_n \ = (\![g_n, \oplus_n, \otimes_n]\!)\ , \quad h_s \ = (\![g_s, \oplus_s, \otimes_s]\!)\ , \quad h_e \ = (\![g_e, \oplus_e, \otimes_e]\!) \\
\quad h_w \ = (\![g_w, \oplus_w, \otimes_w]\!)\ , \quad h_{ne} \ = (\![g_{ne}, \oplus_{ne}, \otimes_{ne}]\!)\ , \quad h_{nw} \ = (\![g_{nw}, \oplus_{nw}, \otimes_{nw}]\!) \\
\quad h_{se} \ = (\![g_{se}, \oplus_{se}, \otimes_{se}]\!)\ , \quad h_{sw} \ = (\![g_{sw}, \oplus_{sw}, \otimes_{sw}]\!)
\end{array}
$$

Here, $\oplus_X$ and $\otimes_X$ are extended to satisfy the following equations: $NIL \oplus_X x = x$, $x \oplus_X NIL = x$, $NIL \otimes_X x = x$, and $x \otimes_X NIL = x$.

Then, there exist a projection function *proj* and operators $\oplus'_f$, $\otimes'_f$, $\oplus'_r$, and $\otimes'_r$, and the following equation holds.

$$\begin{aligned}&\mathsf{map}\ f \circ \mathsf{map}\ \mathit{shrink} \circ \mathit{surrounds} = \\ &\qquad \mathsf{map}\ (f \circ \mathit{proj}) \circ \mathsf{scanr}(\oplus'_r, \otimes'_r) \circ \mathsf{map}\ {f_r}' \circ \mathsf{scan}(\oplus'_f, \otimes'_f) \circ \mathsf{map}\ {f_f}'\end{aligned}$$

Here, the complexity of each of the operators $\oplus'_f$, $\otimes'_f$, $\oplus'_r$, and $\otimes'_r$ is bounded by the largest of $\oplus_X$ and $\otimes_X$.

Proof. The theorem is proved by the promotion of map *shrink* with extending the tuples.

The first step is to promote map *shrink* through $\mathsf{scanr}(\oplus_r, \otimes_r)$ since the target program is written as follows:

$$\mathsf{map}\ \mathit{shrink} \circ \mathit{surrounds} = \mathsf{map}\ \mathit{shrink} \circ \mathsf{scanr}(\oplus_r, \otimes_r) \circ \mathsf{map}\ f_r \circ \mathsf{scan}(\oplus_f, \otimes_f) \circ \mathsf{map}\ f_f\ .$$

To do it, we calculate the terms $\mathit{shrink}\ (a \oplus_r b)$ and $\mathit{shrink}\ (a \otimes_r b)$.

$$\begin{aligned}
& \mathit{shrink}\ ((c_a, n_a, s_a, e_a, w_a, ne_a, nw_a, se_a, sw_a) \oplus_r (c_b, n_b, s_b, e_b, w_b, ne_b, nw_b, se_b, sw_b)) \\
= &\quad \{\ \text{Definition of } \oplus_r\ \} \\
& \mathit{shrink}\ (c_a, n_a, s_a \ominus |c_b| \ominus s_b, e_a, w_a, ne_a, nw_a, se_a \ominus e_b \ominus se_b, sw_a \ominus w_b \ominus sw_b) \\
= &\quad \{\ \text{Definition of } \mathit{shrink}\ \} \\
& (g_c\ c_a, h_n\ n_a, h_s\ (s_a \ominus |c_b| \ominus s_b), h_e\ e_a, h_w\ w_a, h_{ne}\ ne_a, h_{nw}\ nw_a, \\
& \qquad h_{se}\ (se_a \ominus e_b \ominus se_b), h_{sw}\ (sw_a \ominus w_b \ominus sw_b)) \\
= &\quad \{\ \text{Definition of homomorphism}\ \} \\
& (g_c\ c_a, h_n\ n_a, h_s\ s_a \oplus_s g_s\ c_b \oplus_s h_s\ s_b, h_e\ e_a, h_w\ w_a, h_{ne}\ ne_a, h_{nw}\ nw_a, \\
& \qquad h_{se}\ se_a \oplus_{se} h_{se}\ e_b \oplus_{se} h_{se}\ se_b, h_{sw}\ sw_a \oplus_{sw} h_{sw}\ w_b \oplus_{sw} h_{sw}\ sw_b)
\end{aligned}$$

$$\begin{aligned}
& \mathit{shrink}\ ((c_a, n_a, s_a, e_a, w_a, ne_a, nw_a, se_a, sw_a) \otimes_r (c_b, n_b, s_b, e_b, w_b, ne_b, nw_b, se_b, sw_b)) \\
= &\quad \{\ \text{Definition of } \otimes_r\ \} \\
& \mathit{shrink}\ (c_a, n_a, s_a, e_a \phi |c_b| \phi e_b, w_a, ne_a \phi n_b \phi ne_b, nw_a, se_a \phi s_b \phi se_b, sw_a) \\
= &\quad \{\ \text{Definition of } \mathit{shrink}\ \} \\
& (g_c\ c_a, h_n\ n_a, h_s\ s_a, h_e\ (e_a \phi |c_b| \phi e_b), h_w\ w_a, \\
& \qquad h_{ne}\ (ne_a \phi n_b \phi ne_b), h_{nw}\ nw_a, h_{se}\ (se_a \phi s_b \phi se_b), h_{sw}\ sw_a) \\
= &\quad \{\ \text{Definition of homomorphism}\ \} \\
& (g_c\ c_a, h_n\ n_a, h_s\ s_a, h_e\ e_a \otimes_e g_e\ c_b \otimes_e h_e\ e_b, h_w\ w_a, \\
& \quad h_{ne}\ ne_a \otimes_{ne} h_{ne}\ n_b \otimes_{ne} h_{ne}\ ne_b, h_{nw}\ nw_a, h_{se}\ se_a \otimes_{se} h_{se}\ s_b \otimes_{se} h_{se}\ se_b, h_{sw}\ sw_a)
\end{aligned}$$

Here, three different functions, namely g_c, g_e and g_s, are applied to c_b and c_a . To make the values $g_e\ c_b$ and $g_s\ c_b$, we have to store always c_b as well as $g_c\ c_b$ that is the result of the first element. Similarly, n, s, e and w have applications of two different functions. Thus,

we extend the tuple to store such extra values duplicating c, n, s, e and w as follows.

$$
\begin{array}{l}
\mathsf{map}\ \ \mathit{shrink} \circ \mathsf{scanr}(\oplus_r, \otimes_r) \circ \mathsf{map}\ \ f_r = \mathit{proj} \circ \mathsf{map}\ \ \mathit{shrink}' \circ \mathsf{scanr}(\oplus'_r, \otimes'_r) \circ \mathsf{map}\ \ {f_r}' \\
\quad \textbf{where} \\
\qquad \mathit{proj}\ (c, n, s, e, w, ne, nw, se, sw, c', n', s', e', w') = (c, n, s, e, w, ne, nw, se, sw) \\
\qquad \mathit{shrink}' = g_c \times h_n \times h_s \times h_e \times h_w \times h_{ne} \times h_{nw} \times h_{se} \times h_{sw} \times id \times h_{ne} \times h_{se} \times h_{se} \times h_{sw} \\
\qquad {f_r}'\ (c, n, w, nw) = (c, n, \mathit{NIL}, \mathit{NIL}, w, \mathit{NIL}, nw, \mathit{NIL}, \mathit{NIL}, c, n, \mathit{NIL}, \mathit{NIL}, w) \\
\qquad (c_a, n_a, s_a, e_a, w_a, ne_a, nw_a, se_a, sw_a, c'_a, n'_a, s'_a, e'_a, w'_a) \\
\qquad\quad \oplus_r (c_b, n_b, s_b, e_b, w_b, ne_b, nw_b, se_b, sw_b, c'_b, n'_b, s'_b, e'_b, w'_b) \\
\qquad\qquad = (c_a, n_a, s_a \ominus |c'_b| \ominus s_b, e_a, w_a, ne_a, nw_a, se_a \ominus e'_b \ominus se_b, sw_a \ominus w'_b \ominus sw_b, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad c'_a, n'_a, s'_a \ominus |c'_b| \ominus s'_b, e'_a, w'_a) \\
\qquad (c_a, n_a, s_a, e_a, w_a, ne_a, nw_a, se_a, sw_a, c'_a, n'_a, s'_a, e'_a, w'_a) \\
\qquad\quad \otimes_r (c_b, n_b, s_b, e_b, w_b, ne_b, nw_b, se_b, sw_b, c'_b, n'_b, s'_b, e'_b, w'_b) \\
\qquad\qquad = (c_a, n_a, s_a, e_a \phi |c'_b| \phi e_b, w_a, ne_a \phi n'_b \phi ne_b, nw_a, se_a \phi s'_b \phi se_b, sw_a, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad c'_a, n'_a, s'_a, e'_a \phi |c'_b| \phi e'_b, w'_a)
\end{array}
$$

Then, we calculate the new terms $\mathit{shrink}'\ (a \oplus'_r b)$ and $\mathit{shrink}'\ (a \otimes'_r b)$, .

$$
\begin{array}{cl}
 & \mathit{shrink}\ ((c_a, n_a, s_a, e_a, w_a, ne_a, nw_a, se_a, sw_a, c'_a, n'_a, s'_a, e'_a, w'_a) \\
 & \qquad\qquad \oplus'_r\ (c_b, n_b, s_b, e_b, w_b, ne_b, nw_b, se_b, sw_b, c'_b, n'_b, s'_b, e'_b, w'_b) \\
= & \quad \{\ \text{Definition of } \oplus'_r,\ \mathit{shrink}' \text{ and homomorphism}\ \} \\
 & (g_c\ c_a, h_n\ n_a, h_s\ s_a \oplus_s g_s\ c'_b \oplus_s h_s\ s_b, h_e\ e_a, h_w\ w_a, h_{ne}\ ne_a, h_{nw}\ nw_a, \\
 & \qquad h_{se}\ se_a \oplus_{se} h_{se}\ e'_b \oplus_{se} h_{se}\ se_b, h_{sw}\ sw_a \oplus_{sw} h_{sw}\ w'_b \oplus_{sw} h_{sw}\ sw_b, \\
 & \qquad\qquad\qquad c'_a, h_{ne}\ n'_a, h_{se}\ s'_a \oplus_{se} g_{se}\ c'_b \oplus_{se} h_{se}\ s'_b, h_e\ e'_a, h_w\ w'_a) \\
= & \quad \{\ \text{Creating a new operator, Definition of } \mathit{shrink}'\ \} \\
 & \mathit{shrink}\ (c_a, n_a, s_a, e_a, w_a, ne_a, nw_a, se_a, sw_a, c'_a, n'_a, s'_a, e'_a, w'_a) \\
 & \quad \oplus''_r \mathit{shrink}\ (c_b, n_b, s_b, e_b, w_b, ne_b, nw_b, se_b, sw_b, c'_b, n'_b, s'_b, e'_b, w'_b) \\
 & \textbf{where} \\
 & (c_a, n_a, s_a, e_a, w_a, ne_a, nw_a, se_a, sw_a, c'_a, n'_a, s'_a, e'_a, w'_a) \\
 & \quad \oplus''_r (c_b, n_b, s_b, e_b, w_b, ne_b, nw_b, se_b, sw_b, c'_b, n'_b, s'_b, e'_b, w'_b) \\
 & \qquad = (c_a, n_a, s_a \oplus_s g_s\ c'_b \oplus_s s_b, e_a, w_a, ne_a, nw_a, \\
 & \qquad\qquad\qquad se_a \oplus_{se} e'_b \oplus_{se} se_b, sw_a \oplus_{sw} w'_b \oplus_{sw} sw_b, \\
 & \qquad\qquad\qquad\qquad\qquad c'_a, n'_a, s'_a \oplus_{se} g_{se}\ c'_b \oplus_{se} s'_b, e'_a, w'_a)
\end{array}
$$

$$
\begin{array}{cl}
 & \mathit{shrink}\ ((c_a, n_a, s_a, e_a, w_a, ne_a, nw_a, se_a, sw_a, c'_a, n'_a, s'_a, e'_a, w'_a) \\
 & \qquad\qquad \otimes'_r\ (c_b, n_b, s_b, e_b, w_b, ne_b, nw_b, se_b, sw_b, c'_b, n'_b, s'_b, e'_b, w'_b) \\
= & \quad \{\ \text{Definition of } \otimes',\ \mathit{shrink}' \text{ and homomorphism}\ \} \\
 & (g_c\ c_a, h_n\ n_a, h_s\ s_a, h_e\ e_a \otimes_e g_e\ c'_b \otimes_e h_e\ e_b, h_w\ w_a, \\
 & \qquad h_{ne}\ ne_a \otimes_{ne} h_{ne}\ n'_b \otimes_{ne} h_{ne}\ ne_b, h_{nw}\ nw_a, h_{se}\ se_a \otimes_{se} h_{se}\ s'_b \otimes_{se} h_{se}\ se_b, h_{sw}\ sw_a, \\
 & \qquad\qquad\qquad\qquad c_a, h_{ne}\ n'_a, h_{se}\ s'_a, h_{se}\ e'_a \otimes_{se} g_{se}\ c'_b \otimes_{se} h_e\ e'_b, h_{sw}\ w'_a) \\
= & \quad \{\ \text{Creating a new operator, Definition of } \mathit{shrink}'\ \} \\
 & \mathit{shrink}\ (c_a, n_a, s_a, e_a, w_a, ne_a, nw_a, se_a, sw_a, c'_a, n'_a, s'_a, e'_a, w'_a) \\
 & \quad \otimes'' \mathit{shrink}\ (c_b, n_b, s_b, e_b, w_b, ne_b, nw_b, se_b, sw_b, c'_b, n'_b, s'_b, e'_b, w'_b) \\
 & \textbf{where} \\
 & (c_a, n_a, s_a, e_a, w_a, ne_a, nw_a, se_a, sw_a, c'_a, n'_a, s'_a, e'_a, w'_a) \\
 & \quad \otimes''_r (c_b, n_b, s_b, e_b, w_b, ne_b, nw_b, se_b, sw_b, c'_b, n'_b, s'_b, e'_b, w'_b) \\
 & \qquad = (c_a, n_a, s_a, e_a \otimes_e g_e\ c'_b \otimes_e e_b, w_a, ne_a \otimes_{ne} n'_b \otimes_{ne} ne_b, nw_a, se_a \otimes_{se} s'_b \otimes_{se} se_b, sw_a, \\
 & \qquad\qquad\qquad\qquad\qquad\qquad c_a, n'_a, s'_a, e'_a \otimes_{se} g_{se}\ c'_b \otimes_{se} e'_b, w'_a)
\end{array}
$$

Using the above operators, we complete the first promotion:

$$\mathsf{map}\ \ shrink' \circ \mathsf{scanr}(\oplus'_r, \otimes'_r) = \mathsf{scanr}(\oplus''_r, \otimes''_r) \circ \mathsf{map}\ \ shrink'\ .$$

The second step is the promotion of $\mathsf{map}\ \ shrink'$ through $\mathsf{map}\ \ f_r{}'$.

$$\begin{array}{ll}
& shrink'(f_r{}'\ (c, n, w, nw)) \\
= & \quad \{\ \text{Definition of } shrink' \text{ and } f_r{}'\ \} \\
& (g_c\ c, h_n\ n, h_s\ NIL, h_e\ NIL, h_w\ w, h_{ne}\ NIL h_{nw}\ nw, h_{se}\ NIL, h_{sw}\ NIL, \\
& \qquad\qquad\qquad\qquad\qquad\qquad c, h_{ne}\ n, h_{se}\ NIL, h_{se}\ NIL, h_{sw}\ w) \\
= & \quad \{\ \text{Application of } h_X \text{ to } NIL \text{ results in } NIL\ \} \\
& (g_c\ c, h_n\ n, NIL, NIL, h_w\ w, NIL h_{nw}\ nw, NIL, NIL, c, h_{ne}\ n, NIL, NIL, h_{sw}\ w)
\end{array}$$

Here, each of the w and n has two applications of different functions. Thus, we extend the tuple used in the first stage calculation of *surrounds* duplicating w and n as follows.

$$\begin{array}{l}
\mathsf{map}\ \ shrink' \circ \mathsf{map}\ \ f_r{}' \circ \mathsf{scan}(\oplus_f, \otimes_f)\ \ \circ \mathsf{map}\ \ f_f \\
\quad = \mathsf{map}\ \ shrink' \circ \mathsf{map}\ \ f_r{}'' \circ \mathsf{scan}(\oplus'_f, \otimes'_f)\ \ \circ \mathsf{map}\ \ f_f{}' \\
\quad \textbf{where} \\
\qquad f_r{}'\ (c, n, w, nw, n', w', n'', w'') = (c, n, NIL, NIL, w, NIL, nw, NIL, NIL, c, n', NIL, NIL, w') \\
\qquad f_f{}'\ a = (a, NIL, NIL, NIL, NIL, NIL, NIL, NIL) \\
\qquad (c_a, n_a, w_a, nw_a, n'_a, w'_a, n''_a, w''_a) \oplus'_f (c_b, n_b, w_b, nw_b, n'_b, w'_b, n''_b, w''_b) \\
\qquad\qquad = (c_b, n_a \ominus |c_a| \ominus n_b, w_b, nw_a \ominus w''_a \ominus nw_b, n'_a \ominus |c_a| \ominus n'_b, w'_b, n''_a \ominus |c_a| \ominus n''_b, w''_b) \\
\qquad (c_a, n_a, w_a, nw_a, n'_a, w'_a, n''_a, w''_a) \otimes'_f (c_b, n_b, w_b, nw_b, n'_b, w'_b, n''_b, w''_b) \\
\qquad\qquad = (c_b, n_b, w_a \phi |c_a| \phi w_b, nw_a \phi n''_a \phi nw_b, n'_b, w'_a \phi |c_a| \phi w'_b, n''_b, w''_a \phi |c_a| \phi w''_b)
\end{array}$$

The extra duplication of w and n is used later. Now, we do the promotion of $\mathsf{map}\ \ shrink'$ through $\mathsf{map}\ \ f_r{}''$ instead of $\mathsf{map}\ \ f_r{}'$.

$$\begin{array}{ll}
& shrink'(f_r{}''\ (c, n, w, nw, n', w', n'', w'')) \\
= & \quad \{\ \text{Definition of } shrink' \text{ and } f_r{}''\ \} \\
& (g_c\ c, h_n\ n, h_s\ NIL, h_e\ NIL, h_w\ w, h_{ne}\ NIL h_{nw}\ nw, h_{se}\ NIL, h_{sw}\ NIL, \\
& \qquad\qquad\qquad\qquad\qquad\qquad c, h_{ne}\ n', h_{se}\ NIL, h_{se}\ NIL, h_{sw}\ w') \\
= & \quad \{\ \text{Application of } h_X \text{ to } NIL \text{ results in } NIL\ \} \\
& (g_c\ c, h_n\ n, NIL, NIL, h_w\ w, NIL h_{nw}\ nw, NIL, NIL, c, h_{ne}\ n', NIL, NIL, h_{sw}\ w') \\
= & \quad \{\ \text{Creating a new function}\ \} \\
& f_r{}'''\ (shrink''(c, n, w, nw, n', w', n'', w'')) \\
& \textbf{where} \\
& \quad f_r{}'''\ (c, n, w, nw, n', w', n'', w'') \\
& \qquad\qquad\qquad = (g_c\ c, n, NIL, NIL, w, NIL nw, NIL, NIL, c, n', NIL, NIL, w') \\
& \quad shrink'' = id \times h_n \times h_w \times h_{nw} \times h_{ne} \times h_{sw} \times h_{nw} \times h_{nw}
\end{array}$$

Thus, $\mathsf{map}\ \ shrink' \circ \mathsf{map}\ \ f_r{}'' = \mathsf{map}\ \ f_r{}''' \circ \mathsf{map}\ \ shrink''$.

The first step of the rest of the promotion is the calculation of $shrink''\ (a \oplus'_f b)$ as

follows.

$$
\begin{array}{ll}
 & \mathit{shrink}'' \; ((c_a, n_a, w_a, nw_a, n'_a, w'_a, n''_a, w''_a) \oplus'_f (c_b, n_b, w_b, nw_b, n'_b, w'_b, n''_b, w''_b)) \\
= & \quad \{ \text{ Definition of } \oplus'_f \text{ and } \mathit{shrink}'' \; \} \\
 & (c_b, h_n \; (n_a \ominus |c_a| \ominus n_b), h_w \; w_b, h_{nw} \; (nw_a \ominus w_a \ominus nw_b), \\
 & \quad h_{ne} \; (n'_a \ominus |c_a| \ominus n'_b), h_{sw} \; w'_b, h_{nw} \; (n''_a \ominus |c_a| \ominus n''_b), h_{nw} \; w''_b) \\
= & \quad \{ \text{ Definition of homomorphism } \} \\
 & (c_b, h_n \; n_a \oplus_n g_n \; c_a \oplus_n h_n \; n_b, h_w \; w_b, h_{nw} \; nw_a \oplus_{nw} h_{nw} \; w''_a \oplus_{nw} h_{nw} \; nw_b, \\
 & \quad h_{ne} \; n'_a \oplus_{ne} g_{ne} \; c_a \oplus_{ne} h_{ne} \; n'_b, h_{sw} \; w'_b, h_{nw} \; n''_a \oplus_{nw} g_{nw} \; c_a \oplus_{nw} h_{nw} \; n''_b, h_{nw} \; w''_b) \\
= & \quad \{ \text{ Creating a new operator } \} \\
 & \mathit{shrink}'' \; (c_a, n_a, w_a, nw_a, n'_a, w'_a, n''_a, w''_a \oplus''_f \mathit{shrink}'' \; (c_b, n_b, w_b, nw_b, n'_b, w'_b, n''_b, w''_b) \\
 & \textbf{where} \\
 & (c_a, n_a, w_a, nw_a, n'_a, w'_a, n''_a, w''_a) \oplus''_f (c_b, n_b, w_b, nw_b, n'_b, w'_b, n''_b, w''_b) \\
 & \quad = (c_b, n_a \oplus_n g_n \; c_a \oplus_n n_b, w_b, nw_a \oplus_{nw} w''_a \oplus_{nw} nw_b, \\
 & \qquad n'_a \oplus_{ne} g_{ne} \; c_a \oplus_{ne} n'_b, w'_b, n''_a \oplus_{nw} g_{nw} \; c_a \oplus_{nw} n''_b, w''_b)
\end{array}
$$

Similarly, the calculation of $\mathit{shrink}'' \; (a \otimes'_f b)$ is as follows.

$$
\begin{array}{ll}
 & \mathit{shrink}'' \; ((c_a, n_a, w_a, nw_a, n'_a, w'_a, n''_a, w''_a) \otimes'_f (c_b, n_b, w_b, nw_b, n'_b, w'_b, n''_b, w''_b)) \\
= & \quad \{ \text{ Definition of } \otimes', \mathit{shrink}'' \text{ and homomorphism } \} \\
 & (c_b, h_n \; n_b, h_w \; w_a \otimes_w g_w \; c_a \otimes_w h_n \; w_b, h_{nw} \; nw_a \otimes_{nw} h_{nw} \; n''_a \otimes_{nw} h_{nw} \; nw_b, \\
 & \quad h_{ne} \; n'_b, h_{sw} \; w'_a \otimes_{sw} g_{sw} \; c_a \otimes_{sw} h_{sw} \; w'_b, h_{nw} \; n''_b, h_{nw} \; w''_a \otimes_{nw} g_{nw} \; c_a \otimes_{nw} h_{nw} \; w''_b) \\
= & \quad \{ \text{ Creating a new operator } \} \\
 & \mathit{shrink}'' \; (c_a, n_a, w_a, nw_a, n'_a, w'_a, n''_a, w''_a \otimes'' \mathit{shrink}'' \; (c_b, n_b, w_b, nw_b, n'_b, w'_b, n''_b, w''_b) \\
 & \textbf{where} \\
 & (c_a, n_a, w_a, nw_a, n'_a, w'_a, n''_a, w''_a) \otimes''_f (c_b, n_b, w_b, nw_b, n'_b, w'_b, n''_b, w''_b) \\
 & \quad = (c_b, n_b, w_a \otimes_w g_w \; c_a \otimes_w w_b, nw_a \otimes_{nw} n''_a \otimes_{nw} nw_b, \\
 & \qquad n'_b, w'_a \otimes_{sw} g_{sw} \; c_a \otimes_{sw} w'_b, n''_b, w''_a \otimes_{nw} g_{nw} \; c_a \otimes_{nw} w''_b)
\end{array}
$$

The final step is fusion of shrink'' and $f_f{}'$.

$$
\begin{array}{ll}
 & \mathit{shrink}''(f_f{}' \; c) \\
= & \quad \{ \text{ Definition of } \mathit{shrink}'' \text{ and } f_f{}' \; \} \\
 & (c, h_n \; \mathit{NIL}, h_w \; \mathit{NIL}, h_{nw} \; \mathit{NIL}, h_{ne} \; \mathit{NIL}, h_{sw} \; \mathit{NIL}, h_{nw} \; \mathit{NIL}, h_{nw} \; \mathit{NIL}) \\
= & \quad \{ \text{ Application of } h_X \text{ to } \mathit{NIL} \text{ results in } \mathit{NIL} \; \} \\
 & (c, \mathit{NIL}, \mathit{NIL}, \mathit{NIL}, \mathit{NIL}, \mathit{NIL}, \mathit{NIL}, \mathit{NIL}) \\
= & \quad \{ \text{ Definition of } f_f{}' \; \} \\
 & f_f{}' \; c
\end{array}
$$

Gathering the above results, the optimization is done as follows.

$$
\begin{array}{l}
\mathsf{map}\ \ \mathit{shrink} \circ \mathit{surrounds} = \mathit{proj} \circ \mathsf{scanr}(\oplus''_r, \otimes''_r) \circ \mathsf{map}\ \ f_r{}''' \circ \mathsf{scan}(\oplus''_f, \otimes''_f) \circ \mathsf{map}\ \ f_f{}' \\
\quad \textbf{where} \\
\quad\quad f_f{}'\ a = (a, NIL, NIL, NIL, NIL, NIL, NIL, NIL) \\
\quad\quad (c_a, n_a, w_a, nw_a, n'_a, w'_a, n''_a, w''_a) \oplus''_f (c_b, n_b, w_b, nw_b, n'_b, w'_b, n''_b, w''_b) \\
\quad\quad\quad = (c_b, n_a \oplus_n g_n\ c_a \oplus_n n_b, w_b, nw_a \oplus_{nw} w''_a \oplus_{nw} nw_b, \\
\quad\quad\quad\quad\quad n'_a \oplus_{ne} g_{ne}\ c_a \oplus_{ne} n'_b, w'_b, n''_a \oplus_{nw} g_{nw}\ c_a \oplus_{nw} n''_b, w''_b) \\
\quad\quad (c_a, n_a, w_a, nw_a, n'_a, w'_a, n''_a, w''_a) \otimes''_f (c_b, n_b, w_b, nw_b, n'_b, w'_b, n''_b, w''_b) \\
\quad\quad\quad = (c_b, n_b, w_a \otimes_w g_w\ c_a \otimes_w w_b, nw_a \otimes_{nw} n''_a \otimes_{nw} nw_b, \\
\quad\quad\quad\quad\quad n'_b, w'_a \otimes_{sw} g_{sw}\ c_a \otimes_{sw} w'_b, n''_b, w''_a \otimes_{nw} g_{nw}\ c_a \otimes_{nw} w''_b) \\
\quad\quad f_r{}'''\ (c, n, w, nw, n', w', n'', w'') \\
\quad\quad\quad = (g_c\ c, n, NIL, NIL, w, NILnw, NIL, NIL, c, n', NIL, NIL, w') \\
\quad\quad (c_a, n_a, s_a, e_a, w_a, ne_a, nw_a, se_a, sw_a, c'_a, n'_a, s'_a, e'_a, w'_a) \\
\quad\quad\quad \oplus''_r (c_b, n_b, s_b, e_b, w_b, ne_b, nw_b, se_b, sw_b, c'_b, n'_b, s'_b, e'_b, w'_b) \\
\quad\quad\quad\quad = (c_a, n_a, s_a, e_a \otimes_e g_e\ c'_b \otimes_e e_b, w_a, \\
\quad\quad\quad\quad\quad ne_a \otimes_{ne} n'_b \otimes_{ne} ne_b, nw_a, se_a \otimes_{se} e'_b \otimes_{se} se_b, sw_a, \\
\quad\quad\quad\quad\quad\quad c_a, n'_a, s'_a, e'_a \otimes_{se} g_{se}\ c'_b \otimes_{se} e'_b, w'_a) \\
\quad\quad (c_a, n_a, s_a, e_a, w_a, ne_a, nw_a, se_a, sw_a, c'_a, n'_a, s'_a, e'_a, w'_a) \\
\quad\quad\quad \otimes''_r (c_b, n_b, s_b, e_b, w_b, ne_b, nw_b, se_b, sw_b, c'_b, n'_b, s'_b, e'_b, w'_b) \\
\quad\quad\quad\quad = (c_a, n_a, s_a, e_a \otimes_e g_e\ c'_b \otimes_e e_b, w_a, \\
\quad\quad\quad\quad\quad ne_a \otimes_{ne} n'_b \otimes_{ne} ne_b, nw_a, se_a \otimes_{se} s'_b \otimes_{se} se_b, sw_a, \\
\quad\quad\quad\quad\quad\quad c_a, n'_a, s'_a, e'_a \otimes_{se} g_{se}\ c'_b \otimes_{se} e'_b, w'_a)
\end{array}
$$

□

The resulting program uses $O(n^2)$ operations for a two-dimensional array of $n \times n$, while the original general form uses $O(n^4)$ operations. The parallel complexity of the resulting program is $O((n^2/P + \sqrt{n^2/P}\log P)T_{(\oplus_X,\otimes_X)})$ for P processors, provided that the computational complexity of $\oplus_X$ and $\otimes_X$ in the homomorphisms are $T_{(\oplus_X,\otimes_X)}$.

All the examples shown in the previous section are described in the left hand side of the theorem. Thus, we can apply this theorem to all of them, and they are executed in $O(n^2/P + \sqrt{n^2/P}\log P)$ complexity using the skeletons.

It is easily seen that the duplication of elements in the tuple during the derivation is not necessary if some of the functions h_X $(X = \{n, s, e, w, ne, nw, se, sw\})$ have the same operators. For example, if h_n and h_{ne} have the same operator, i.e. $h_n = (\![g_{ne}, \oplus_{ne}, \otimes_n]\!)$ and $h_{ne} = (\![g_{ne}, \oplus_{ne}, \otimes_{ne}]\!)$, then we can use $h_n\ n$ instead of $h_{ne}\ n$ because $h_n\ n = h_{ne}\ n$ when $\mathit{width}\ n = 1$. Thus, in this case we can remove the duplication of n for $h_{ne}\ n$ in the tuple. Note that $\otimes_n$ does not need to be the same of $\otimes_{ne}$ because it is not used.

Corollary 5.48 (Reduction of the tuple)**.** If the neighboring functions of h_X $(X \in \{n, s, e, w, ne, nw, se, sw\})$ have the same operators, then, the size of the tuples used in the final program can be reduced. The minimum size is nine.

Finally, we note that we may perform more optimizations by using the shifting of the edges instead of butterfly computations for the global computations of scan and scanr, provided that the operators influence only a fixed number of elements. This leads to the parallel complexity of $O((n^2/P + \sqrt{n^2/P})T_{(\oplus_X,\otimes_X)})$ for P processors.

Optimization Theory of *rects'*

We will show a lemma and a theorem that optimizes nested reductions with *rects'*. The theorem is the generalized result of the derivation in Section 4.3.4.

Lemma 5.49 (Map promotion for *rects'*)**.** For any function f, the following equation holds.

$$\mathsf{map}\ (\mathsf{map}\ f) \circ \mathit{rects}' = \mathit{rects}' \circ \mathsf{map}\ f$$

Proof. This lemma is proved by the following calculation.

$$\begin{array}{cl}
 & \mathsf{map}\ (\mathsf{map}\ f) \circ \mathit{rects}' \\
= & \{\ \text{definition of } \mathit{rects}'\ \} \\
 & \mathsf{map}\ (\mathsf{map}\ f) \circ \mathit{flatten} \circ \mathsf{map}\ (\mathsf{TLs}) \circ \mathsf{BRs} \\
= & \{\ \text{promotion of } \mathsf{map}\ f \text{ through } \mathit{flatten}\ \} \\
 & \mathit{flatten} \circ \mathsf{map}\ (\mathsf{map}\ (\mathsf{map}\ f)) \circ \mathsf{map}\ (\mathsf{TLs}) \circ \mathsf{BRs} \\
= & \{\ \text{map-map fusion}\ \} \\
 & \mathit{flatten} \circ \mathsf{map}\ (\mathsf{map}\ (\mathsf{map}\ f) \circ \mathsf{TLs}) \circ \mathsf{BRs} \\
= & \{\ \text{map promotion through } \mathsf{TLs}\ \} \\
 & \mathit{flatten} \circ \mathsf{map}\ (\mathsf{TLs} \circ \mathsf{map}\ f) \circ \mathsf{BRs} \\
= & \{\ \text{the inverse of map-map fusion, and map promotion through } \mathsf{BRs}\ \} \\
 & \mathit{flatten} \circ \mathsf{map}\ \mathsf{TLs} \circ \mathsf{BRs} \circ \mathsf{map}\ f \\
= & \{\ \text{definition of } \mathit{rects}'\ \} \\
 & \mathit{rects}' \circ \mathsf{map}\ f
\end{array}$$

□

Theorem 5.50. For any associative, commutative binary operators $\oplus$ and $\otimes$ with identities such that $\otimes$ distributes over $\oplus$, the following equation holds.

$$\mathsf{reduce}\ (\oplus,\oplus) \circ \mathsf{map}\ (\mathsf{reduce}\ (\otimes,\otimes)) \circ \mathit{rects}' = \pi_1 \circ (\!\![\ \mathop{\triangle}_1^{11} f_i''', \mathop{\triangle}_1^{11} \odot_i'', \mathop{\triangle}_1^{11} \ominus_i''\ ]\!\!)$$

Here, functions f_i''' and the operators $\odot_i''$ and $\ominus_i''$ are those given by generalizing the operators $\uparrow$ and $+$ in the derivation of Section 4.3.4 to $\oplus$ and $\otimes$.

Proof. The proof is given by generalizing the operators $\uparrow$ and $+$ in the derivation of Section 4.3.4 to $\oplus$ and $\otimes$. □

5.4 Related Work

Optimization of Skeleton Programs Involving Neighbor Elements

One of the simplest fusion optimizations on lists so far uses a general form called cataJ [MKI+04]. This cataJ is actually a list homomorphism that has a function applied to each element of the input list, and an associative binary operator used to perform reduction on elements. Thus, cataJ can describe any computation written as composition of any number of map and at most one reduce (a skeleton to perform reduction) at the last. In this sense, cataJ is a normal form of skeleton programs of such compositions.

Hu et al. [HIT02] proposed a general fusion optimization using a general form called accumulate and a set of fusion rules. The accumulate can describe skeleton scan′, which calculates an accumulation of the input list with an associative binary operator, as well as map and reduce. So, it can be a normal form of skeleton programs described with compositions of these skeletons. Although accumulate can describe also shift$_{\gg}$ and shift$_{\ll}$, it causes some overheads due to lack of consideration of elements on edges. Main overheads are as follows: (1) extensions of elements for uniform manipulation by the associative binary operator, and (2) logarithmic steps of inter-processor communications for general implementation of accumulation. Thus, we need to consider a specific fusion optimization, i.e. a normal form, fusion rules and efficient implementation.

Our proposed normal form extends these fusions optimizations with consideration of elements on edges introduced by shift$_{\gg}$ and shift$_{\ll}$. The normal form separates computation of elements on breaks from that of center part, so that it does not introduce the overheads that accumulate causes.

Grelck et al. [GS06] proposed an optimization based on fusion of a general skeleton called WITH-loop. Since their WITH-loop are based on index accessing, their method can deal with shifting operation such as shift$_{\gg}$ and shift$_{\ll}$. Also, their fusion can handle nesting use of skeletons. However, skeletons that perform accumulation or reduction, in which a region of elements required in computation of an element varies, cannot be described with WITH-loop. Our skeletons can handle such computation owing to the homomorphism-based design.

Generalized scan (gen_scan) proposed by Fischer et al. [FPS01] can perform accumulation that involves neighbor elements. We can describe shifting operations such as shift$_{\gg}$ and shift$_{\ll}$ with their gen_scan. We think program that perform accumulation after combinations of map, zip, shift$_{\gg}$ and shift$_{\ll}$ can be described by gen_scan. However, since the condition for parallel implementation of gen_scan is very complicated, formalization of fusion rules of gen_scan is very difficult. Our skeletons are easier to build fusion rules owing to their uniform design based on homomorphism.

Maximum Marking Problem

Maximum marking problem is one of optimization problems on sequences. The objective of maximum marking problem is to find a marking of the input sequence that gives the maximum weight sum. We can produce many instances of maximum marking problem by changing the rule of marking on the input sequence. For example, maximum prefix sum problem , maximum suffix sum problem, and maximum segment sum problem are instances of maximum marking problem.

Derivation of efficient sequential programs for maximum marking problems was studied by Sasano et al. [SHTO00,SHT01] and Bird [Bir01]. They showed systematic construction of efficient sequential programs when rules of marking are given as recursive functions of a certain class. Since marking of a prefix/suffix/segment is given as a recursive function of the class, an efficient sequential program for maximum prefix/suffix/segment problem is obtained systematically by their results.

Longest Segment Problems

The objective of longest segment problems is to find the longest segment that satisfies a given predicate. Derivation of efficient sequential programs for longest segment problems were studied by Zatema [Zan92], Jeuring [Jeu93] and Zhao [Zha02]. Zatema [Zan92] and Jeuring [Jeu93] showed derivation of efficient sequential programs for various predicates. Zhao [Zha02] studied derivation of efficient sequential programs for predicates constructed as a combination of primitive predicates. She applied it for data mining to make a querying system that supports efficient querying of combined predicates.

Nested Reductions on Two-dimensional Arrays

SKiPPER [SG02] is a skeleton-based parallel programming environment for real-time image processing. It has skeletons specialized for image processing, while we use general skeletons on two-dimensional arrays. Thus, a program developed with SKiPPER may be faster than our skeleton programs in the domain of image processing. Our skeletons deal with wider range of problems, though.

Chapter 6

Implementation of Skeletons and Optimizations

In this chapter, we will show parallel implementations of our designed parallel skeletons for lists and two-dimensional arrays, simple systems for the fusion optimizations, and a library with optimization capabilities for nested reductions.

First, we will show to implement the designed skeletons in parallel for distributed parallel environments. The implementation relies on the parallelism of the homomorphism. We will report some experiment results.

Next, we will explain small systems for fusion optimizations designed in Chapters 4 and 5, and show experiment results of the optimizations. 1 Finally, we will propose a general design of libraries with optimization capabilities based on optimization theorems for domain-specific optimizations, and show implementation of a library for nested reductions in Fortress [ACH+08], of which principle for optimization have been developed in Chapter 5.

6.1 Implementation of Parallel Skeletons on Lists

We briefly view the parallel implementation of the skeletons. In the following, we consider the simplest case: there are n processors and each of them has one element of the list of length n .

The implementation of map and zipwith is quite simple as shown in Figure 6.1. The figure shows that the list $[1, 2, 3, 4, 5, 6, 7, 8]$ is distributively stored in eight processors P0 through P7. Each element of the resulting list can be calculated independently on each processor. Thus, the parallel computational time complexity of map is $O(T_f)$, where T_f is the complexity of the given function. Similarly, the complexity of zipwith is $O(T_f)$.

The calculation of reduce is performed by tree-like computation shown in Figure 6.2. Each stage of the computation is as follows: active processors of the even number positions receive the values from corresponding active processors of the odd number positions, then they apply the operator to the values, and finally they store

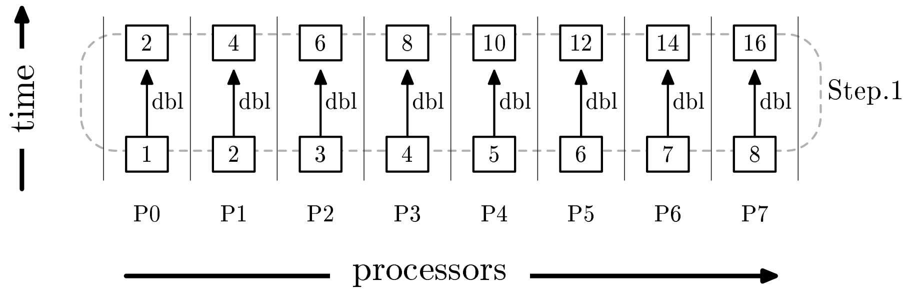


Figure 6.1. Parallel Computation of map *dbl* $[1, 2, 3, 4, 5, 6, 7, 8]$

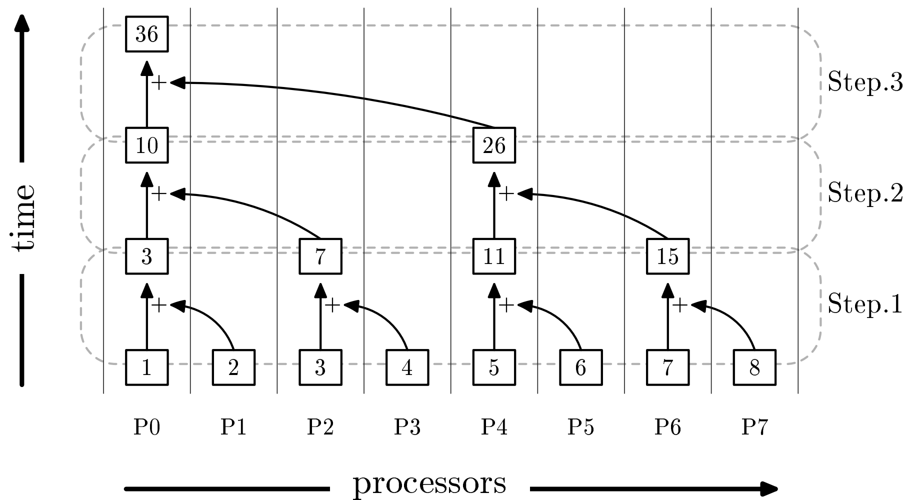


Figure 6.2. Parallel Computation of reduce $(+)$ $[1, 2, 3, 4, 5, 6, 7, 8]$

the results for the nest stage. The processors of the odd number positions become idle after sending their values. One stage of the tree-like computation is done in $O(T_\oplus)$ time in parallel, where $T_\oplus$ is the complexity of the operator. Repeating the stages the communication forms a tree, and its height is $O(\log(n))$. Thus, the parallel computational time complexity of reduce is $O(\log(n)T_\oplus)$.

The implementation of scan is similar to that of reduce. It uses the butterfly computation in stead of the tree-like computation. It is shown in Figure 6.3. Here, each processor stores the pair of the results of applying reduce to the preceding elements and to all elements in the segment considered, respectively. Thus, the parallel computational complexity of scan is $O(\log(n)T_\oplus)$.

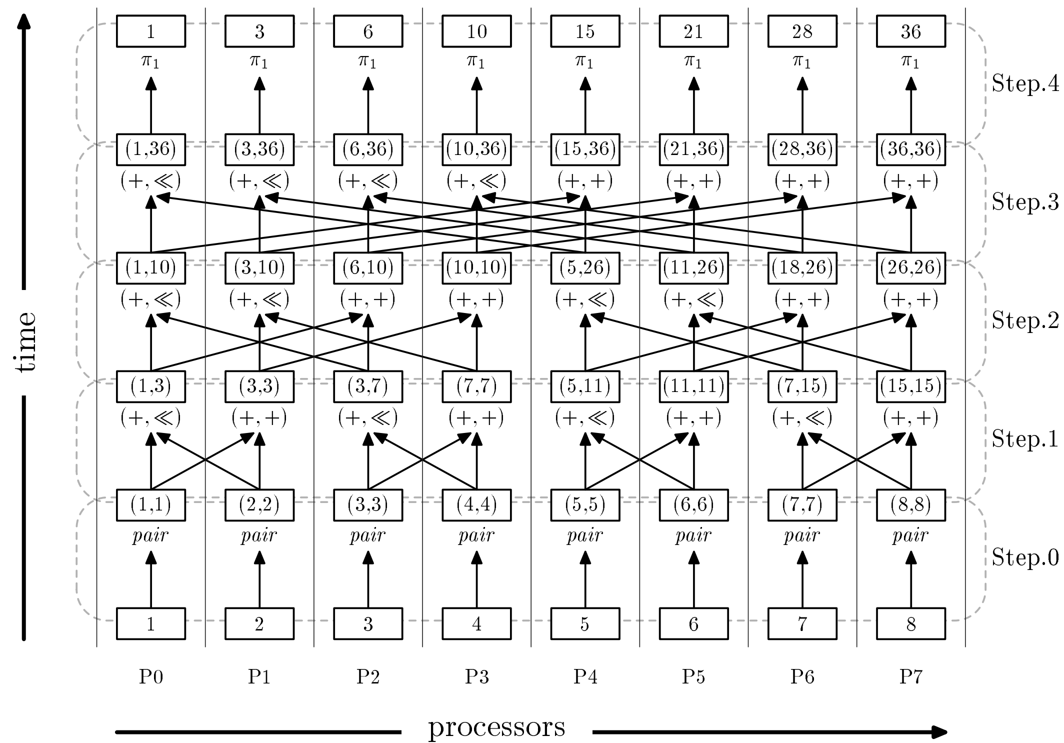


Figure 6.3. Parallel Computation of scan $(+)$ $[1, 2, 3, 4, 5, 6, 7, 8]$

6.2 Implementation of Parallel Skeletons on Two-dimensional Arrays

In this section, we will report efficient parallel implementations of parallel skeletons for two-dimensional data structures on distributed parallel machines, such as PC clusters. We will show some experimental results on the parallel skeletons implemented on C++ with MPI [SO98, GLS99], to show programs described with skeletons can be executed efficiently in parallel. We used MPI to implement lower-level communication so that we can use skeletons on distributed-memory machines. Actually, we have alternatives to implement skeletons. For example, we may use OpenMP [CDK+01, CJvdP07] to provide implementation of skeletons for shared-memory machines.

First, we will show implementations of map, zipwith, and reduce, which have simple implementations. Then, we will give implementation of scan, which has more involved implementation. After that, we will report some experiment results.

6.2.1 Implementation of Simple Parallel Skeletons

The basic parallel skeletons map, zipwith, and reduce can be efficiently implemented on distributed memory systems. To illustrate this, we separate computations of a skeleton into two parts: local computations within a processor and global computations crossing processors. To this end, we will introduce two functions that represent distribution and gathering of arrays.

The function dist abstracts distribution of a two-dimensional array to $p \times q$ processors, and is defined as follows.

$$\begin{array}{l} \mathsf{dist}\ p\ q\ x\ =\ (\mathit{flatten} \circ \mathsf{map}\ (\mathit{grp}_c\ n) \circ \mathit{grp}_r\ m)\ x \\ \quad \textbf{where}\ m = \lceil \mathit{height}\, x/p \rceil,\ n = \lceil \mathit{width}\, x/q \rceil \end{array}$$

The function grp_r divides the given array into sub-arrays of height k, except for the last one. It is defined as follows.

$$\begin{array}{lcll} \mathit{grp}_r\, k\, (x \ominus y) & = & |x| \ominus (\mathit{grp}_r\, k\, y) & \text{while}\ \ \mathit{height}\, x = k \\ \mathit{grp}_r\, k\, x & = & |x| & \text{when}\ \ \mathit{height}\, x \leq k \end{array}$$

Note that our arrays can change its structure so that x of $x \ominus y$ has the height of k when the height of $x \ominus y$ is greater than k. The function grp_c divides the input array similarly in the alternative direction, and is defined similarly.

The distribution $\mathsf{dist}\, p\, q\, x$ means that the two-dimensional array x will be divided into $p \times q$ subarrays (i.e. x is divided into p subarrays in vertical direction, then each subarray is divided into q subarrays in horizontal direction), and each subarray is distributed to each processor.

The function gather, the inverse operator of dist, abstracts gathering of two-dimensional arrays distributed to the processors into a two-dimensional array on the root processor.

$$\mathsf{gather} = \mathsf{reduce}\ \ (\ominus, \phi)$$

Although definitions of these functions may seem complicated, actual operations are simple as illustrated in Figure 6.4. What is significant here is that these functions satisfy the relation of $\mathit{id} = \mathsf{gather} \circ \mathsf{dist}\ p\, q$.

Now, we will proceed to implementations of skeletons. Here, we will separate computations of a skeleton into two parts: local computations within a processor and global computations crossing processors.

For map skeleton, we can separate its computation as follows.

$$\begin{array}{cl} \mathsf{map}\, f & = \quad \{\ \mathit{id} = \mathsf{gather} \circ \mathsf{dist}\ p\, q\ \} \\ & \quad \mathsf{map}\, f \circ \mathsf{gather} \circ \mathsf{dist}\ p\, q \\ & = \quad \{\ \text{Definition of gather}\ \} \\ & \quad \mathsf{map}\, f \circ \mathsf{reduce}\ \ (\ominus, \phi) \circ \mathsf{dist}\ p\, q \\ & = \quad \{\ \text{Promotion of map}\, f\ \ \} \\ & \quad \mathsf{reduce}\ \ (\ominus, \phi) \circ \mathsf{map}\, (\mathsf{map}\, f) \circ \mathsf{dist}\ p\, q \\ & = \quad \{\ \text{Definition of gather}\ \} \\ & \quad \mathsf{gather} \circ \mathsf{map}\, (\mathsf{map}\, f) \circ \mathsf{dist}\ p\, q \end{array}$$

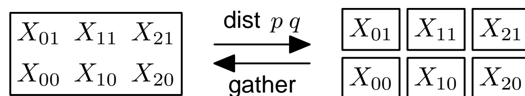


Figure 6.4. An image of distribution function $\mathsf{dist}\ p\ q$ and gathering function gather with $p = 2$ and $q = 3$. Each rectangle corresponds to each processor; X_{ij} represents a subarray.

The last formula indicates that we can compute $\mathsf{map}\ f$ by distributing a two-dimensional array of the argument to the processors by $\mathsf{dist}\ p\ q$, applying $\mathsf{map}\ f$ to each local array independently on each processor, and finally gathering the results onto the root processor by gather. Thus, for a two-dimensional array of $n \times n$ size we can compute $\mathsf{map}\ f$ in $\mathrm{O}(T_f n^2/P)$ parallel time, using $P = pq$ processors and ignoring the distribution and the collection, provided that the function f can be computed in $\mathrm{O}(T_f)$ time.

The implementation of $\mathsf{zipwith}$ is almost the same as map.

For reduce skeleton, we can separate its computation as follows.

$$
\begin{array}{rcl}
\mathsf{reduce}\ (\oplus, \otimes) & = & \quad \{\ id = \mathsf{gather} \circ \mathsf{dist}\ p\ q\ \} \\
 & & \mathsf{reduce}\ (\oplus, \otimes) \circ \mathsf{gather} \circ \mathsf{dist}\, p\, q \\
 & = & \quad \{\ \text{Definition of } \mathsf{gather}\ \} \\
 & & \mathsf{reduce}\ (\ominus, \phi) \circ \mathsf{dist}\, p\, q \\
 & = & \quad \{\ \text{Promotion of } \mathsf{reduce}\ (\oplus, \otimes)\ \} \\
 & & \mathsf{reduce}\ (\oplus, \otimes) \circ \mathsf{map}\, (\mathsf{reduce}\ (\oplus, \otimes)) \circ \mathsf{dist}\, p\, q
\end{array}
$$

The last formula indicates that we can compute $\mathsf{reduce}\ (\oplus, \otimes)$ by distributing a two-dimensional array of the argument to the processors by $\mathsf{dist}\ p\ q$, applying $\mathsf{reduce}\ (\oplus, \otimes)$ to each local array independently on each processor, and finally reducing the results into the root processor by $\mathsf{reduce}\ (\oplus, \otimes)$ described in the last formula. From the property of Eq. (3.2), the last reduction over the results of all processors can be computed by using tree-like computation in column and row directions respectively like parallel computation of reduction on one-dimensional lists (see Chapter 2). Thus, for a two-dimensional array of $n \times n$ size we can compute $\mathsf{reduce}\ (\oplus, \otimes)$ in $\mathrm{O}(T_{\oplus,\otimes}(n^2/P + \log P))$ parallel time, using $P = pq$ processors and ignoring the distribution, provided that the binary operators $\oplus$ and $\otimes$ can be computed in $\mathrm{O}(T_{\oplus,\otimes})$ time. Note that we can also execute the tree-like computation in both of column and row directions simultaneously, which results in the same cost.

6.2.2 Implementation of Scan Skeleton

In this section, we will show the parallel implementation of scan, as well as the implementation of $\mathsf{accumulate}$. We will first give the implementation of scan for

the simple case that each processor has one element of the two-dimensional array. Then, we will give the general implementation of scan, in which each processor has a subarray of the input rather than a single element. We will first show the general implementation of accumulate using the simple case implementation of scan, and then instantiate it for scan.

To formalize the simple case implementation of scan, we first extend the distributable homomorphism [Gor96] on lists to that on two-dimensional arrays. The distributable homomorphism on lists is a special case of homomorphism, and can be used as the basis of parallel implementation of scan on lists. The distributable homomorphism on two-dimensional arrays is also a special case of the homomorphism, as defined below.

$$\begin{array}{lcl}
\mathsf{DH}_2\ (\boxplus, \boxplus', \boxtimes, \boxtimes')\ (x \mathbin{⦵} y) & = & \textbf{let}\ x' = \mathsf{DH}_2\ (\boxplus, \boxplus', \boxtimes, \boxtimes')\ x \\
 & & \quad\ \ y' = \mathsf{DH}_2\ (\boxplus, \boxplus', \boxtimes, \boxtimes')\ y \\
 & & \textbf{in}\ \mathsf{zipwith}\ (\boxplus)\ x'\ y' \mathbin{⦵} \mathsf{zipwith}\ (\boxplus')\ x'\ y' \\
\mathsf{DH}_2\ (\boxplus, \boxplus', \boxtimes, \boxtimes')\ (x \mathbin{⦶} y) & = & \textbf{let}\ x' = \mathsf{DH}_2\ (\boxplus, \boxplus', \boxtimes, \boxtimes')\ x \\
 & & \quad\ \ y' = \mathsf{DH}_2\ (\boxplus, \boxplus', \boxtimes, \boxtimes')\ y \\
 & & \textbf{in}\ \mathsf{zipwith}\ (\boxtimes)\ x'\ y' \mathbin{⦶} \mathsf{zipwith}\ (\boxtimes')\ x'\ y' \\
\mathsf{DH}_2\ (\boxplus, \boxplus', \boxtimes, \boxtimes')\ |a| & = & |a|
\end{array}$$

It is easily seen that the distributable homomorphism $\mathsf{DH}_2\ (\boxplus, \boxplus', \boxtimes, \boxtimes')$ is executed in parallel by the two-dimensional version of the butterfly computation as shown in Figure 6.5. It is also clear that its computational complexity is $\mathrm{O}(T \log(n))$ for an $n \times n$ array, where T is the complexity of the operators.

Then, we give the theorem that shows the skeleton scan can be described in terms of the distributable homomorphism; the skeleton scan can be implemented using the butterfly computation of the distributable homomorphism.

Theorem 6.1 (Implementation of scan by distributable homomorphism)**.** The skeleton scan is described in terms of distributable homomorphism as follows.

$$\begin{array}{l}
\mathsf{scan}\ (\oplus, \otimes) = \mathsf{map}\ \pi_1 \circ \mathsf{DH}_2\ (\boxplus, \boxplus', \boxtimes, \boxtimes') \circ \mathsf{map}\ p_4 \\
\quad \textbf{where} \\
\qquad (nw_a, n_a, w_a, c_a) \boxplus (nw_b, n_b, w_b, c_b) = (nw_a, n_a, w_a \oplus w_b, c_a \oplus c_b) \\
\qquad (nw_a, n_a, w_a, c_a) \boxplus' (nw_b, n_b, w_b, c_b) = (w_a \oplus nw_b, c_a \oplus n_b, w_a \oplus w_b, c_a \oplus c_b) \\
\qquad (nw_a, n_a, w_a, c_a) \boxtimes (nw_b, n_b, w_b, c_b) = (nw_a, n_a \otimes n_b, w_a, c_a \otimes c_b) \\
\qquad (nw_a, n_a, w_a, c_a) \boxtimes' (nw_b, n_b, w_b, c_b) = (n_a \otimes nw_b, n_a \otimes n_b, c_a \otimes w_b, c_a \otimes c_b) \\
\qquad p_4\ a = (a, a, a, a)
\end{array}$$

Proof. The theorem is proved by induction on the structure of two-dimensional arrays. The base case is proved as follows.

$$\begin{array}{cl}
 & \mathsf{scan}\ (\oplus, \otimes)\ |a| \\
= & \quad \{\ \text{Definition of } \mathsf{scan}\ \} \\
 & |a| \\
= & \quad \{\ \text{Definition of } \pi_1,\ p_4,\ \mathsf{map} \text{ and } \mathsf{DH}_2\ \} \\
 & (\mathsf{map}\ \pi_1 \circ \mathsf{DH}_2\ (\boxplus, \boxplus', \boxtimes, \boxtimes') \circ \mathsf{map}\ p_4)\ |a|
\end{array}$$

The inductive case for $⦵$ is proved as follows.

$$
\begin{array}{ll}
 & \mathsf{scan}\ (\oplus,\otimes)\ (x_1 ⦵ x_2) \\
= & \quad \{\ \text{Definition of } \mathsf{scan}\ \} \\
 & \mathsf{scan}\,(\oplus,\otimes)\,x_1 ⦵ \mathsf{map}_r\,(\mathsf{zipwith}\,(\oplus)\,(\mathit{bottom}\,(\mathsf{scan}\,(\oplus,\otimes)\,x_1)))\,(\mathsf{scan}\,(\oplus,\otimes)\,x_2) \\
= & \quad \{\ \text{Hypothesis of induction}\ \} \\
 & (\mathsf{map}\ \pi_1 \circ \mathsf{DH}_2\ (\boxplus,\boxplus',\boxtimes,\boxtimes') \circ \mathsf{map}\ p_4)\ x_1 \\
 & \quad ⦵ \mathsf{map}_r\ (\mathsf{zipwith}\ (\oplus)\ (\mathit{bottom}\ (\mathsf{scan}\ (\oplus,\otimes)\ x_1))) \\
 & \qquad ((\mathsf{map}\ \pi_1 \circ \mathsf{DH}_2\ (\boxplus,\boxplus',\boxtimes,\boxtimes') \circ \mathsf{map}\ p_4)\ x_2) \\
= & \quad \{\ \text{The property shown below, and function composition}\ \} \\
 & \mathsf{map}\ \pi_1\ (\mathsf{DH}_2\ (\boxplus,\boxplus',\boxtimes,\boxtimes')\ (\mathsf{map}\ p_4\ x_1)) \\
 & \quad ⦵ \mathsf{zipwith}\ (\oplus)\ (\mathsf{map}\ \pi_3\ (\mathsf{DH}_2\ (\boxplus,\boxplus',\boxtimes,\boxtimes')\ (\mathsf{map}\ p_4\ x_1))) \\
 & \qquad (\mathsf{map}\ \pi_1\ (\mathsf{DH}_2\ (\boxplus,\boxplus',\boxtimes,\boxtimes')\ (\mathsf{map}\ p_4\ x_2))) \\
= & \quad \{\ \text{Extracting the same terms using `}\mathbf{where}\text{'}\ \} \\
 & \mathsf{map}\ \pi_1\ x_1' ⦵ \mathsf{zipwith}\ (\oplus)\ (\mathsf{map}\ \pi_3\ x_1')\ (\mathsf{map}\ \pi_1\ x_2') \\
 & \qquad \mathbf{where}\ x_1' = \mathsf{DH}_2\ (\boxplus,\boxplus',\boxtimes,\boxtimes')\ (\mathsf{map}\ p_4\ x_1) \\
 & \qquad\qquad\quad\ x_2' = \mathsf{DH}_2\ (\boxplus,\boxplus',\boxtimes,\boxtimes')\ (\mathsf{map}\ p_4\ x_2) \\
= & \quad \left\{ \begin{array}{l} \pi_1\ ((nw_a,n_a,w_a,c_a) \boxplus' (nw_b,n_b,w_b,c_b)) \\ \quad = w_a \oplus nw_b = \pi_3\ (nw_a,n_a,w_a,c_a) \oplus \pi_1\ (nw_b,n_b,w_b,c_b) \end{array} \right\} \\
 & \mathsf{map}\ \pi_1\ x_1' ⦵ \mathsf{map}\ \pi_1\ (\mathsf{zipwith}\ (\boxplus')\ x_1'\ x_2') \\
 & \qquad \mathbf{where}\ x_1' = \mathsf{DH}_2\ (\boxplus,\boxplus',\boxtimes,\boxtimes')\ (\mathsf{map}\ p_4\ x_1) \\
 & \qquad\qquad\quad\ x_2' = \mathsf{DH}_2\ (\boxplus,\boxplus',\boxtimes,\boxtimes')\ (\mathsf{map}\ p_4\ x_2) \\
= & \quad \left\{ \begin{array}{l} \pi_1\ ((nw_a,n_a,w_a,c_a) \boxplus (nw_b,n_b,w_b,c_b)) \\ \quad = nw_a = \pi_1\ (nw_a,n_a,w_a,c_a) \end{array} \right\} \\
 & \mathsf{map}\ \pi_1\ (\mathsf{zipwith}\ (\boxplus)\ x_1'\ x_2') ⦵ \mathsf{map}\ \pi_1\ (\mathsf{zipwith}\ (\boxplus')\ x_1'\ x_2') \\
 & \qquad \mathbf{where}\ x_1' = \mathsf{DH}_2\ (\boxplus,\boxplus',\boxtimes,\boxtimes')\ (\mathsf{map}\ p_4\ x_1) \\
 & \qquad\qquad\quad\ x_2' = \mathsf{DH}_2\ (\boxplus,\boxplus',\boxtimes,\boxtimes')\ (\mathsf{map}\ p_4\ x_2) \\
= & \quad \{\ \text{Definition of } \mathsf{map}\ \} \\
 & \mathsf{map}\ \pi_1\ (\mathsf{zipwith}\ (\boxplus)\ x_1'\ x_2' ⦵ \mathsf{zipwith}\ (\boxplus')\ x_1'\ x_2') \\
 & \qquad \mathbf{where}\ x_1' = \mathsf{DH}_2\ (\boxplus,\boxplus',\boxtimes,\boxtimes')\ (\mathsf{map}\ p_4\ x_1) \\
 & \qquad\qquad\quad\ x_2' = \mathsf{DH}_2\ (\boxplus,\boxplus',\boxtimes,\boxtimes')\ (\mathsf{map}\ p_4\ x_2) \\
= & \quad \{\ \text{Definition of } \mathsf{DH}_2 \text{ and } \mathsf{map}\ \} \\
 & \mathsf{map}\ \pi_1\ (\mathsf{DH}_2\ (\boxplus,\boxplus',\boxtimes,\boxtimes')\ (\mathsf{map}\ p_4\ (x_1 ⦵ y_1)))
\end{array}
$$

To complete the proof, we have to show the following property.

$$
\begin{array}{l}
\mathsf{map}_r\ (\mathsf{zipwith}\ (\oplus)\ (\mathit{bottom}\ (\mathsf{scan}\ (\oplus,\otimes)\ x))) \\
= \mathsf{zipwith}\ (\oplus)\ (\mathsf{map}\ \pi_3\ (\mathsf{DH}_2\ (\boxplus,\boxplus',\boxtimes,\boxtimes')\ (\mathsf{map}\ p_4\ x)))
\end{array}
$$

It is proved by three properties: (1) if all rows of z are the same, then the equation

$$\mathsf{zipwith}\ (\oplus)\ z\ y = \mathsf{map}_r\ (\mathsf{zipwith}\ (\mathit{bottom}\ z))\ y$$

holds, (2) all rows of $\mathsf{map}\ \pi_3\ (\mathsf{DH}_2\ (\boxplus,\boxplus',\boxtimes,\boxtimes')\ y)$ are the same, and (3) the following equation holds.

$$\mathit{bottom}\ (\mathsf{scan}\ (\oplus,\otimes)\ x) = \mathit{bottom}\ (\mathsf{map}\ \pi_3\ (\mathsf{DH}_2\ (\boxplus,\boxplus',\boxtimes,\boxtimes')\ (\mathsf{map}\ p_4\ x)))$$

The first one is obvious since each row of $\mathsf{zipwith}\ (\oplus)\ z\ y$ is equal to the result of application of $\mathsf{zipwith}\ (\oplus)\ (bottom\ z)$ to the corresponding row of y.

The second one is proved by induction as follows. The base case is obvious. The inductive case for $\ominus$ is as follows.

$$
\begin{array}{ll}
 & \mathsf{map}\ \pi_3\ (\mathsf{DH}_2\ (\boxplus, \boxplus', \boxtimes, \boxtimes')\ (y_1 \ominus y_2)) \\
= & \quad \{\ \text{Definition of } \mathsf{DH}_2 \text{ and } \mathsf{map}\ \} \\
 & \mathsf{map}\ \pi_3\ (\mathsf{zipwith}\ (\boxplus)\ y'_1\ y'_2) \ominus \mathsf{map}\ \pi_3\ (\mathsf{zipwith}\ (\boxplus')\ y'_1\ y'_2) \\
 & \quad\quad \textbf{where}\ y'_1 = \mathsf{DH}_2\ (\boxplus, \boxplus', \boxtimes, \boxtimes')\ y_1 \\
 & \quad\quad\quad\quad\quad\ \ y'_2 = \mathsf{DH}_2\ (\boxplus, \boxplus', \boxtimes, \boxtimes')\ y_2 \\
= & \quad \{\ \text{Definition of } \boxplus \text{ and } \boxplus'\text{, promotion of } \mathsf{map}\ \} \\
 & \mathsf{zipwith}\ (\oplus)\ (\mathsf{map}\ \pi_3\ y'_1)\ (\mathsf{map}\ \pi_3\ y'_2) \ominus \mathsf{zipwith}\ (\oplus)\ (\mathsf{map}\ \pi_3\ y'_1)\ (\mathsf{map}\ \pi_3\ y'_2) \\
 & \quad\quad \textbf{where}\ y'_1 = \mathsf{DH}_2\ (\boxplus, \boxplus', \boxtimes, \boxtimes')\ y_1 \\
 & \quad\quad\quad\quad\quad\ \ y'_2 = \mathsf{DH}_2\ (\boxplus, \boxplus', \boxtimes, \boxtimes')\ y_2
\end{array}
$$

By hypothesis of induction, all rows of the final equation are the same. The inductive case for ϕ is as follows.

$$
\begin{array}{ll}
 & \mathsf{map}\ \pi_3\ (\mathsf{DH}_2\ (\boxplus, \boxplus', \boxtimes, \boxtimes')\ (y_1 \phi y_2)) \\
= & \quad \{\ \text{Definition of } \mathsf{DH}_2 \text{ and } \mathsf{map}\ \} \\
 & \mathsf{map}\ \pi_3\ (\mathsf{zipwith}\ (\boxtimes)\ y'_1\ y'_2) \phi \mathsf{map}\ \pi_3\ (\mathsf{zipwith}\ (\boxtimes')\ y'_1\ y'_2) \\
 & \quad\quad \textbf{where}\ y'_1 = \mathsf{DH}_2\ (\boxplus, \boxplus', \boxtimes, \boxtimes')\ y_1 \\
 & \quad\quad\quad\quad\quad\ \ y'_2 = \mathsf{DH}_2\ (\boxplus, \boxplus', \boxtimes, \boxtimes')\ y_2 \\
= & \quad \{\ \text{Definition of } \boxtimes \text{ and } \boxtimes'\text{, promotion of } \mathsf{map}\ \} \\
 & \mathsf{map}\ \pi_3\ y'_1 \phi \mathsf{zipwith}\ (\otimes)\ (\mathsf{map}\ \pi_3\ y'_1)\ (\mathsf{map}\ \pi_3\ y'_2) \\
 & \quad\quad \textbf{where}\ y'_1 = \mathsf{DH}_2\ (\boxplus, \boxplus', \boxtimes, \boxtimes')\ y_1 \\
 & \quad\quad\quad\quad\quad\ \ y'_2 = \mathsf{DH}_2\ (\boxplus, \boxplus', \boxtimes, \boxtimes')\ y_2
\end{array}
$$

By hypothesis of induction, all rows of the final equation are the same. Similarly, we can prove that all columns of $\mathsf{map}\ \pi_2\ (\mathsf{DH}_2\ (\boxplus, \boxplus', \boxtimes, \boxtimes')\ y)$ are the same, and that all elements of $\mathsf{map}\ \pi_4\ (\mathsf{DH}_2\ (\boxplus, \boxplus', \boxtimes, \boxtimes')\ y)$ are the same.

The third property is proved by induction. The base case is obvious. The inductive case for $\ominus$ as follows.

$$
\begin{array}{ll}
 & bottom\ (\mathsf{scan}\ (\oplus, \otimes)\ (x_1 \ominus x_2)) \\
= & \quad \{\ \text{Definition of } \mathsf{scan} \text{ and } bottom\ \} \\
 & bottom\ (\mathsf{map}_r\ (\mathsf{zipwith}\ (\oplus)\ (bottom\ (\mathsf{scan}\ (\oplus, \otimes)\ x_1)))\ (\mathsf{scan}\ (\oplus, \otimes)\ x_2)) \\
= & \quad \{\ \text{Promotion of } bottom \text{ through } \mathsf{map}_r\ \} \\
 & \mathsf{map}_r\ (\mathsf{zipwith}\ (\oplus)\ (bottom\ (\mathsf{scan}\ (\oplus, \otimes)\ x_1)))\ (bottom\ (\mathsf{scan}\ (\oplus, \otimes)\ x_2)) \\
= & \quad \{\ \text{Application of } \mathsf{map}_r\ f \text{ to a row vector is application of } f\ \} \\
 & \mathsf{zipwith}\ (\oplus)\ (bottom\ (\mathsf{scan}\ (\oplus, \otimes)\ x_1))\ (bottom\ (\mathsf{scan}\ (\oplus, \otimes)\ x_2)
\end{array}
$$

$$
\begin{array}{ll}
= & \{ \text{ Hypothesis of induction } \} \\
& \mathsf{zipwith}\ (\oplus)\ (\mathit{bottom}\ (\mathsf{map}\ \pi_3\ x_1'))\ (\mathit{bottom}\ (\mathsf{map}\ \pi_3\ x_2')) \\
& \quad \textbf{where}\ x_1' = \mathsf{DH}_2\ (\boxplus, \boxplus', \boxtimes, \boxtimes')\ (\mathsf{map}\ p_4\ x_1) \\
& \quad \phantom{\textbf{where}\ } x_2' = \mathsf{DH}_2\ (\boxplus, \boxplus', \boxtimes, \boxtimes')\ (\mathsf{map}\ p_4\ x_2) \\
= & \{ \text{ Promotion of } \mathsf{zipwith} \ \} \\
& \mathit{bottom}\ (\mathsf{zipwith}\ (\oplus)\ (\mathsf{map}\ \pi_3\ x_1')\ (\mathsf{map}\ \pi_3\ x_2')) \\
& \quad \textbf{where}\ x_1' = \mathsf{DH}_2\ (\boxplus, \boxplus', \boxtimes, \boxtimes')\ (\mathsf{map}\ p_4\ x_1) \\
& \quad \phantom{\textbf{where}\ } x_2' = \mathsf{DH}_2\ (\boxplus, \boxplus', \boxtimes, \boxtimes')\ (\mathsf{map}\ p_4\ x_2) \\
= & \{ \text{ Definition of } \boxplus', \text{ and promotion of } \mathsf{zipwith} \ \} \\
& \mathit{bottom}\ (\mathsf{map}\ \pi_3\ (\mathsf{zipwith}\ (\boxplus')\ x_1'\ x_2')) \\
& \quad \textbf{where}\ x_1' = \mathsf{DH}_2\ (\boxplus, \boxplus', \boxtimes, \boxtimes')\ (\mathsf{map}\ p_4\ x_1) \\
& \quad \phantom{\textbf{where}\ } x_2' = \mathsf{DH}_2\ (\boxplus, \boxplus', \boxtimes, \boxtimes')\ (\mathsf{map}\ p_4\ x_2) \\
= & \{ \text{ Definition of } \mathit{bottom} \ \} \\
& \mathit{bottom}\ (\mathsf{map}\ \pi_3\ (\mathsf{zipwith}\ (\boxplus)\ x_1'\ x_2') \ominus \mathsf{map}\ \pi_3\ (\mathsf{zipwith}\ (\boxplus')\ x_1'\ x_2')) \\
& \quad \textbf{where}\ x_1' = \mathsf{DH}_2\ (\boxplus, \boxplus', \boxtimes, \boxtimes')\ (\mathsf{map}\ p_4\ x_1) \\
& \quad \phantom{\textbf{where}\ } x_2' = \mathsf{DH}_2\ (\boxplus, \boxplus', \boxtimes, \boxtimes')\ (\mathsf{map}\ p_4\ x_2) \\
= & \{ \text{ Definition of } \mathsf{DH}_2 \text{ and } \mathsf{map} \ \} \\
& \mathit{bottom}\ (\mathsf{map}\ \pi_3\ (\mathsf{DH}_2\ (\boxplus, \boxplus', \boxtimes, \boxtimes')\ (\mathsf{map}\ p_4\ (x_1 \ominus x_2))))
\end{array}
$$

The inductive case for ϕ as follows.

$$
\begin{array}{ll}
& \mathit{bottom}\ (\mathsf{scan}\ (\oplus, \otimes)\ (x_1 \phi x_2)) \\
= & \{ \text{ Definition of } \mathsf{scan}, \text{ distribution of } \mathit{bottom} \ \} \\
& \mathit{bottom}\ (\mathsf{scan}\ (\oplus, \otimes)\ x_1) \phi \\
& \quad \mathsf{map}_c\ (\mathsf{zipwith}\ (\oplus)\ (\mathit{bottom}\ (\mathit{right}\ (\mathsf{scan}\ (\oplus, \otimes)\ x_1))))\ (\mathit{bottom}\ (\mathsf{scan}\ (\oplus, \otimes)\ x_2)) \\
= & \{ \text{ Hypothesis of induction, property of the fourth element } \} \\
& \mathit{bottom}\ (\mathsf{map}\ \pi_3\ x_1') \phi \\
& \quad \mathsf{map}_c\ (\mathsf{zipwith}\ (\oplus)\ (\mathit{bottom}\ (\mathit{right}\ (\mathsf{map}\ \pi_4\ x_1'))))\ (\mathit{bottom}\ (\mathsf{map}\ \pi_3\ x_2')) \\
& \quad \textbf{where}\ x_1' = \mathsf{DH}_2\ (\boxplus, \boxplus', \boxtimes, \boxtimes')\ (\mathsf{map}\ p_4\ x_1) \\
& \quad \phantom{\textbf{where}\ } x_2' = \mathsf{DH}_2\ (\boxplus, \boxplus', \boxtimes, \boxtimes')\ (\mathsf{map}\ p_4\ x_2) \\
= & \{ \ \mathit{bottom} \circ \mathit{right} = \mathit{right} \circ \mathit{bottom} \ \} \\
& \mathit{bottom}\ (\mathsf{map}\ \pi_3\ x_1') \phi \\
& \quad \mathsf{map}_c\ (\mathsf{zipwith}\ (\oplus)\ (\mathit{right}\ (\mathit{bottom}\ (\mathsf{map}\ \pi_4\ x_1'))))\ (\mathit{bottom}\ (\mathsf{map}\ \pi_3\ x_2')) \\
& \quad \textbf{where}\ x_1' = \mathsf{DH}_2\ (\boxplus, \boxplus', \boxtimes, \boxtimes')\ (\mathsf{map}\ p_4\ x_1) \\
& \quad \phantom{\textbf{where}\ } x_2' = \mathsf{DH}_2\ (\boxplus, \boxplus', \boxtimes, \boxtimes')\ (\mathsf{map}\ p_4\ x_2) \\
= & \{ \text{ All elements of } \mathsf{map}\ \pi_4\ x_1' \text{ are the same } \} \\
& \mathit{bottom}\ (\mathsf{map}\ \pi_3\ x_1') \phi \mathsf{zipwith}\ (\oplus)\ (\mathit{bottom}\ (\mathsf{map}\ \pi_4\ x_1'))\ (\mathit{bottom}\ (\mathsf{map}\ \pi_3\ x_2')) \\
& \quad \textbf{where}\ x_1' = \mathsf{DH}_2\ (\boxplus, \boxplus', \boxtimes, \boxtimes')\ (\mathsf{map}\ p_4\ x_1) \\
& \quad \phantom{\textbf{where}\ } x_2' = \mathsf{DH}_2\ (\boxplus, \boxplus', \boxtimes, \boxtimes')\ (\mathsf{map}\ p_4\ x_2) \\
= & \{ \text{ Promotion of } \mathsf{zipwith} \ \} \\
& \mathit{bottom}\ (\mathsf{map}\ \pi_3\ x_1') \phi \mathit{bottom}\ (\mathsf{zipwith}\ (\oplus)\ (\mathsf{map}\ \pi_4\ x_1')\ (\mathsf{map}\ \pi_3\ x_2')) \\
& \quad \textbf{where}\ x_1' = \mathsf{DH}_2\ (\boxplus, \boxplus', \boxtimes, \boxtimes')\ (\mathsf{map}\ p_4\ x_1) \\
& \quad \phantom{\textbf{where}\ } x_2' = \mathsf{DH}_2\ (\boxplus, \boxplus', \boxtimes, \boxtimes')\ (\mathsf{map}\ p_4\ x_2)
\end{array}
$$

$$
\begin{array}{ll}
= & \{ \text{ Distributivity of } \mathit{bottom} \text{ over } \phi \ \} \\
 & \mathit{bottom}\ (\mathsf{map}\ \pi_3\ x'_1 \ \phi\ \mathsf{zipwith}\ (\oplus)\ (\mathsf{map}\ \pi_4\ x'_1)\ (\mathsf{map}\ \pi_3\ x'_2)) \\
 & \qquad \textbf{where } x'_1 = \mathsf{DH_2}\ (\boxplus, \boxplus', \boxtimes, \boxtimes')\ (\mathsf{map}\ p_4\ x_1) \\
 & \qquad \qquad \quad\ x'_2 = \mathsf{DH_2}\ (\boxplus, \boxplus', \boxtimes, \boxtimes')\ (\mathsf{map}\ p_4\ x_2) \\
= & \{ \text{ Definition of } \boxtimes \text{ and } \boxtimes' \ \} \\
 & \mathit{bottom}\ (\mathsf{map}\ \pi_3\ (\mathsf{zipwith}\ (\boxtimes)\ x'_1 x'_2 \ \phi\ \mathsf{zipwith}\ (\boxtimes')\ x'_1 x'_2)) \\
 & \qquad \textbf{where } x'_1 = \mathsf{DH_2}\ (\boxplus, \boxplus', \boxtimes, \boxtimes')\ (\mathsf{map}\ p_4\ x_1) \\
 & \qquad \qquad \quad\ x'_2 = \mathsf{DH_2}\ (\boxplus, \boxplus', \boxtimes, \boxtimes')\ (\mathsf{map}\ p_4\ x_2) \\
= & \{ \text{ Definition of } \mathsf{DH_2} \text{ and } \mathsf{map} \ \} \\
 & \mathit{bottom}\ (\mathsf{map}\ \pi_3\ (\mathsf{DH_2}\ (\boxplus, \boxplus', \boxtimes, \boxtimes')\ (\mathsf{map}\ p_4\ (x_1 \ \phi\ x_2))
\end{array}
$$

The property of the fourth component is as follows.

$$
\begin{array}{l}
\mathit{bottom}\ (\mathit{right}\ (\mathsf{scan}\ (\oplus, \otimes)\ x)) \\
\quad = \mathit{bottom}\ (\mathit{right}\ (\mathsf{map}\ \pi_4\ (\mathsf{DH_2}\ (\boxplus, \boxplus', \boxtimes, \boxtimes')\ (\mathsf{map}\ p_4\ x))))
\end{array}
$$

This property is obvious because the bottom-right element of scan $(\oplus, \otimes)\ x$ is the result of reduce $(\oplus, \otimes)\ x$, and the fourth component of each element in the result of $\mathsf{DH_2}\ (\boxplus, \boxplus', \boxtimes, \boxtimes')\ (\mathsf{map}\ p_4\ x)$ is also the result of reduce $(\oplus, \otimes)\ x$.

The inductive case for ϕ of the theorem is proved similarly. □

Since the complexity of distributable homomorphism is $\mathrm{O}(T \log(n))$ and that of the operators is T, this theorem says that scan can be implemented in parallel with the complexity of $\mathrm{O}(T \log(n))$. This implementation is used in global computation of the general implementation shown below. Figure 6.5 shows the implementation of scan by distributable homomorphism, where the number of steps is $\mathrm{O}(\log(n))$.

Now, we will develop the general implementation, i.e., the segmented implementation of accumulate and scan, in which each processor has one subarray of the input two-dimensional array rather than a single element. First, we will make the general implementation of accumulate. Then, we will instantiate it to make the general implementation of scan.

Let's consider to perform the main computation of accumulate independently on processors. Each processor can perform its local computation on the subarray, when it has the accumulation parameters necessary for its local computation. Thus, for the computation of accumulate, we first compute the accumulation parameters for each processor by global computation, then perform the local computation independently on each processor, and finally combine the local results by a global reduction.

The segmented parallel implementation of the accumulate is given by the following theorem. Note that we can use the simple case implementation of scan for the global computation of the accumulation parameters for the local computations.

Theorem 6.2 (Segmented accumulate)**.** The skeleton accumulate is executed seg-

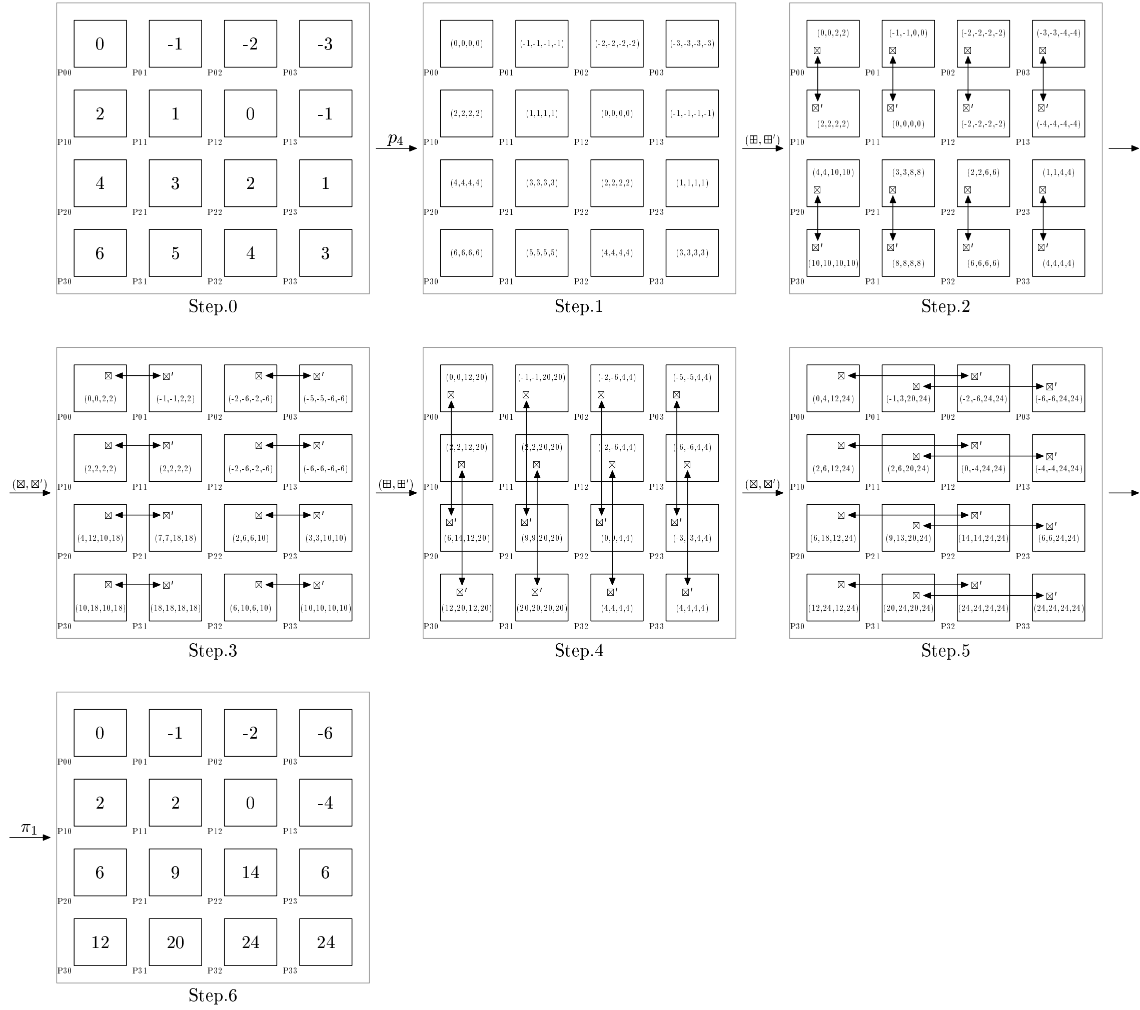


Figure 6.5. Parallel Computation of scan $(+,+)$ by Distributable Homomorphism

mentally in parallel as follows.

$$
\begin{aligned}
&[\![(\oplus',\otimes',f'),(\oplus,\otimes,f)]\!]\ x\ (e,u,v)\\
&\quad = \mathsf{reduce}\,(\oplus',\otimes')\,(\mathsf{map}\,(\lambda(x,e).[\![(\oplus',\otimes',f'),(\oplus,\otimes,f)]\!]\,x\,e)\,(\mathsf{zip}\ xs\ (\mathsf{zip}_3\ es\ us\ vs)))\\
&\quad \textbf{where}\\
&\qquad xs = \mathsf{dist}\ p\ q\ x\\
&\qquad dus = \mathsf{dist}\ 1\ q\ u\\
&\qquad dvs = \mathsf{dist}\ p\ 1\ v\\
&\qquad z = (|\!|e|\!| \varphi\ dus) \ominus (dvs\ \varphi\ fxs)\\
&\qquad fxs = \mathsf{map}\ (\mathsf{map}\ f)\ xs\\
&\qquad us = drop_c\ 1\ (take_r\ p\ (\mathsf{scan}\ (\mathsf{zipwith}\,(\oplus),\gg)\ (\mathsf{map}\ (\mathsf{reduce}_c\ (\oplus))\ z)))\\
&\qquad vs = drop_r\ 1\ (take_c\ q\ (\mathsf{scan}\ (\gg,\mathsf{zipwith}\,(\otimes))\ (\mathsf{map}\ (\mathsf{reduce}_r\ (\otimes))\ z)))\\
&\qquad es = take_r\ p\ (take_c\ q\ (\mathsf{scan}\ (\oplus,\otimes)\ (\mathsf{map}\ (\mathsf{reduce}\ (\oplus,\otimes))\ z)))
\end{aligned}
$$

Proof. The theorem is proved by induction on division of the input two-dimensional

array. The base case ($p = q = 1$) is proved as follows.

$$
\begin{array}{ll}
& \text{RHS } 1\ 1\ x\ (e,u,v) \\
= & \quad \{\ \text{The right hand side}\ \} \\
& \mathsf{reduce}\ (\oplus',\otimes')\ (\mathsf{map}\ (\lambda(x,e).[\![(\oplus',\otimes',f'),(\oplus,\otimes,f)]\!]\ x\ e)\ (\mathsf{zip}\ xs\ (\mathsf{zip}_3\ \ es\ us\ vs))) \\
& \quad\ \ \textbf{where}\ \ xs = \mathsf{dist}\ 1\ 1\ x \\
& \qquad\qquad\quad dus = \mathsf{dist}\ 1\ 1\ u \\
& \qquad\qquad\quad dvs = \mathsf{dist}\ 1\ 1\ v \\
& \qquad\qquad\quad z = (|\,|e|\,|\ \phi\ dus) \ominus (dvs\ \phi\ fxs) \\
& \qquad\qquad\quad qxs = \mathsf{map}\ (\mathsf{map}\ f)\ xs \\
& \qquad\qquad\quad us = drop_c\ 1\ (take_r\ 1\ (\mathsf{scan}\ (\mathsf{zipwith}\,(\oplus), \gg)\ (\mathsf{map}\ (\mathsf{reduce}_c\ (\oplus))\ z))) \\
& \qquad\qquad\quad vs = drop_r\ 1\ (take_c\ 1\ (\mathsf{scan}\ (\gg, \mathsf{zipwith}\,(\otimes))\ (\mathsf{map}\ (\mathsf{reduce}_r\ (\otimes))\ z))) \\
& \qquad\qquad\quad es = take_r\ 1\ (take_c\ 1\ (\mathsf{scan}\ (\oplus,\otimes)\ (\mathsf{map}\ (\mathsf{reduce}\ (\oplus,\otimes))\ z))) \\
= & \quad \{\ \text{Definition of } \mathsf{dist}\ \} \\
& \mathsf{reduce}\ (\oplus',\otimes')\ (\mathsf{map}\ (\lambda(x,e).[\![(\oplus',\otimes',f'),(\oplus,\otimes,f)]\!]\ x\ e)\ (\mathsf{zip}\ |x|\ (\mathsf{zip}_3\ \ es\ us\ vs))) \\
& \quad\ \ \textbf{where}\ \ z = (|\,|e|\,|\ \phi\ |u|) \ominus (|v|\ \phi\ |\,\mathsf{map}\ f\ x|) \\
& \qquad\qquad\quad us = drop_c\ 1\ (take_r\ 1\ (\mathsf{scan}\ (\mathsf{zipwith}\,(\oplus), \gg)\ (\mathsf{map}\ (\mathsf{reduce}_c\ (\oplus))\ z))) \\
& \qquad\qquad\quad vs = drop_r\ 1\ (take_c\ 1\ (\mathsf{scan}\ (\gg, \mathsf{zipwith}\,(\otimes))\ (\mathsf{map}\ (\mathsf{reduce}_r\ (\otimes))\ z))) \\
& \qquad\qquad\quad es = take_r\ 1\ (take_c\ 1\ (\mathsf{scan}\ (\oplus,\otimes)\ (\mathsf{map}\ (\mathsf{reduce}\ (\oplus,\otimes))\ z))) \\
= & \quad \{\ \text{Definition of } \mathsf{scan},\ take_c,\ take_r,\ dropc \text{ and } dropr\ \} \\
& \mathsf{reduce}\ (\oplus',\otimes')\ (\mathsf{map}\ (\lambda(x,e).[\![(\oplus',\otimes',f'),(\oplus,\otimes,f)]\!]\ x\ e)\ (\mathsf{zip}\ |x|\ (\mathsf{zip}_3\ \ es\ us\ vs))) \\
& \quad\ \ \textbf{where}\ \ us = |u|, vs = |v|, es = |e| \\
= & \quad \{\ \text{Definition of } \mathsf{zip} \text{ and } \mathsf{map}\ \} \\
& \mathsf{reduce}\ (\oplus',\otimes')\ |[\![(\oplus',\otimes',f'),(\oplus,\otimes,f)]\!]\ x\ (e,u,v)| \\
= & \quad \{\ \text{Definition of } \mathsf{reduce}\ \} \\
& [\![(\oplus',\otimes',f'),(\oplus,\otimes,f)]\!]\ x\ (e,u,v) \\
= & \quad \{\ \text{The left hand side}\ \} \\
& \text{LHS } x\ (e,u,v)
\end{array}
$$

The inductive case for $p = p_1 + p_2$ (i.e. $x = x_1 \ominus x_2$ and $v = v_1 \ominus v_2$) is proved as follows.

$$
\begin{array}{ll}
& \text{RHS } (p_1+p_2)\ q\ (x_1 \ominus x_2)\ (e,u,v_1 \ominus v_2) \\
= & \quad \{\ \text{The right hand side}\ \} \\
& \mathsf{reduce}\ (\oplus',\otimes')\ (\mathsf{map}\ (\lambda(x,e).[\![(\oplus',\otimes',f'),(\oplus,\otimes,f)]\!]\ x\ e)\ (\mathsf{zip}\ xs\ (\mathsf{zip}_3\ \ es\ us\ vs))) \\
& \quad \textbf{where}\ \ xs = \mathsf{dist}\ (p_1+p_2)\ q\ (x_1 \ominus x_2) \\
& \qquad\qquad\ dus = \mathsf{dist}\ 1\ q\ u \\
& \qquad\qquad\ dvs = \mathsf{dist}\ (p_1+p_2)\ 1\ (v_1 \ominus v_2) \\
& \qquad\qquad\ z = (|\,|e|\,|\ \phi\ dus) \ominus (dvs\ \phi\ fxs) \\
& \qquad\qquad\ qxs = \mathsf{map}\ (\mathsf{map}\ f)\ xs \\
& \qquad\qquad\ us = drop_c\ 1\ (take_r\ (p_1+p_2)\ (\mathsf{scan}\ (\mathsf{zipwith}\,(\oplus), \gg)\ (\mathsf{map}\ (\mathsf{reduce}_c\ (\oplus))\ z))) \\
& \qquad\qquad\ vs = drop_r\ 1\ (take_c\ q\ (\mathsf{scan}\ (\gg, \mathsf{zipwith}\,(\otimes))\ (\mathsf{map}\ (\mathsf{reduce}_r\ (\otimes))\ z))) \\
& \qquad\qquad\ es = take_r\ (p_1+p_2)\ (take_c\ q\ (\mathsf{scan}\ (\oplus,\otimes)\ (\mathsf{map}\ (\mathsf{reduce}\ (\oplus,\otimes))\ z))) \\
= & \quad \{\ \text{Definition of } \mathsf{dist} \text{ and , and introducing new variables}\ \} \\
& \mathsf{reduce}\ (\oplus',\otimes')\ (\mathsf{map}\ (\lambda(x,e).[\![(\oplus',\otimes',f'),(\oplus,\otimes,f)]\!]\ x\ e)\ (\mathsf{zip}\ (xs_1 \ominus xs_2)\ (\mathsf{zip}_3\ \ es\ us\ vs))) \\
& \quad \textbf{where}\ \ xs_1 = \mathsf{dist}\ p_1\ 1\ x_1,\ \ xs_2 = \mathsf{dist}\ p_2\ 1\ x_2 \\
& \qquad\qquad\ dus = \mathsf{dist}\ 1\ q\ u \\
& \qquad\qquad\ dvs_1 = \mathsf{dist}\ p_1\ 1\ v_1,\ \ dvs_2 = \mathsf{dist}\ p_2\ 1\ v_2 \\
& \qquad\qquad\ z_1 = (|\,|e|\,|\ \phi\ dus) \ominus (dvs_1\ \phi\ fxs_1),\ \ z_2' = (dvs_2\ \phi\ fxs_2) \\
& \qquad\qquad\ qxs_1 = \mathsf{map}\ (\mathsf{map}\ f)\ xs_1,\ \ qxs_2 = \mathsf{map}\ (\mathsf{map}\ f)\ xs_2 \\
& \qquad\qquad\ us = drop_c\ 1\ (take_r\ (p_1+p_2)\ (\mathsf{scan}\ (\mathsf{zipwith}\,(\oplus), \gg)\ (\mathsf{map}\ (\mathsf{reduce}_c\ (\oplus))\ (z_1 \ominus z_2')))) \\
& \qquad\qquad\ vs = drop_r\ 1\ (take_c\ q\ (\mathsf{scan}\ (\gg, \mathsf{zipwith}\,(\otimes))\ (\mathsf{map}\ (\mathsf{reduce}_r\ (\otimes))\ (z_1 \ominus z_2')))) \\
& \qquad\qquad\ es = take_r\ (p_1+p_2)\ (take_c\ q\ (\mathsf{scan}\ (\oplus,\otimes)\ (\mathsf{map}\ (\mathsf{reduce}\ (\oplus,\otimes))\ (z_1 \ominus z_2'))))
\end{array}
$$

$$
\begin{array}{ll}
= & \{ \text{ Properties shown below } \} \\
& \mathsf{reduce}\ (\oplus', \otimes')\ (\mathsf{map}\ \ (\lambda(x,e).[\![(\oplus',\otimes',f'),(\oplus,\otimes,f)]\!]\ x\ e) \\
& \qquad\qquad\qquad\qquad (\mathsf{zip}\ \ (xs_1 \ominus xs_2)\ (\mathsf{zip}_3\ \ (es_1 \ominus es_2)\ (us_1 \ominus us_2)\ (vs_1 \ominus vs_2)))) \\
& \quad \textbf{where} \\
& \qquad e_2 = e \oplus \mathsf{reduce}\ \ (\oplus,\otimes)\ v_1 \\
& \qquad u_2 = \mathsf{zipwith}\,(\oplus)\ u\ (\mathsf{reduce}_c(\oplus)\ (\mathsf{map}\ \ f\ x_1)) \\
& \qquad xs_1 = \mathsf{dist}\ \ p_1\ 1\ x_1,\ \ xs_2 = \mathsf{dist}\ \ p_2\ 1\ x_2 \\
& \qquad dus = \mathsf{dist}\ \ 1\ q\ u \\
& \qquad dus_2 = \mathsf{dist}\ \ 1\ q\ u_2 \\
& \qquad dvs_1 = \mathsf{dist}\ \ p_1\ 1\ v_1,\ \ dvs_2 = \mathsf{dist}\ \ p_2\ 1\ v_2 \\
& \qquad z_1 = (||e|| \phi\ dus) \ominus (dvs_1 \phi\ fxs_1),\ \ z_2 = (||e_2|| \phi\ dus_2) \ominus (dvs_2 \phi\ fxs_2) \\
& \qquad qxs_1 = \mathsf{map}\ \ (\mathsf{map}\ \ f)\ xs_1,\ \ qxs_2 = \mathsf{map}\ \ (\mathsf{map}\ \ f)\ xs_2 \\
& \qquad us_1 = drop_c\ 1\ (take_r\ p_1\ (\mathsf{scan}\ \ (\mathsf{zipwith}\,(\oplus), \gg)\ (\mathsf{map}\ \ (\mathsf{reduce}_c\ \ (\oplus))\ z_1))) \\
& \qquad vs_1 = drop_r\ 1\ (take_c\ q\ (\mathsf{scan}\ \ (\gg, \mathsf{zipwith}\,(\otimes))\ (\mathsf{map}\ \ (\mathsf{reduce}_r\ \ (\otimes))\ z_1))) \\
& \qquad es_1 = take_r\ p_1\ (take_c\ q\ (\mathsf{scan}\ \ (\oplus,\otimes)\ (\mathsf{map}\ \ (\mathsf{reduce}\ \ (\oplus,\otimes))\ z_1))) \\
& \qquad us_2 = drop_c\ 1\ (take_r\ p_2\ (\mathsf{scan}\ \ (\mathsf{zipwith}\,(\oplus), \gg)\ (\mathsf{map}\ \ (\mathsf{reduce}_c\ \ (\oplus))\ z_2))) \\
& \qquad vs_2 = drop_r\ 1\ (take_c\ q\ (\mathsf{scan}\ \ (\gg, \mathsf{zipwith}\,(\otimes))\ (\mathsf{map}\ \ (\mathsf{reduce}_r\ \ (\otimes))\ z_2))) \\
& \qquad es_2 = take_r\ p_2\ (take_c\ q\ (\mathsf{scan}\ \ (\oplus,\otimes)\ (\mathsf{map}\ \ (\mathsf{reduce}\ \ (\oplus,\otimes))\ z_2))) \\
= & \{ \text{ Definition of } \mathsf{reduce}, \mathsf{map}, \text{ and } \mathsf{zip} \ \} \\
& \mathsf{reduce}\ \ (\oplus',\otimes')\,(\mathsf{map}\,(\lambda(x,e).[\![(\oplus',\otimes',f'),(\oplus,\otimes,f)]\!]\,x\,e)\,(\mathsf{zip}\ xs_1\,(\mathsf{zip}_3\ \ es_1\ us_1\ vs_1))) \\
& \ \oplus \mathsf{reduce}\ \ (\oplus',\otimes')\,(\mathsf{map}\,(\lambda(x,e).[\![(\oplus',\otimes',f'),(\oplus,\otimes,f)]\!]\,x\,e)\,(\mathsf{zip}\ xs_2\,(\mathsf{zip}_3\ \ es_2\ us_2\ vs_2))) \\
& \quad \textbf{where} \\
& \qquad e_2 = e \oplus \mathsf{reduce}\ \ (\oplus,\otimes)\ v_1 \\
& \qquad u_2 = \mathsf{zipwith}\,(\oplus)\ u\ (\mathsf{reduce}_c\ \ (\oplus)\ (\mathsf{map}\ \ f\ x_1)) \\
& \qquad xs_1 = \mathsf{dist}\ \ p_1\ 1\ x_1,\ \ xs_2 = \mathsf{dist}\ \ p_2\ 1\ x_2 \\
& \qquad dus = \mathsf{dist}\ \ 1\ q\ u,\ \ dus_2 = \mathsf{dist}\ \ 1\ q\ u_2 \\
& \qquad dvs_1 = \mathsf{dist}\ \ p_1\ 1\ v_1,\ \ dvs_2 = \mathsf{dist}\ \ p_2\ 1\ v_2 \\
& \qquad z_1 = (||e|| \phi\ dus) \ominus (dvs_1 \phi\ fxs_1),\ \ z_2 = (||e_2|| \phi\ dus_2) \ominus (dvs_2 \phi\ fxs_2) \\
& \qquad qxs_1 = \mathsf{map}\ \ (\mathsf{map}\ \ f)\ xs_1,\ \ qxs_2 = \mathsf{map}\ \ (\mathsf{map}\ \ f)\ xs_2 \\
& \qquad us_1 = drop_c\ 1\ (take_r\ p_1\ (\mathsf{scan}\ \ (\mathsf{zipwith}\,(\oplus), \gg)\ (\mathsf{map}\ \ (\mathsf{reduce}_c\ \ (\oplus))\ z_1))) \\
& \qquad vs_1 = drop_r\ 1\ (take_c\ q\ (\mathsf{scan}\ \ (\gg, \mathsf{zipwith}\,(\otimes))\ (\mathsf{map}\ \ (\mathsf{reduce}_r\ \ (\otimes))\ z_1))) \\
& \qquad es_1 = take_r\ p_1\ (take_c\ q\ (\mathsf{scan}\ \ (\oplus,\otimes)\ (\mathsf{map}\ \ (\mathsf{reduce}\ \ (\oplus,\otimes))\ z_1))) \\
& \qquad us_2 = drop_c\ 1\ (take_r\ p_2\ (\mathsf{scan}\ \ (\mathsf{zipwith}\,(\oplus), \gg)\ (\mathsf{map}\ \ (\mathsf{reduce}_c\ \ (\oplus))\ z_2))) \\
& \qquad vs_2 = drop_r\ 1\ (take_c\ q\ (\mathsf{scan}\ \ (\gg, \mathsf{zipwith}\,(\otimes))\ (\mathsf{map}\ \ (\mathsf{reduce}_r\ \ (\otimes))\ z_2))) \\
& \qquad es_2 = take_r\ p_2\ (take_c\ q\ (\mathsf{scan}\ \ (\oplus,\otimes)\ (\mathsf{map}\ \ (\mathsf{reduce}\ \ (\oplus,\otimes))\ z_2))) \\
= & \{ \text{ The right hand side } \} \\
& \mathrm{RHS}\ p_1\ q\ x_1\ (e,u,v_1) \oplus \mathrm{RHS}\ p_2\ q\ x_2\ (e,u,v_2) \\
& \quad \textbf{where} \\
& \qquad e_2 = e \oplus \mathsf{reduce}\ \ (\oplus,\otimes)\ v_1 \\
& \qquad u_2 = \mathsf{zipwith}\,(\oplus)\ u\ (\mathsf{reduce}_c\ \ (\oplus)\ (\mathsf{map}\ \ f\ x_1)) \\
= & \{ \text{ Hypothesis of induction } \} \\
& [\![(\oplus',\otimes',f'),(\oplus,\otimes,f)]\!]\ x_1\ (e,u,v_1) \oplus [\![(\oplus',\otimes',f'),(\oplus,\otimes,f)]\!]\ x_2\ (e_2,u_2,v_2) \\
& \quad \textbf{where} \\
& \qquad e_2 = e \oplus \mathsf{reduce}\ \ (\oplus,\otimes)\ v_1 \\
& \qquad u_2 = \mathsf{zipwith}\,(\oplus)\ u\ (\mathsf{reduce}_c\ \ (\oplus)\ (\mathsf{map}\ \ f\ x_1)) \\
= & \{ \text{ Definition of } \mathsf{accumulate} \ \} \\
& [\![(\oplus',\otimes',f'),(\oplus,\otimes,f)]\!]\ (x_1 \ominus x_2)\ (e,u,v_1 \ominus v_2) \\
= & \{ \text{ The left hand side } \} \\
& \mathrm{LHS}\ (x_1 \ominus x_2)\ (e,u,v_1 \ominus v_2)
\end{array}
$$

To complete the above proof, we show the following properties.

$$\begin{array}{l} \mathit{drop}_c\ 1\ (\mathit{take}_r\ (p_1+p_2)\ (\mathsf{scan}\ (\mathsf{zipwith}\,(\oplus), \gg)\ (\mathsf{map}\ (\mathsf{reduce}_c\ (\oplus))\,(z_1 \ominus z_2'))))\\ \quad = \mathit{drop}_c\ 1\ (\mathit{take}_r\ p_1\ (\mathsf{scan}\ (\mathsf{zipwith}\,(\oplus), \gg)\ (\mathsf{map}\ (\mathsf{reduce}_c\ (\oplus))\,z_1)))\\ \qquad \ominus \mathit{drop}_c\ 1\ (\mathit{take}_r\ p_2\ (\mathsf{scan}\ (\mathsf{zipwith}\,(\oplus), \gg)\ (\mathsf{map}\ (\mathsf{reduce}_c\ (\oplus))\,z_2))) \end{array}$$

$$\begin{array}{l} \mathit{drop}_r\ 1\ (\mathit{take}_c\ q\ (\mathsf{scan}\ (\gg, \mathsf{zipwith}\,(\otimes))\ (\mathsf{map}\ (\mathsf{reduce}_r\ (\otimes))\ (z_1 \ominus z_2'))))\\ \quad = \mathit{drop}_r\ 1\ (\mathit{take}_c\ q\ (\mathsf{scan}\ (\gg, \mathsf{zipwith}\,(\otimes))\ (\mathsf{map}\ (\mathsf{reduce}_r\ (\otimes))\ z_1)))\\ \qquad \ominus \mathit{drop}_r\ 1\ (\mathit{take}_c\ q\ (\mathsf{scan}\ (\gg, \mathsf{zipwith}\,(\otimes))\ (\mathsf{map}\ (\mathsf{reduce}_r\ (\otimes))\ z_2))) \end{array}$$

$$\begin{array}{l} \mathit{take}_r\ (p_1+p_2)\ (\mathit{take}_c\ q\ (\mathsf{scan}\ (\oplus,\otimes)\ (\mathsf{map}\ (\mathsf{reduce}\ (\oplus,\otimes))\ (z_1 \ominus z_2'))))\\ \quad = \mathit{take}_r\ p_1\ (\mathit{take}_c\ q\ (\mathsf{scan}\ (\oplus,\otimes)\ (\mathsf{map}\ (\mathsf{reduce}\ (\oplus,\otimes))\ z_1)))\\ \qquad \ominus \mathit{take}_r\ p_2\ (\mathit{take}_c\ q\ (\mathsf{scan}\ (\oplus,\otimes)\ (\mathsf{map}\ (\mathsf{reduce}\ (\oplus,\otimes))\ z_2))) \end{array}$$

$$\begin{array}{l} \textbf{where}\\ \quad z_1 = (|\!|e|\!| \phi\ \mathit{dus}) \ominus (\mathit{dvs}_1 \phi\ \mathit{fxs}_1),\ \ z_2 = (|\!|e_2|\!| \phi\ \mathit{dus}_2) \ominus z_2'\\ \quad e_2 = e \oplus \mathsf{reduce}\ (\oplus,\otimes)\ (\mathsf{map}\ (\mathsf{reduce}\ (\oplus,\otimes))\ \mathit{dvs}_1)\\ \quad \mathit{dus}_2 = \mathsf{zipwith}\,(\mathsf{zipwith}\,(\oplus))\ \mathit{dus}\ (\mathsf{reduce}_c\ (\mathsf{zipwith}\,(\oplus))\ (\mathsf{map}\ (\mathsf{reduce}_c\ (\oplus))\ \mathit{fxs}_1)) \end{array}$$

These properties are proved using the following property (see the last part of proof of Theorem 3.1 .)

$$\mathit{bottom}\ (\mathsf{scan}\ (\oplus,\otimes)\ y) = \mathsf{scan}\ (\oplus,\otimes)\ (\mathsf{reduce}_c\ (\oplus)\ y)$$

The first property is proved as follows.

$$\begin{array}{ll} & \mathit{drop}_c\ 1\ (\mathit{take}_r\ (p_1+p_2)\ (\mathsf{scan}\ (\mathsf{zipwith}\,(\oplus), \gg)\ (\mathsf{map}\ (\mathsf{reduce}_c\ (\oplus))\,(z_1 \ominus z_2'))))\\ = & \{\ \text{Definition of } \mathsf{scan}\ \}\\ & \mathit{drop}_c\ 1\ (\mathit{take}_r\ (p_1+p_2)\ (s_1 \ominus \mathsf{map}_r\ (\mathsf{zipwith}\,(\mathsf{zipwith}\,(\oplus))\ (\mathit{bottom}\ s_1))\ s_2))\\ & \quad \textbf{where}\ \ s_1 = \mathsf{scan}\ (\mathsf{zipwith}\,(\oplus), \gg)\ (\mathsf{map}\ (\mathsf{reduce}_c\ (\oplus))\,z_1)\\ & \qquad\qquad\ \ s_2 = \mathsf{scan}\ (\mathsf{zipwith}\,(\oplus), \gg)\ (\mathsf{map}\ (\mathsf{reduce}_c\ (\oplus))\,z_2')\\ = & \{\ \text{Definition of } \mathit{take}_r,\ \mathit{drop}_c \text{ and } z_1\ \}\\ & \mathit{drop}_c\ 1\ (\mathit{take}_r\ p_1\ s_1)\\ & \quad \ominus \mathit{drop}_c\ 1\ (\mathit{take}_r\ p_2\ (\mathit{bottom}\ s_1 \ominus \mathsf{map}_r\ (\mathsf{zipwith}\,(\mathsf{zipwith}\,(\oplus))\ (\mathit{bottom}\ s_1))\ s_2))\\ & \quad \textbf{where}\ \ s_1 = \mathsf{scan}\ (\mathsf{zipwith}\,(\oplus), \gg)\ (\mathsf{map}\ (\mathsf{reduce}_c\ (\oplus))\,z_1)\\ & \qquad\qquad\ \ s_2 = \mathsf{scan}\ (\mathsf{zipwith}\,(\oplus), \gg)\ (\mathsf{map}\ (\mathsf{reduce}_c\ (\oplus))\,z_2')\\ = & \{\ \text{The above property, and } \mathit{bottom} \circ \mathit{bottom} = \mathit{bottom}\ \}\\ & \mathit{drop}_c\ 1\ (\mathit{take}_r\ p_1\ (\mathsf{scan}\ (\mathsf{zipwith}\,(\oplus), \gg)\ (\mathsf{map}\ (\mathsf{reduce}_c\ (\oplus))\,z_1)))\\ & \quad \ominus \mathit{drop}_c\ 1\ (\mathit{take}_r\ p_2\ (\mathsf{scan}\ (\mathsf{zipwith}\,(\oplus), \gg)\ r\\ & \qquad\qquad \ominus \mathsf{map}_r\ (\mathsf{zipwith}\,(\mathsf{zipwith}\,(\oplus))\ (\mathit{bottom}\ (\mathsf{scan}\ (\mathsf{zipwith}\,(\oplus), \gg)\ r)))\ s_2))\\ & \quad \textbf{where}\ \ r = \mathsf{reduce}_c\ (\mathsf{zipwith}\,(\oplus))\ (\mathsf{map}\ (\mathsf{reduce}_c\ (\oplus))\ z_1)\\ & \qquad\qquad\ \ s_2 = \mathsf{scan}\ (\mathsf{zipwith}\,(\oplus), \gg)\ (\mathsf{map}\ (\mathsf{reduce}_c\ (\oplus))\,z_2')\\ = & \{\ \text{Definition of } \mathsf{scan}\ \}\\ & \mathit{drop}_c\ 1\ (\mathit{take}_r\ p_1\ (\mathsf{scan}\ (\mathsf{zipwith}\,(\oplus), \gg)\ (\mathsf{map}\ (\mathsf{reduce}_c\ (\oplus))\,z_1)))\\ & \quad \ominus \mathit{drop}_c\ 1\ (\mathit{take}_r\ p_2\ (\mathsf{scan}\ (\mathsf{zipwith}\,(\oplus), \gg)\ (r \ominus \mathsf{map}\ (\mathsf{reduce}_c\ (\oplus))\,z_2')))\\ & \quad \textbf{where}\ \ r = \mathsf{reduce}_c\ (\mathsf{zipwith}\,(\oplus))\ (\mathsf{map}\ (\mathsf{reduce}_c\ (\oplus))\ z_1)\\ = & \{\ \text{Calculation of } r \text{ shown below}\ \}\\ & \mathit{drop}_c\ 1\ (\mathit{take}_r\ p_1\ (\mathsf{scan}\ (\mathsf{zipwith}\,(\oplus), \gg)\ (\mathsf{map}\ (\mathsf{reduce}_c\ (\oplus))\,z_1)))\\ & \quad \ominus \mathit{drop}_c\ 1\ (\mathit{take}_r\ p_2\ (\mathsf{scan}\ (\mathsf{zipwith}\,(\oplus), \gg)\ (r \ominus \mathsf{map}\ (\mathsf{reduce}_c\ (\oplus))\,z_2')))\\ & \quad \textbf{where}\\ & \qquad r = \mathsf{map}\ (\mathsf{reduce}_c\ (\oplus))\ (|\!|e_2|\!| \phi\ \mathit{dus}_2)\\ & \qquad e_2 = e \oplus \mathsf{reduce}\ (\oplus,\otimes)\ (\mathsf{map}\ (\mathsf{reduce}\ (\oplus,\otimes))\ \mathit{dvs}_1)\\ & \qquad \mathit{dus}_2 = \mathsf{zipwith}\,(\mathsf{zipwith}\,(\oplus))\ \mathit{dus}\ (\mathsf{reduce}_c\ (\mathsf{zipwith}\,(\oplus))\ (\mathsf{map}\ (\mathsf{reduce}_c\ (\oplus))\ \mathit{fxs}_1)) \end{array}$$

$$
\begin{aligned}
= & \quad \{ \text{ Introducing the variable } z_2 \ \} \\
& \mathit{drop}_c\ 1\ (\mathit{take}_r\ p_1\ (\mathsf{scan}\ (\mathsf{zipwith}\,(\oplus), \gg)\ (\mathsf{map}\ (\mathsf{reduce}_c\ (\oplus))\ z_1))) \\
& \quad \ominus \mathit{drop}_c\ 1\ (\mathit{take}_r\ p_2\ (\mathsf{scan}\ (\mathsf{zipwith}\,(\oplus), \gg)\ (r \ominus \mathsf{map}\ (\mathsf{reduce}_c\ (\oplus))\ z_2))) \\
& \quad \textbf{where} \\
& \qquad z_2 = (||e_2|| \phi\ \mathit{dus}_2) \ominus z_2' \\
& \qquad e_2 = e \oplus \mathsf{reduce}\ (\oplus, \otimes)\ (\mathsf{map}\ (\mathsf{reduce}\ (\oplus, \otimes))\ \mathit{dvs}_1) \\
& \qquad \mathit{dus}_2 = \mathsf{zipwith}\,(\mathsf{zipwith}\,(\oplus))\ \mathit{dus}\ (\mathsf{reduce}_c\ (\mathsf{zipwith}\,(\oplus))\ (\mathsf{map}\ (\mathsf{reduce}_c\ (\oplus))\ \mathit{fxs}_1))
\end{aligned}
$$

The calculation of r is as follows.

$$
\begin{aligned}
& \mathsf{reduce}_c\ (\mathsf{zipwith}\,(\oplus))\ (\mathsf{map}\ (\mathsf{reduce}_c\ (\oplus))\ z_1) \\
= & \quad \{ \text{ Definition of } z_1 \ \} \\
& \mathsf{reduce}_c\ (\mathsf{zipwith}\,(\oplus))\ (\mathsf{map}\ (\mathsf{reduce}_c\ (\oplus))\ ((||e|| \phi\ \mathit{dus}) \ominus (\mathit{dvs}_1 \phi\ \mathit{fxs}_1))) \\
= & \quad \{ \text{ Definition of } \mathsf{reduce}_c \text{ and } \mathsf{map} \ \} \\
& (\mathsf{zipwith}\,(\mathsf{zipwith}\,(\oplus))\ ||e||\ (\mathsf{reduce}_c\ (\mathsf{zipwith}\,(\oplus))\ (\mathsf{map}\ (\mathsf{reduce}_c\ (\oplus))\ \mathit{dvs}_1))) \\
& \quad \phi (\mathsf{zipwith}\,(\mathsf{zipwith}\,(\oplus))\ (\mathsf{map}\ (\mathsf{reduce}_c\ (\oplus))\ \mathit{dus}) \\
& \qquad\qquad\qquad (\mathsf{reduce}_c\ (\mathsf{zipwith}\,(\oplus))\ (\mathsf{map}\ (\mathsf{reduce}_c\ (\oplus))\ \mathit{fxs}_1))) \\
= & \quad \left\{ \begin{array}{l} \text{If } \mathit{width}\ x = 1, \text{ then } \mathsf{reduce}_c\ (\oplus)\ x = |\,\mathsf{reduce}\ (\oplus, \otimes)\ x|, \\ \mathsf{reduce}\ (\mathsf{zipwith}\,(\oplus, \otimes)) \circ \mathsf{map}\,|\cdot| = |\cdot| \circ \mathsf{reduce}\ (\oplus, \otimes) \end{array} \right\} \\
& (\mathsf{zipwith}\,(\mathsf{zipwith}\,(\oplus))\ ||e||\ ||\,\mathsf{reduce}\ (\oplus, \otimes)\ (\mathsf{map}\ (\mathsf{reduce}\ (\oplus, \otimes))\ \mathit{dvs}_1)||) \\
& \quad \phi (\mathsf{zipwith}\,(\mathsf{zipwith}\,(\oplus))\ (\mathsf{map}\ (\mathsf{reduce}_c\ (\oplus))\ \mathit{dus}) \\
& \qquad\qquad\qquad (\mathsf{reduce}_c\ (\mathsf{zipwith}\,(\oplus))\ (\mathsf{map}\ (\mathsf{reduce}_c\ (\oplus))\ \mathit{fxs}_1))) \\
= & \quad \{ \text{ If the height of each element of x is one, then } \mathsf{map}\ (\mathsf{reduce}_c\ (\oplus))\ x = x \ \} \\
& (\mathsf{zipwith}\,(\mathsf{zipwith}\,(\oplus))\ ||e||\ ||\,\mathsf{reduce}\ (\oplus, \otimes)\ (\mathsf{map}\ (\mathsf{reduce}\ (\oplus, \otimes))\ \mathit{dvs}_1)||) \\
& \quad \phi\ \mathsf{map}\ (\mathsf{reduce}_c\ (\oplus))\ (\mathsf{zipwith}\,(\mathsf{zipwith}\,(\oplus))\ \mathit{dus} \\
& \qquad\qquad\qquad (\mathsf{reduce}_c\ (\mathsf{zipwith}\,(\oplus))\ (\mathsf{map}\ (\mathsf{reduce}_c\ (\oplus))\ \mathit{fxs}_1))) \\
= & \quad \{ \text{ Definition of } \mathsf{zipwith}, \text{ introducing the variable } \mathit{dus}_2 \ \} \\
& ||e \oplus \mathsf{reduce}\ (\oplus, \otimes)\ (\mathsf{map}\ (\mathsf{reduce}\ (\oplus, \otimes))\ \mathit{dvs}_1)|| \phi\ \mathsf{map}\ (\mathsf{reduce}_c\ (\oplus))\ \mathit{dus}_2 \\
& \quad \textbf{where} \\
& \qquad \mathit{dus}_2 = \mathsf{zipwith}\,(\mathsf{zipwith}\,(\oplus))\ \mathit{dus}\ (\mathsf{reduce}_c\ (\mathsf{zipwith}\,(\oplus))\ (\mathsf{map}\ (\mathsf{reduce}_c\ (\oplus))\ \mathit{fxs}_1)) \\
= & \quad \{ \text{ Definition of } \mathsf{map} \text{ and } \mathsf{reduce}_c, \text{ introducing the variable } e_2 \ \} \\
& \mathsf{map}\ (\mathsf{reduce}_c\ (\oplus))\ (||e_2|| \phi\ \mathit{dus}_2) \\
& \quad \textbf{where} \\
& \qquad e_2 = e \oplus \mathsf{reduce}\ (\oplus, \otimes)\ (\mathsf{map}\ (\mathsf{reduce}\ (\oplus, \otimes))\ \mathit{dvs}_1) \\
& \qquad \mathit{dus}_2 = \mathsf{zipwith}\,(\mathsf{zipwith}\,(\oplus))\ \mathit{dus}\ (\mathsf{reduce}_c\ (\mathsf{zipwith}\,(\oplus))\ (\mathsf{map}\ (\mathsf{reduce}_c\ (\oplus))\ \mathit{fxs}_1))
\end{aligned}
$$

Now, the proof of the first property is completed.

The second property is proved similarly. Assuming that $\mathit{height}\ y_1 = 1$, the key calculation is as follows.

$$
\begin{aligned}
& \mathit{drop}_r\ 1\ (\mathsf{scan}\ (\gg, \mathsf{zipwith}\,(\otimes))\ (y_1 \ominus y_2)) \\
= & \quad \{ \text{ Definition of } \mathsf{scan} \ \} \\
& \mathit{drop}_r\ 1\ (\mathsf{scan}\ (\gg, \mathsf{zipwith}\,(\otimes))\ y_1 \\
& \qquad \ominus \mathsf{map}_r\ (\mathsf{zipwith}\,(\gg)\ (\mathsf{scan}\ (\gg, \mathsf{zipwith}\,(\otimes))\ y_1))\ (\mathsf{scan}\ (\gg, \mathsf{zipwith}\,(\otimes))\ y_2)) \\
= & \quad \{ \text{ Drop the top row } \} \\
& \mathsf{map}_r\ (\mathsf{zipwith}\,(\gg)\ (\mathsf{scan}\ (\gg, \mathsf{zipwith}\,(\otimes))\ y_1))\ (\mathsf{scan}\ (\gg, \mathsf{zipwith}\,(\otimes))\ y_2)) \\
= & \quad \{ \ \mathsf{zipwith}\,(\gg)\ x\ y = y \ \} \\
& \mathsf{scan}\ (\gg, \mathsf{zipwith}\,(\otimes))\ y_2
\end{aligned}
$$

The third property is proved similarly. The key calculation is as follows.

$$
\begin{array}{ll}
 & \mathsf{reduce}_c\ (\oplus)\ (\mathsf{map}\ (\mathsf{reduce}\ (\oplus,\otimes))\ z_1) \\
= & \quad \{\ \text{Definition of } z_1\ \} \\
 & \mathsf{reduce}_c\ (\oplus)\ (\mathsf{map}\ (\mathsf{reduce}\ (\oplus,\otimes))\ ((|\,|e|\,|\ \phi\ dus) \ominus (dvs_1\ \phi\ fxs_1))) \\
= & \quad \{\ \text{Definition of } \mathsf{reduce}_c \text{ and } \mathsf{map}\ \} \\
 & (\mathsf{zipwith}\,(\oplus)\ |e|\ (\mathsf{reduce}_c\ (\oplus)\ (\mathsf{map}\ (\mathsf{reduce}\ (\oplus,\otimes))\ dvs_1)))) \\
 & \quad \phi(\mathsf{zipwith}\,(\oplus)\ (\mathsf{map}\ (\mathsf{reduce}\ (\oplus,\otimes))\ dus)\ (\mathsf{reduce}_c\ (\oplus)\ (\mathsf{map}\ (\mathsf{reduce}\ (\oplus,\otimes))\ fxs_1))) \\
= & \quad \{\ \text{If } width\ x = 1, \text{ then } \mathsf{reduce}_c\ (\oplus)\ x = |\,\mathsf{reduce}\ (\oplus,\otimes)\ x|\ \} \\
 & (\mathsf{zipwith}\,(\oplus)\ |e|\ |\,\mathsf{reduce}\ (\oplus,\otimes)\ (\mathsf{map}\ (\mathsf{reduce}_c\ (\oplus))\ dvs_1)|) \\
 & \quad \phi(\mathsf{zipwith}\,(\oplus)\ (\mathsf{map}\ (\mathsf{reduce}\ (\oplus,\otimes))\ dus)\ (\mathsf{reduce}_c\ (\oplus)\ (\mathsf{map}\ (\mathsf{reduce}\ (\oplus,\otimes))\ fxs_1))) \\
= & \quad \{\ \text{Definition of } \mathsf{zipwith}, \text{ moving the application of } \otimes \text{ to the front}\ \} \\
 & |e \oplus \mathsf{reduce}\ (\oplus,\otimes)\ (\mathsf{map}\ (\mathsf{reduce}_c\ (\oplus))\ dvs_1)| \\
 & \quad \phi\ \mathsf{map}\ (\mathsf{reduce}\ (\oplus,\otimes))\ (\mathsf{zipwith}\,(\mathsf{zipwith}(\oplus))\ dus \\
 & \qquad\qquad\qquad\qquad (\mathsf{reduce}_c\ (\mathsf{zipwith}\,(\oplus))\ (\mathsf{map}\ (\mathsf{reduce}_c\ (\oplus))\ fxs_1))) \\
= & \quad \{\ \text{Introducing } e_2 \text{ and } dus_2, \text{ definition of } \mathsf{map} \text{ and } \mathsf{reduce}\ \} \\
 & \mathsf{map}\ (\mathsf{reduce}\ (\oplus,\otimes))\ (|\,|e_2|\,|\ \phi\ dus_2) \\
 & \quad \textbf{where} \\
 & \qquad e_2 = e \oplus \mathsf{reduce}\ (\oplus,\otimes)\ (\mathsf{map}\ (\mathsf{reduce}\ (\oplus,\otimes))\ dvs_1) \\
 & \qquad dus_2 = \mathsf{zipwith}\,(\mathsf{zipwith}\,(\oplus))\ dus\ (\mathsf{reduce}_c\ (\mathsf{zipwith}\,(\oplus))\ (\mathsf{map}\ (\mathsf{reduce}_c\ (\oplus))\ fxs_1))
\end{array}
$$

Finally, the following calculation completes the proof.

$$
\begin{array}{ll}
 & \mathsf{zipwith}\,(\mathsf{zipwith}\,(\oplus))\ dus\ (\mathsf{reduce}_c\ (\mathsf{zipwith}\,(\oplus))\ (\mathsf{map}\ (\mathsf{reduce}_c\ (\oplus))\ fxs_1)) \\
= & \quad \{\ \text{Definition of } dus \text{ and } fxs_1\ \} \\
 & \mathsf{zipwith}\,(\mathsf{zipwith}\,(\oplus))\ (\mathsf{dist}\ 1\ q\ u) \\
 & \qquad\qquad (\mathsf{reduce}_c\ (\mathsf{zipwith}\,(\oplus))\ (\mathsf{map}\ (\mathsf{reduce}_c\ (\oplus))\ (\mathsf{map}\ (\mathsf{map}\ f)\ (\mathsf{dist}\ p\ q\ x)))) \\
= & \quad \{\ \text{Moving } \mathsf{dist} \text{ to the front}\ \} \\
 & \mathsf{dist}\ 1\ q\ (\mathsf{zipwith}\,(\oplus)\ u\ (\mathsf{reduce}_c\ (\oplus)\ (\mathsf{map}\ f\ x)))
\end{array}
$$

$$
\begin{array}{ll}
 & e \oplus \mathsf{reduce}\ (\oplus,\otimes)\ (\mathsf{map}\ (\mathsf{reduce}\ (\oplus,\otimes))\ dvs_1) \\
= & \quad \{\ \text{Definition of } dvs_1 \text{ and (reverse) promotion of } \mathsf{reduce}\ (\oplus,\otimes)\ \} \\
 & e \oplus \mathsf{reduce}\ (\oplus,\otimes)\ (\mathsf{reduce}\ (\ominus,\phi)\ (\mathsf{dist}\ p\ q\ v_1)) \\
= & \quad \{\ \mathsf{gather} = \mathsf{reduce}\ (\ominus,\phi) \text{ and } \mathsf{gather} \circ \mathsf{dist}\ p\ q = id\ \} \\
 & e \oplus \mathsf{reduce}\ (\oplus,\otimes)\ v_1
\end{array}
$$

The inductive case for $q = q_1 + q_2$ is proved similarly. □

The computation of us, vs and es is done using the previous implementation of scan by distributable homomorphism, in which the complexity of the operators is $\mathrm{O}(\sqrt{n/(pq)})$ for an $n \times n$ array and pq processors. Thus, its computational complexity is $\mathrm{O}(\sqrt{n/(pq)}\log(pq))$. The computation of $\mathsf{reduce}\ (\oplus', \otimes')$ is $\mathrm{O}(\log(pq))$ using tree-like computation, and that of $\mathsf{map}\ (\lambda(x,e).[\![(\oplus', \otimes', f'), (\oplus, \otimes, f)]\!]\ x\ e)$ is $\mathrm{O}(n^2/(pq))$ since it is performed independently on each processor. Thus, total complexity is $\mathrm{O}(n^2/P + \sqrt{n^2/P}\log(P))$ for an $n \times n$ array and $P = pq$ processors. Note that we can compute us, vs, and es simultaneously with a single scan by tupling them, since they have the same structure of computation.

How, we can obtain the segmented implementation of scan using the theorem.

Corollary 6.3 (Segmented scan)**.** The skeleton scan is executed segmentally in

parallel as follows.

$$\begin{array}{l}
\mathsf{scan}\ (\oplus,\otimes) \\
\quad = \mathsf{gather}\ (\mathsf{map}\ (\lambda(x,e).[\![(\ominus, \phi, \lambda x\,e.e),(\oplus,\otimes,f)]\!]\ x\ e)\ (\mathsf{zip}\ xs\ (\mathsf{zip}_3\ \ es\ us\ vs))) \\
\quad \mathbf{where} \\
\qquad xs = \mathsf{dist}\ p\ q\ x \\
\qquad dus = \mathsf{dist}\ 1\ q\ (|\imath| \ \phi\ \cdots \ \phi\ |\imath|) \\
\qquad dvs = \mathsf{dist}\ p\ 1\ (|\imath| \ominus \cdots \ominus |\imath|) \\
\qquad z = (||e|| \ \phi\ dus) \ominus (dvs \ \phi\ fxs) \\
\qquad fxs = \mathsf{map}\ (\mathsf{map}\ f)\ xs \\
\qquad us = drop_c\ 1\ (take_r\ p\ (\mathsf{scan}\ (\mathsf{zipwith}\,(\oplus), \gg)\ (\mathsf{map}\ (\mathsf{reduce}_c\ (\oplus))\ z))) \\
\qquad vs = drop_r\ 1\ (take_c\ q\ (\mathsf{scan}\ (\gg, \mathsf{zipwith}\,(\otimes))\ (\mathsf{map}\ (\mathsf{reduce}_r\ (\otimes))\ z))) \\
\qquad es = take_r\ p\ (take_c\ q\ (\mathsf{scan}\ (\oplus,\otimes)\ (\mathsf{map}\ (\mathsf{reduce}\ (\oplus,\otimes))\ z))) \\
\qquad \mathbf{where}\, \imath \oplus a = a \oplus \imath = a, \imath \otimes a = a \otimes \imath = a
\end{array}$$

The computational complexity of scan is $O(n^2/P + \sqrt{n^2/P}\log(P))$ for an $n \times n$ array and P processors, which is equal to that argued in previous section. Note that $[\![(\ominus, \phi, \lambda x\,e.e),(\oplus,\otimes,f)]\!]$ is scan except that it takes the initial values (accumulation parameters.)

6.2.3 Refined Implementation for Nested Use of Skeletons

The ordinary implementation for tree-like computation of the skeleton reduce uses only one processor to each operator. Precisely saying, one processor receives the value of the other processor and applies the operator to the pair of its value and the receive value by only itself, while the other processor becomes idle after sending its value.

This becomes a problem when the complexity is not small, although it is not a problem when the computational complexity of the operator is small (e.g., $O(1)$ or $O(\log(n))$ for the input of size n). For example, the program *mrs* for the maximum rectangle sum problem in Section 4.3.3 has a reduce with operators of which complexity is $O(n^3)$ for the input matrix of $n \times n$. Its parallel computational complexity for the ordinary implementation using P processors is as follows:

$$T(n \times n, P) = n^3 + T(n \times n/2, P/2) = O(n^3)\ .$$

Since the complexity of the last operator is big and dominant, the parallel complexity does not depend on the number of processors P. Thus, this program is not efficiently executed in parallel even if the number of processors becomes large. Generally, this problem arises when the complexity of the operator is equal or bigger than $O(n)$. This situation occurs when the operator contains skeletons, i.e., when the program contains nested uses of skeletons.

The main cause of this problem is that the implementation uses only one processor for each operator in the tree-like computation; some processors remain idle

during the computation. If we can use these idle processors for the computation of reduction operators, we may improve the efficiency.

We will use those idle processors to solve this problem. In the refined implementation, a set of available processors is attached to each operator of the reduce skeleton so that these processors are used for the parallel execution of the skeletons of the operator. Furthermore, since map is often used to describe explicit parallelization of independent computations, we also give refined implementation of map having a set of processors.

In the following, $\underline{f}_P$ means performing the computation of f to a distributed matrix using the set of processors P in parallel, and $\underline{\underline{f}}_P$ means to distribute the argument matrix and perform the computation f using P in parallel.

We formalize the computation of skeletons with the attached set of processors. An outermost skeleton marked by $\underline{\underline{\cdot}}_P$ distributes the argument matrix among the set of processors P ($|P| \geq p \times q$) and does local computation in parallel by using map marked by $\underline{\cdot}_P$. The reduce performs global tree-like computation marked by $\underline{\cdot}_P$.

$$\begin{array}{lcl} \underline{\underline{\mathsf{map}\ f}}_P & = & \mathsf{gather} \circ \underline{\mathsf{map}\ (\mathsf{map}\ f)}_P \circ \mathsf{dist}\ p\ q \\ \underline{\underline{\mathsf{reduce}\ (\oplus, \otimes)}}_P & = & \underline{\mathsf{reduce}\ (\oplus, \otimes)}_P \circ \underline{\mathsf{map}\ (\mathsf{reduce}\ (\oplus, \otimes))}_P \circ \mathsf{dist}\ p\ q \end{array}$$

The map marked by $\underline{\cdot}_P$ divides the set of available processors P into two sets of P_1 and P_2 in the cases of $\ominus$ and Φ, and performs the computation of f in parallel on P in the base case. The division of P is done so that the ratio of the sizes of P_1 and P_2 is the same of that of x and y.

$$\begin{array}{lcl} \underline{\mathsf{map}\ f}_P\ |a| & = & \left|\underline{\underline{f}}_P\ a\right| \\ \underline{\mathsf{map}\ f}_P\ (x \ominus y) & = & \underline{\mathsf{map}\ f}_{P_1} x \ominus \underline{\mathsf{map}\ f}_{P_2} y \\ \underline{\mathsf{map}\ f}_P\ (x \Phi y) & = & \underline{\mathsf{map}\ f}_{P_1} x \Phi \underline{\mathsf{map}\ f}_{P_2} y \end{array}$$

The reduce marked by $\underline{\cdot}_P$ for the global computation divides the set of available processors P into two sets of P_1 and P_2 in the cases of $\ominus$ and Φ, and performs the computation of each operator in parallel on P that is the set of all available processors for the operator. The division of P is done so that the ratio of the sizes of P_1 and P_2 is the same of that of x and y.

$$\begin{array}{lcl} \underline{\mathsf{reduce}\ (\oplus, \otimes)}_P\ |a| & = & a \\ \underline{\mathsf{reduce}\ (\oplus, \otimes)}_P\ (x \ominus y) & = & \underline{\mathsf{reduce}\ (\oplus, \otimes)}_{P_1} x\ \underline{\underline{\oplus}}_P\ \underline{\mathsf{reduce}\ (\oplus, \otimes)}_{P_2} y \\ \underline{\mathsf{reduce}\ (\oplus, \otimes)}_P\ (x \Phi y) & = & \underline{\mathsf{reduce}\ (\oplus, \otimes)}_{P_1} x\ \underline{\underline{\otimes}}_P\ \underline{\mathsf{reduce}\ (\oplus, \otimes)}_{P_2} y \end{array}$$

This refined implementation enables parallel execution of the last operator in the tree-like computation. It can reduce the dominant complexity, and thus, improve the parallel complexity of the program.

Using this refined implementation, computational complexity of mrs using P processors is as follows:

$$T(n \times n, P) = n^3/P + \log(P) + T(n \times n/2, P/2) = \mathrm{O}(n^3/P + \log(P)) .$$

The parallel complexity of the last operator is inversely proportional to P because the computation of the operator is performed in parallel among P processors. Thus, this program is efficiently executed in parallel as the number of processors becomes large.

6.2.4 Experiment Results

We implemented the parallel skeletons as a library [1] with C++ and MPI, and did our experiments on a small-scale PC cluster of sixteen uniform PCs connected with Gigabit Ethernet. Each PC has a CPU of Pentium4 3.0GHz (Hyper Threading ON) and 1GB memory, with Linux 2.6.8 for the OS, gcc 3.4 for the compiler, and mpich 1.2.6 for the MPI.

Figures 6.6 through 6.8 show speedups of the following parallel programs described with the parallel skeletons.

Frobenius Norm
$\quad fnorm = \mathsf{reduce}\ (+,+) \circ \mathsf{map}\ \ square\ ,$

Matrix Multiplication
$\quad mm$ (composition of skeletons; see Section 3.2.2).

Maximum Rectangle Sum
$\quad mrs$ (derived program written by skeletons; see Section 4.3.3).

Image Filter
$\quad ifilter = \mathsf{map}\ \ f \circ \mathsf{scanr}\ \ (\oplus'_r, \otimes'_r) \circ \mathsf{map}\ \ f_r \circ \mathsf{scan}(\oplus'_f, \otimes'_f)\ \circ \mathsf{map}\ \ f_f$
(optimized program written by composition of skeletons); see Section 5.3.2.

The inputs are a 4000×4000 matrix for *fnorm*, , 1000×1000 matrices for *mm* and *mrs*, and a 2000×2000 matrix for *ifilter*. The computational times of the above programs on one processor are 0.21s, 2.06s, 14.1s, and 116.8s respectively.

The result shows programs described with skeletons can be executed efficiently in parallel, and proves the success of our framework. Both of the simple program (*fnorm*) and the more complicated program (*ifilter*) achieve linear speedups by the parallel implementation of the skeletons. The speedup of matrix multiplication is super-linear. This can happen in large matrix operations where the matrix on a single processor is large with respect to the cache size. It is reasonable that the super-linear speedup is achieved here. Two lines in Figure 6.8 are the results of *mrs* using the simple implementation (labeled 'mrs') and the refined implementation for nested use of skeletons (labeled 'mrs(refined)'), respectively. The refined one gives us a good speedup nearly proportional to the number of processors for 16 processors, while the speedup of the simple one is reaching the limit soon. This result indicates that even if a program written by the skeletons contains nested use of skeletons, we can execute it efficiently in parallel when we successfully execute

[1] This implementation is included in the skeleton library 'SkeTo'. It is available at the web page http://www.ipl.t.u-tokyo.ac.jp/sketo/

the inner skeletons in parallel. However, even for the refined implementation of *mrs*, the speedup is expected to reach the limit earlier than other examples, because it uses heavy operators and needs to transfer larger data structures among processors.

Finally, we list a part of the C++ code of *mm* written with the skeleton library to give a concrete impression of the conciseness our library provides. Figure 6.10 shows the C++ code of the function `mm` that implements *mm*. The function takes three distributed matrices (i.e., two-dimensional arrays): `Z2` to store the result, and `X2` and `Y2` for the input. Two functions `all_rows` and `all_cols` implements the corresponding functions in *mm* with provided skeletons. The results of these functions are stored in two arrays `A2` and `B2`. These arrays are then supplied to the provided skeleton `zipwith`, which is an implementation of skeleton zipwith. Here, users do not need to be conscious of its parallel implementation; what they need to do is simply to supply two distributed matrices and the function object to be applied to elements.

6.3 Implementation of Fusion Optimizations

In this section, we will briefly report our simple implementations of fusion optimizations: the domain-independent fusion optimization in Section 4.1.3, and the domain-specific fusion optimization in Section 5.1. We will also report some experiment results to show the effect of optimizations.

6.3.1 A Simple System for Domain-Independent Fusion Optimization

SkeTo [MIEH06] library is equipped with the domain-independent fusion optimization shown in Section 4.1.3. We will briefly review its mechanism.

The optimization mechanism consists of three components: the user-interface database, the transformation engine, and the implementation database. Taking a C++ program with skeletons, the transformation system first converts the skeletons into their *structured forms* (i.e., accumulate and build$_J$ in Section 4.1.3) by applying rules given as meta-programs in the user-interface database. Then, the transformation engine manipulates and fuses the structured forms with the shortcut fusion rules (Theorem 4.4 and its variants). Finally, the system generates optimized skeleton compositions from the fused structured forms by the rules in the implementation database.

The optimization system is implemented in OpenC++, a meta language for manipulating C++ programs.

For example, the following two lines of code to apply two functions f and g to the list x

```
y = x->map(f);
z = y->map(g);
```

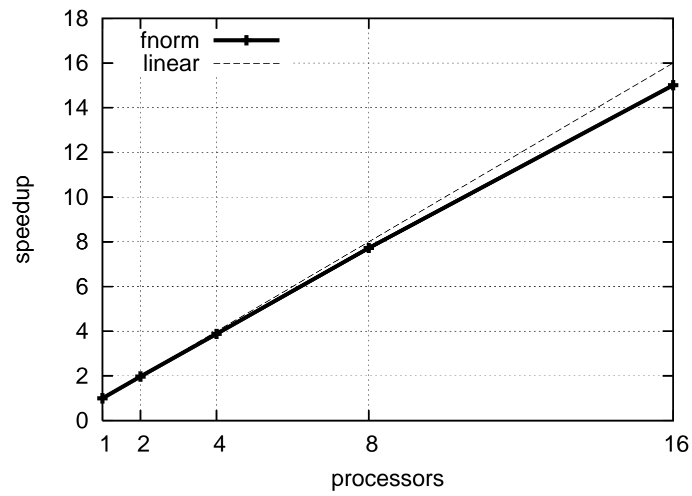


Figure 6.6. Speedup of F-norm

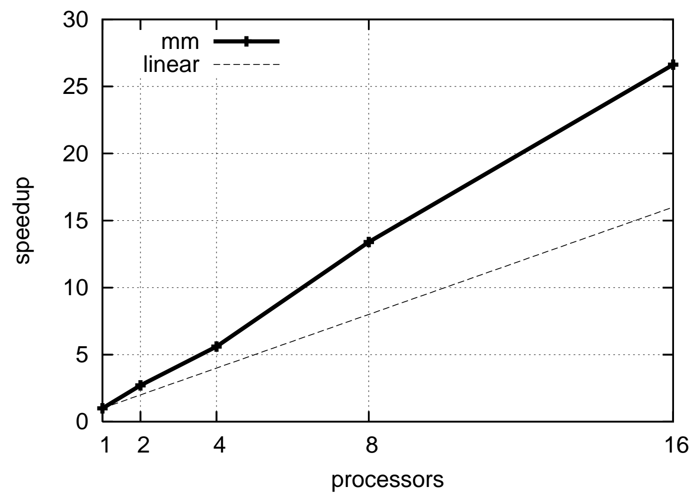


Figure 6.7. Speedup of matrix multiplication

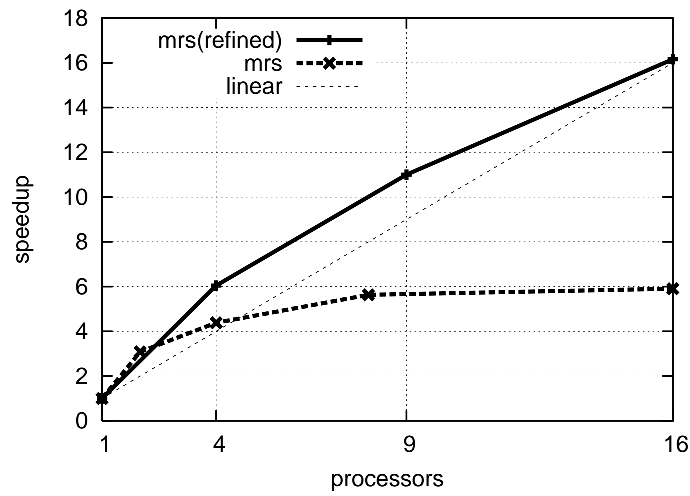


Figure 6.8. Speedup of maximum rectangle sum

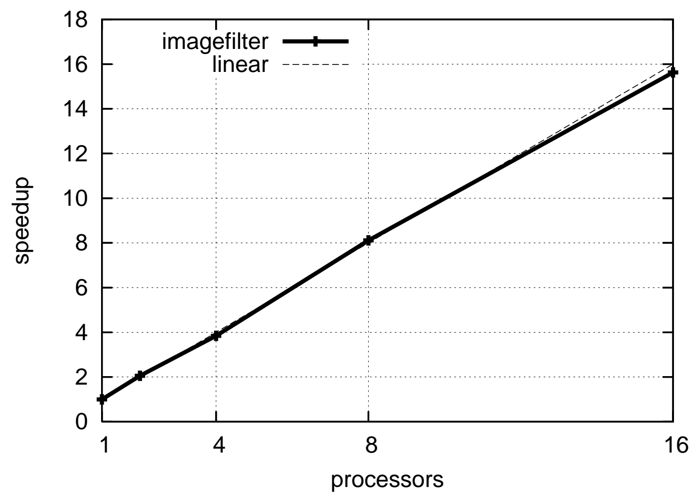


Figure 6.9. Speedup of Image Filter

```
template <class C, class A, class B>
void mm(dist_matrix<C> &Z2, const dist_matrix<A> &X2, const dist_matrix<B> &Y2)
{
    dist_matrix < matrix < int > > *A2;
    dist_matrix < matrix < int > > *B2;
    A2 = all_rows2(X2);
    B2 = all_cols2(Y2);
    m_skeletons::zipwith(Iprod<C>(), *A2, *B2, Z2);
    delete B2;
    delete A2;
}
```

Figure 6.10. C++ code of matrix multiplication described with parallel skeletons. `dist_matrix` is the type of distributed tow-dimensional arrays. Two functions `all_rows` and `all_cols` are also implemented with skeletons. The function `m_skeleton::zipwith` is the implementation of zipwith skeleton.

is converted by OpenC++ into the following parse tree , which is a nested list of tokens.

```
[[sqs = [[subs -> map] ( [sq] )]] ;]
[[sqs = [[subs -> map] ( [sq] )]] ;]
```

Then, those parse trees are transformed into structured forms by rules given as meta-programs in the user-interface database as follows.

```
[‘buildJ‘ y x [[var_c] [var_s [f]] [var_e]] ;]
[‘buildJ‘ z y [[var_c] [var_s [g]] [var_e]] ;]
```

Here, the two maps are structured by the build_J. The transformation engine then transform the structured forms into the following one structured form, by applying the fusion rules.

```
[‘buildJ‘ z y [[var_c] [var_s [g] [f]] [var_e]] ;]
```

Finally, the system transform the structured form back into the following invocation of the skeleton.

```
y = x->map(compose(g, f));
```

Here, `compose` is a function to compose two functions. The system has a map to transform structured forms into skeletons.

6.3.2 A Simple System for Domain-Specific Fusion Optimization

The system reads a skeleton program written with skeleton library SkeTo [MIEH06], and generates an optimized C++ code of the program. The optimized code is written with direct use of MPI or skeletons in SkeTo library.

The optimization flow of the system is as follow.

Phase. 1 Parsing the input program to extract skeleton compositions.

Phase. 2 Building normal forms from the extracted skeleton compositions by domain-specific fusions.

Phase. 3 Generating optimized codes of efficient implementations to replace the skeleton compositions in the input program.

We will explain the flow with the domain-specific fusion developed in Section 5.2.

In the first phase, the system reads C++ program written with skeletons provided by skeleton library SkeTo. Figure 6.11 shows an example C++ program that implements the running example *next* in Section 5.2 as the C++ function `nextZW`, which uses zipwith instead of the composition of map and zip.

Then, the system extracts skeleton compositions of specific patterns from the C++ program. For the program in Figure 6.11, it extracts an instance of *Program*, i.e., the following composition.

$$\begin{array}{l} \underline{\mathsf{map}}\ add\ (\\ \quad \underline{\mathsf{zip}}\ (\underline{\mathsf{map}}\ add\ (\underline{\mathsf{zip}}\ (\underline{\mathsf{map}}\ \mathtt{c_2x}\ (\underline{\mathsf{shift}}_{\gg}\ \mathtt{b0}\ (\underline{\mathsf{shift}}_{\gg}\ \mathtt{b1}\ u)))\ (\underline{\mathsf{map}}\ \mathtt{c_1x}\ (\underline{\mathsf{shift}}_{\gg}\ \mathtt{b1}\ u)))) \\ \qquad (\underline{\mathsf{map}}\ add\ (\underline{\mathsf{zip}}\ (\underline{\mathsf{map}}\ \mathtt{c0x}\ u)\ (\underline{\mathsf{map}}\ \mathtt{c1x}\ (\underline{\mathsf{shift}}_{\ll}\ \mathtt{b2}\ u))))) \end{array}$$

When there are more than one patterns of compositions (i.e., domains of computations), the system tests those patterns one by one. The unit of extracting skeleton compositions is a function declaration in the input program.

In the second phase, the system transforms the extracted compositions into normal forms by using fusions specific to the compositions patterns. For example, the above extracted composition is transformed into the following normal form by the fusion rules defined in Section 5.2 (the fusion rules can work well as Haskell functions to carry out the transformation).

$$\begin{array}{l} [\![\ [\ add(add(\mathtt{c_2x}(\mathtt{b0}), \mathtt{c_1x}(\mathtt{b1})), add(\mathtt{c0x}(\overrightarrow{u[0]}), \mathtt{c1x}(\overrightarrow{u[1]}))), \\ \quad add(add(\mathtt{c_2x}(\mathtt{b1}), \mathtt{c_1x}(\overrightarrow{u[0]})), add(\mathtt{c0x}(\overrightarrow{u[1]}), \mathtt{c1x}(\overrightarrow{u[2]})))], \\ \ add(add(\mathtt{c_2x}(u_{\ll 2}), \mathtt{c_1x}(u_{\ll 1})), add(\mathtt{c0x}(u), \mathtt{c1x}(u_{\gg 1}))), \\ \ [add(add(\mathtt{c_2x}(\overleftarrow{u[2]}), \mathtt{c_1x}(\overleftarrow{u[1]})), add(\mathtt{c0x}(\overleftarrow{u[0]}), \mathtt{c1x}(\mathtt{b2})))]\ ]\!] \end{array}$$

The the last phase, the system generates C++ code of efficient implementations of built normal forms, to replace the skeleton compositions with these implementations. The replacement code is written with direct use of MPI or skeletons provided by SkeTo library. Figure 6.12 shows the generated code to replace the function `nextZW` of Figure 6.11, which implements the efficient implementation of the above normal form discussed in Section 5.2.

Finally, the system outputs the input C++ program of which skeleton compositions are replaced with the generated code.

```
1  // ...
2  // definitions of parameter functions
3  struct times_t : public skeleton::unary_function<double,double>{
4         const double c;
5         times_t(double c) : c(c) {}
6         double operator()(double r) const{
7      return c * r;
8    }
9  } c_1x(c_1), c0x(c0);
10 struct ADD : public skeleton::binary_function<double,double,double>{
11   double operator()(double x, double y) const {
12     return x+y;
13   }
14 }add;
15
16 //start of skeleton program
17 dist_list<double> *nextZW(dist_list<double> *u) {
18   dist_list<double> *sl1 = list_skeletons::shiftl(b2,u);
19   dist_list<double> *sr1 = list_skeletons::shiftr(b1,u);
20   dist_list<double> *sr2 = list_skeletons::shiftr(b0,sr1);
21
22   dist_list<double> *msl1 = list_skeletons::map(c1x,sl1);
23   dist_list<double> *m0 = list_skeletons::map(c0x,u);
24   dist_list<double> *msr1 = list_skeletons::map(c_1x,sr1);
25   dist_list<double> *msr2 = list_skeletons::map(c_2x,sr2);
26
27   dist_list<double> *sz1 = list_skeletons::zipwith(add, msr2, msr1);
28   dist_list<double> *sz2 = list_skeletons::zipwith(add, m0, msl1);
29   dist_list<double> *res = list_skeletons::zipwith(add, sz1, sz2);
30
31   delete sz2;
32   delete sz1;
33   delete msr2;
34   delete msr1;
35   delete m0;
36   delete msl1;
37   delete sr2;
38   delete sr1;
39   delete sl1;
40   return res;
41 }
42 //end of skeleton program
43
44 // the main function
45 int SketoMain(int argc, char **argv) {
46   // ...
47   for(int t=0;t<count;t++){
48     tmp = nextZW(ary);
49     delete ary;
50     ary = tmp;
51   }
52   // ...
53   return 0;
54 }
```

Figure 6.11. The input source code written with skeletons in SkeTo library (some parts are omitted). The function `nextZW` corresponds to the running example *next* in Section 5.2.

```
1  dist_list<double> *nextZW(dist_list<double> *u) {
2    dist_list<double> *res = new dist_list<double>(n);
3    enum { TAG1, TAG2 };
4    const int L = 1;
5    const int R = 2;
6    const int rank = skeleton::rank;
7    const int procs = skeleton::procs;
8    const int localn = u->get_local_size(rank);
9    double *uData = &(u->at(0));
10   double *resData = &(res->at(0));
11   double *sbufl = new double[L];
12   double *rbufl = new double[R];
13   double *sbufr = new double[R];
14   double *rbufr = new double[L];
15   MPI_Request reqs1, reqr1, reqs2, reqr2;
16   if(rank!=0){
17     for(int i = 0; i < L; i++) { sbufl[i] = uData[i]; }
18     MPI_Isend(sbufl,L,MPI_DOUBLE,rank-1,TAG2,MPI_COMM_WORLD,&reqs1);
19     MPI_Irecv(rbufl,R,MPI_DOUBLE,rank-1,TAG1,MPI_COMM_WORLD,&reqr1);
20   }
21   if(rank!=procs-1){
22     for(int i = 0; i < R; i++) { sbufr[i] = uData[localn-R+i]; }
23     MPI_Isend(sbufr,R,MPI_DOUBLE,rank+1,TAG1,MPI_COMM_WORLD,&reqs2);
24     MPI_Irecv(rbufr,L,MPI_DOUBLE,rank+1,TAG2,MPI_COMM_WORLD,&reqr2);
25   }
26
27   if(rank==0) {
28     resData[0] = add(add(c_2x(b0), c_1x(b1)), add(c0x(uData[0]), c1x(uData[1])));
29     resData[1] = add(add(c_2x(b1), c_1x(uData[0])), add(c0x(uData[1]), c1x(uData[2])));
30   }
31   if(rank==procs-1) {
32     resData[localn-1-0] = add(add(c_2x(uData[localn-1-2]), c_1x(uData[localn-1-1])), add(c0x
           (uData[localn-1-0]), c1x(b2)));
33   }
34
35   for(int i = R; i < localn-L; i++) {
36     resData[i] = add(add(c_2x(uData[i-2]), c_1x(uData[i-1])), add(c0x(uData[i+0]), c1x(
           uData[i+1])));
37   }
38   if(rank!=0){ MPI_Wait(&reqr1,MPI_STATUS_IGNORE); }
39   if(rank!=procs-1){ MPI_Wait(&reqr2,MPI_STATUS_IGNORE); }
40   if(rank!=0) {
41     resData[0] = add(add(c_2x(rbufl[0]), c_1x(rbufl[1])), add(c0x(uData[0]), c1x(uData[1])))
           ;
42     resData[1] = add(add(c_2x(rbufl[1]), c_1x(uData[0])), add(c0x(uData[1]), c1x(uData[2])
           ));
43   }
44   if(rank!=procs-1) {
45     resData[localn-1-0] = add(add(c_2x(uData[localn-1-2]), c_1x(uData[localn-1-1])), add(c0x
           (uData[localn-1-0]), c1x(rbufr[0])));
46   }
47   delete [] sbufl; delete [] rbufl; delete [] sbufr; delete [] rbufr;
48   return res;
49 }
```

Figure 6.12. The output optimized code to replace the function `nextZW` of Figure 6.11.

6.3.3 Experiment Results

We will show experiment results of the domain-specific fusion developed in Section 5.2 to evaluate its effectiveness.

We measured running times of naively-composed skeleton programs and their optimized programs for the examples *next* and *solveTS* in Section 5.2 as well as the following simpler example *upwind*.

$$upwind\ u = \mathsf{zipwith}\ (+)\ (\mathsf{shift}_{\gg}\ 0\ (\mathsf{map}\ (c_1\times)\ u))\ (\mathsf{map}\ (c_0\times)\ u)$$

Here, each element of the output is computed from its corresponding element in the input u and its left element. We used a PC cluster where each of the nodes connected with Gigabit Ethernet has a CPU of Intel® Xeon®2.80GHz and 2GB memory, with Linux 2.4.21, GCC 4.1.1, and mpich 1.2.7.

Result of *upwind*

Table 6.1 lists measured running times and speedups, and Figure 6.13 shows the speedups. Running time is of applying the function 10 times to an input list of 40,000,000 elements. A speedup is a ratio of running time of a parallel program to running time of a sequential program of a single for-loop.

The program **upwind_f** optimized by the domain-independent fusion achieves two times faster running time than the original skeleton program **upwind**. The program **upwind_dsf** optimized with the domain-specific optimization achieves ten times faster running time than the original skeleton program, and the same running time as a sequential program on one processor. These improvements were gained by elimination of redundant intermediate data and communications.

Precisely saying, the domain-independent optimization reduces the number of traversals on lists from four in the naive program to two in the optimized program. Actually, the optimized program contains only one skeleton accumulate. However, the skeleton essentially traverses the list twice. Therefore, the reduction of the number of traversals is limited to two but not one. Note that the computation (multiplications of constants and additions) are far cheaper than memory access in this example program.

On the other hand, the domain-specific optimization reduces the number of traversals on lists from four to one, because it dispatches implementation specific to the computation pattern. Therefore, the domain-specific optimization achieves four times faster running time.

Also, the optimized program achieves good speedups against the number of processors. The drop of speedup is expected to be suppressed when the computation is carried out on larger input.

These results guarantee effectiveness of the proposed optimization.

Table 6.1. Running times and speedups of **upwind** (naive skeleton program), **upwind_f** (optimized by fusion), and **upwind_dsf** (optimized by DS fusion) against the number of processors. A speedup is one with respect to a sequential program.

#processors		1	2	4	8	16	32	64
upwind	time (s)	32.97	16.14	8.13	3.95	1.97	1.05	0.73
	speedup	0.26	0.53	1.05	2.15	4.31	8.14	11.66
upwind_f	time (s)	10.13	5.07	2.60	1.46	0.65	0.47	0.35
	speedup	0.84	1.68	3.27	5.85	13.00	17.94	24.03
upwind_dsf	time (s)	8.51	4.14	2.08	1.04	0.51	0.28	0.17
	speedup	1.00	2.06	4.09	8.21	16.57	30.04	49.98

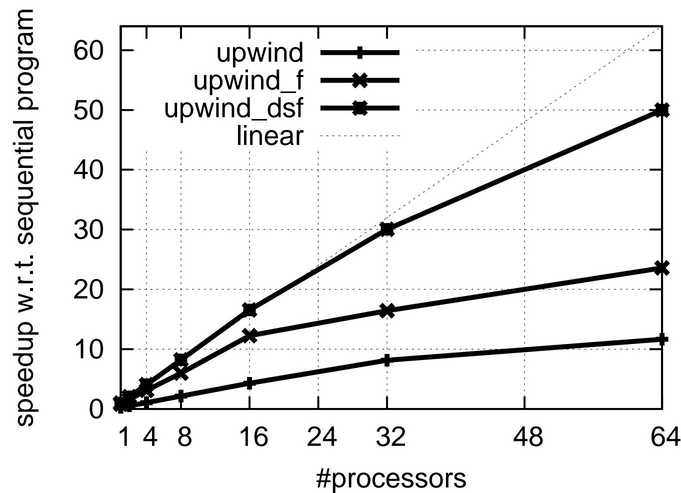


Figure 6.13. Speedups of **upwind** (naive skeleton program), **upwind_f** (optimized by fusion), and **upwind_dsf** (optimized by DS fusion) with respect to the number of processors.

```
r[0] = c_2*b0 + c_1*b1 + c0*u[0] + c1*u[1];
r[1] = c_2*b1 + c_1*u[0] + c0*u[1] + c1*u[2];
for(int i = 2; i < n-1; i++) {
  r[i] = c_2*u[i-2] + c_1*u[i-1] + c0*u[i] + c1*u[i+1];
}
r[n-1] = c_2*u[n-3] + c_1*u[n-2] + c0*u[n-1] + c1*b2;
```

Figure 6.15. A sequential program of *next*

Result of *next*

Table 6.2 lists measured running times and speedups, and Figure 6.14 shows the speedups. Running time is of applying the function 100 times to an input list of 10,000,000 elements. A speedup is a ratio of running time of a parallel program to running time of a sequential program (shown in Figure 6.15). It is worth noting that the domain-independent fusion does not work for this example, and thus we have no result for the domain-independent fusion here.

The optimized program **next_dsf** achieves ten times faster running time than the original skeleton program **next**, and the same running time as a sequential program on one processor. This improvement was gained by elimination of redundant intermediate data and communications. Precisely saying, the optimization reduces the number of traversals on lists from ten in the naive program (see Figure 6.11) to one in the optimized program. Therefore, the optimized program runs ten times faster than the naive program. Note that the computation (multiplications of constants and additions) are far cheaper than memory access in this example program.

Also, the optimized program achieves good speedups against the number of processors. The speedup drops earlier than the previous example **upwind**, because the optimized implementation of **next** performs communication twice, and thus its overhead appears earlier, while the optimized program of **upwind** performs communication once. The drop of speedup can be suppressed when the computation is carried out on larger input.

These results guarantee effectiveness of the proposed optimization.

Result of *solveTS*

Table 6.3 lists measured running times and speedups, and Figure 6.16 shows the speedups. Running time is of applying the function 10 times to an input list of 1,000,000 elements. A speedup is a ratio of running time of a parallel program to running time of a sequential program

The optimized program **solveTS_dsf** achieves about 20% faster running time than the original skeleton program **solveTS**. This improvement was gained by elimination of redundant intermediate data and communications, although its effectiveness is small because both programs have the same heavy computation of `scan`.

Also, the optimized program achieves good speedups against the number of pro-

Table 6.2. Running times and speedups of **next** (naive skeleton program) and **next_dsf** (optimized by DS fusion) against the number of processors. A speedup is one with respect to a sequential program.

#processors		1	2	4	8	16	32	64
next	time (s)	210.25	100.84	48.12	24.41	13.31	6.52	3.50
	speedup	0.09	0.20	0.41	0.81	1.49	3.04	5.67
next_dsf	time (s)	19.86	9.64	4.93	2.44	1.26	0.70	0.47
	speedup	1.00	2.06	4.03	8.14	15.79	28.26	42.22

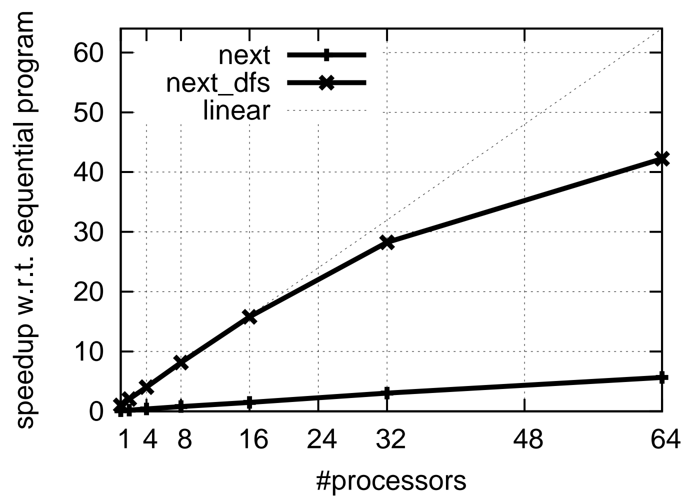


Figure 6.14. Speedups of **next** (naive skeleton program) and **next_dsf** (optimized by DS fusion) with respect to the number of processors.

Table 6.3. Running times and speedups of **solveTS** (naive skeleton program) and **solveTS_dsf** (optimized by DS fusion) against the number of processors. A speedup is one with respect to a sequential program.

#processors		1	2	4	8	16	32	64
solveTS	time (s)	117.03	69.92	40.54	20.21	10.14	4.97	2.79
	speedup 0.53	0.89	1.54	3.08	6.13	12.52	22.28	
solveTS_dsf	time (s)	62.20	60.81	30.47	15.26	7.66	3.91	2.18
	speedup	1.00	1.02	2.04	4.08	8.12	15.90	28.49

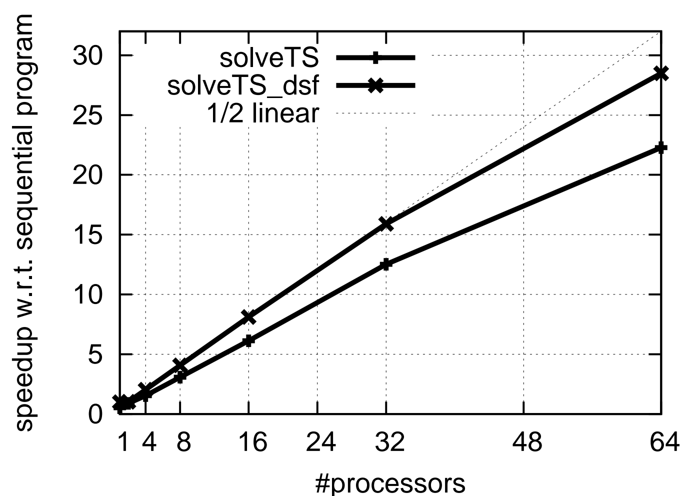


Figure 6.16. Speedups of **solveTS** (naive skeleton program) and **solveTS_dsf** (optimized by DS fusion) with respect to the number of processors.

cessors, although the maximum speedup is limited to half of the number of processors. This is because the parallel implementation of `scan` essentially doubles its computation with respect to that of sequential implementation.

6.4 Libraries with Optimization Capabilities

In this section, we will propose design and implementation of libraries with optimization capabilities, which can optimize naively written structured computations with knowledge of optimization theorems for skeleton programs. These libraries bring easy use of the optimization theorems, to solve the problem of difficulty in applying those theorems by hand, such as misjudgment of conditions, misimplementation of complicated efficient implementation, and so on.

The library consists of two collections: one collection of the core objects to abstract various data structures, and the other collection of theorems to optimize programs using the core objects. Each core object also carries out the whole com-

putation of the specific pattern on the structure, which is a quite natural design to apply our theories in which the structures of the data and the computation are closely related to each other. The core objects exploit the knowledge of theorems in the other collection, to dispatch suitable efficient implementations to programs naively written with the core objects.

We will show our idea with a development of a concrete library for nested reductions in new programming language Fortress [ACH+08]. The target computation of the library is those discussed in Section 5.3.

In the example library, we will provide a collection of objects representing nested data structures discussed in Section 5.3. For example, a code fragment to compute the maximum prefix sum is written with comprehensions as follows.

$$\texttt{BIG MAX}\,\langle\, \sum \langle\, x \mid x \leftarrow px \,\rangle \mid px \leftarrow \mathit{prefixes}\ xs \,\rangle$$

Here, all prefixes are abstracted by the core object *prefixes xs*, in which the function *prefixes* takes the input list *xs* and returns the core object. This core object can be simply seen as a nested list of prefixes. The inner comprehension $\langle\, x \mid x \leftarrow px \,\rangle$ with the reduction $\sum$ computes a sum of each prefix px, and the outer $\langle \ldots \mid px \leftarrow \mathit{prefixes}\ xs \,\rangle$ with the reduction `BIG MAX` (the maximum) takes the maximum of sums.

The example library is equipped with the theorems in Section 5.3, to optimize those programs using the core objects. For example, the library applies Theorem 5.33 to the above program to execute it with the provided efficient implementation, since the program satisfies the application condition of the theorem.

Equipped with the collection of theorems, the library enables us to exploit knowledge of theorems easily, without applying theorems by hand. This clearly reduces human errors such as misjudgment of conditions, misimplementation of complicated efficient implementation, and so on.

In the rest of this section, we will first introduce the general design of libraries with optimization capabilities. Then, to proceed to the example library for nested reductions in Fortress, we will introduce the programming language Fortress. After that, we will show design and implementation of the library for nested reductions.

6.4.1 General Design of Growable Optimizing Libraries

Figure 6.17 shows the general design of our growable optimizing libraries.

The point of the design of libraries is the two collections: One collection of the core objects is provided to describe naive structured programs of specific patterns, and the other collection of theorems is given to provide efficient implementations under specific conditions on parameters of the computations.

The core objects represent the target data structures of computations, and perform the whole computations of specific patterns on the data structures. The core object should handle the whole computation completely. This is a requirement for exploiting knowledge of our theories because the computation structures in our theories are very closely related to the structures of target data.

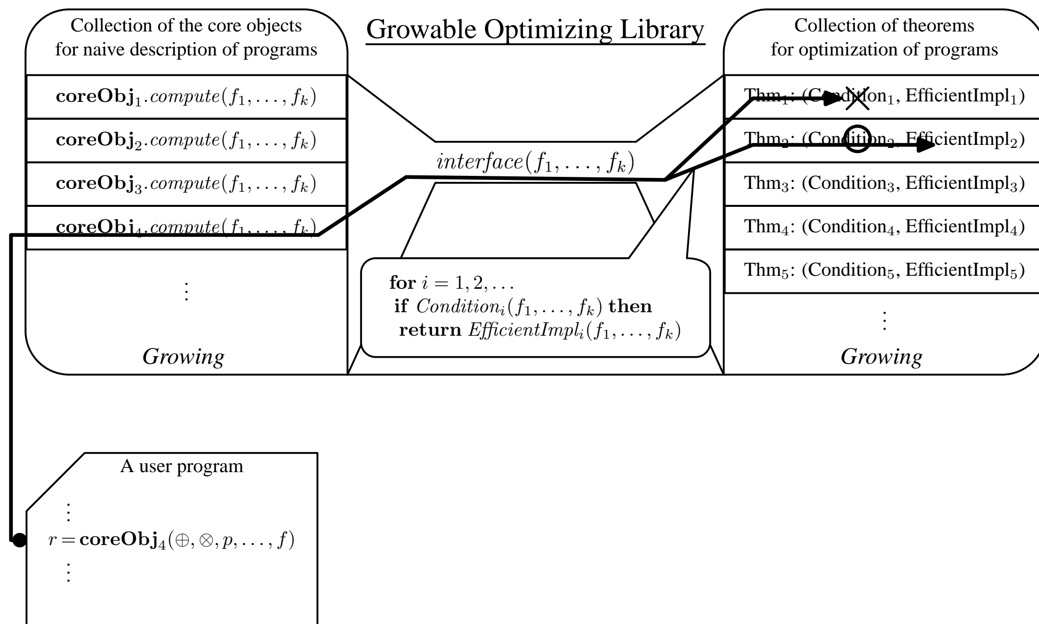


Figure 6.17. Two-collection design of growable optimizing libraries. Libraries optimize programs (specifications) naively written with the core objects. Each core object carries out the optimization in the interface method, checking application conditions of theorems and executing the computation with efficient implementations provided by applicable theorems. This structure enables easy use of knowledge of theorems.

For example, a list may be the core object, since it is a data structure, and it can handle the computation (homomorphism); the computation structure of the homomorphism on the list is completely specified by the list. Actually, this idea is used in Fortress to provide simple computation pattern on data structures.

The computation on the core objects is carried out within an interface method $\mathit{compute}(f_1, \ldots, f_k)$, which has some parameters of the computation such as functions, operators, predicates, and so on. For example, the homomorphism on a list has two parameters.

The core objects optimize the computations on their representing data structures, exploiting the collection of theorems. They dispatch suitable efficient implementations to the computations on the data structures themselves, when they find some theorems are applicable to the computation with the given parameters.

The requirement that the core objects can handle the whole computation on them is necessary to optimize the whole computation flexibly. The core objects optimize the execution of the computation by replacing the naive algorithm to compute the result with efficient implementation. Thus, they have to handle the whole computation.

The check on application conditions of theorems can be easily implemented by the check on types. Since our theories formalize computation patterns as higher order functions, the application conditions of theorems are usually described with whether the parameters have specific mathematical properties or not. Also, the test of possession of mathematical properties can be easily implemented by checking types, once we annotate the parameter objects with specific types to represent mathematical properties. Therefore, the check on application conditions can be implemented by the check on types.

Finally, it is worth noting that the library grows up by extending the collections. The expressiveness is grown by enriching the collection of core objects. The power of optimization is grown by enriching the collection of theorems.

This is our proposing design of growable optimizing libraries.

6.4.2 Parallel Programming Language Fortress

In this section we will review new parallel programming language Fortress [ACH+08]. The review starts at overviews, and proceeds to various expressions and the type system of Fortress.

Brief Overview of Fortress

Fortress is a new ambitious programming language being developed by Programming Language Research Group of Sun Microsystems Laboratories and its open source community. Some interesting features are its default support for easy use of parallelism, its philosophy of a growing language, and its programming notation closer to math.

Fortress is a parallel language with high productivity for various parallel machines, supporting parallelism at several levels. The highest level support is expressions evaluated in parallel. The next level includes loops, reductions, and comprehensions. The lower level provides parallel code regions, and explicit multithreading. In this thesis, we only use the high level support for parallelism.

Fortress is a growing language, i.e., grows over time, since we cannot build a big language all at once. Observing this growing language philosophy, Fortress puts as many language features as possible into libraries rather than the compiler, and supports extensible syntax and many kinds of operators for DSLs. We also observe the philosophy, and make a library with optimization capabilities.

Fortress supports notation closer to math. For example, it supports juxtaposition notation so that we can write $2x$ to double x instead of `2 * x` in the usual languages. Also, we can write a function application without parenthesis, e.g., $\sin x$ instead of `sin(x)`. Fortress' notation supports also many mathematical operators, and various comprehensions. Therefore, we can write $\sum\langle x^2 \mid x \leftarrow xs\rangle$ to take a square sum of xs, in which $\sum$ is an operator to take summation (a reduction with $+$) and $\langle\ldots\rangle$ is a comprehension notation.

Expressions Evaluated in Parallel

Let's consider the following tuple expression, in which f is a function and e_is are certain expressions.

$$(f(e_1, e_2), e_3 + e_4)$$

Fortress may evaluate elements of a tuple in parallel. Thus, sub expressions $f(e_1, e_2)$ and $e_3 + e_4$ may be evaluated in parallel. In addition, arguments of a function (e_1 and e_2), and operands of an operator (e_3 and e_4) may be evaluated in parallel, respectively.

Expressions to Structure Programs

Let's see some small programs to compute Fibonacci numbers.

The first program defines a recursive function *fib*.

$$
\begin{aligned}
\mathit{fib}(n : \mathbb{Z}32) : \mathbb{Z}32 = {} & \texttt{if } n \leq 1 \texttt{ then } 1 \\
& \texttt{else } (f_1, f_2) = (\mathit{fib}(n-1), \mathit{fib}(n-2)) \\
& \qquad f_1 + f_2 \\
& \texttt{end}
\end{aligned}
$$

The function *fib* takes the input n of type $\mathbb{Z}32$ (32-bit integer), and returns the result (the nth Fibonacci number) of type $\mathbb{Z}32$. The type of the input argument follows the name n of the input after a colon, and the type of the result follows the parenthesis enclosing the input arguments. The function body is the 'if' expression, which returns the last expression of the 'then' part when the condition holds, and otherwise returns the last expression of the 'else' part. Thus, the body return 1

when $n \leq 1$, and otherwise returns the sum $f_1 + f_2$, i.e., the sum of the results of the recursive calls $fib(n-1)$ and $fib(n-2)$ that are bound to variables f_1 and f_2 by the binding operator $=$. Types of f_1 and f_2 are not written in the code, and will be inferred by Fortress. It is worth noting that an 'if' expression without the 'else' part is allowed.

The following program defines function *fibL* to compute Fibonacci numbers with linear cost. Its argument type and result type are omitted here, and will be inferred automatically by Fortress.

```
fibL(n) = do (* defines a local function sub *)
            sub(m) = if  m ≤ 1 then (1, 1)
                     else (f1, f2) = sub(m − 1)
                          (f1 + f2, f1)
                     end (* end of function sub *)
            (fst, snd) = sub(n)
            fst
          end
```

The function body is the 'do' expression that executes its enclosing expressions sequentially, and returns the result of the last expression. Therefore, the function *fibL* returns the value of the variable *fst*, which is computed by the preceding expression that calls the local function *sub* computing the pair $(fib(n), fib(n-1))$. Fortress allows declaration of local functions like *sub*. A comment in Fortress programs is enclosed with (* and *).

We can also use a 'while' expression to compute Fibonacci numbers by a loop as follows.

```
fibLW(n) = do
  (f1, f2, m) : (ℤ32, ℤ32, ℤ32) := (1, 1, 1)
  while m < n do (f1, f2, m) := (f1 + f2, f1, m + 1) end
  f1
end
```

A 'while' expression evaluates its body (enclosed with do-end) repeatedly while the condition holds. The pair (f_1, f_2) holds $(fib(m), fib(m-1))$ during the iteration in this program. The variable declaration uses operator $:=$ to assign their initial values, which indicates that the variables are assignable (updatable). Since we have to use the operator $:=$ to update such assignable variables in Fortress, the body uses the operator to update the variables with their new values, instead of the binding operator $=$.

We can break an iteration of 'while' by an 'exit-with' expression. The following program code replaces the 'while' loop followed by the expression f_1 in the previous program.

```
label whileLoop
  while true do
    if m = n then exit whileLoop with f1 end
    (f1, f2, m) := (f1 + f2, f1, m + 1)
  end
end whileLoop
```

The 'exit-with' expression exits from a block labeled by the 'label' expression, and returns the expression following the 'with' as the result of the labeled block. Thus, the 'exit-with' expression of the above program exits the *whileLoop* block with the value of f_1.

Expressions to Branch on Type

The next program prints the type of the given argument, using 'typecase' expression that branches according to types.

```
printType(x) = typecase x of
                   ℤ32 ⇒ println("x is an integer.")
                   String ⇒ println("x is a string.")
                   else⇒ println("x is unknown type.")
               end
```

The 'typecase' branches to the first case of which specified type matches with the type of the given argument x. Therefore, the program prints "x is an integer." when x has type $\mathbb{Z}32$, and it prints "x is a string." when x is a string. Otherwise "x is unknown type." is printed. Of course, the dispatching based on types can also be implemented with overloaded functions.

Fortress Type System

The types in Fortress consists of traits and objects. Traits are like interfaces in Java [GJSB05], but may contain code. Objects may have methods and fields, while traits may not have fields. In that sense, objects are the actual data. The name of a trait or an object may be used as a type. Traits and objects may extend multiple traits, but may not extend any objects. In other words, objects are the leaves of the hierarchy. Traits, objects, and methods may be parameterized. The parameters may be types or compile-time constants.

Figure 6.18 shows some Fortress codes of traits and objects, of which functionalities will be explained in Section 6.4.3. Trait Generator is parameterized with type E (enclosed with $[\![$ and $]\!]$), and has methods *generate* and *reduce*. The method *generate* is also parameterized with another type R, and takes two arguments of types Reduction$[\![R]\!]$ (defined below) and $E \to R$. Here, $E \to R$ is the type of functions from E to R, and functions are first-class in Fortress. The other method *reduce* has its body definition. Trait Reduction is parameterized with type L, and

```
trait Generator⟦E⟧
      generate⟦R⟧(red: Reduction⟦R⟧, body: E → R): R
      reduce(r: Reduction⟦E⟧) : E = generate⟦E⟧(r, fn (e : E) ⇒ e)
end
trait Reduction⟦L⟧
      empty(): L
      join(a: L, b: L): L
end
trait SomeCommutativeReduction end
object SumReduction
   extends { Reduction⟦Number⟧, SomeCommutativeReduction }
      empty(): Number = 0
      join(a: Number, b: Number): Number = a + b
end
```

Figure 6.18. Traits for generators and reductions.

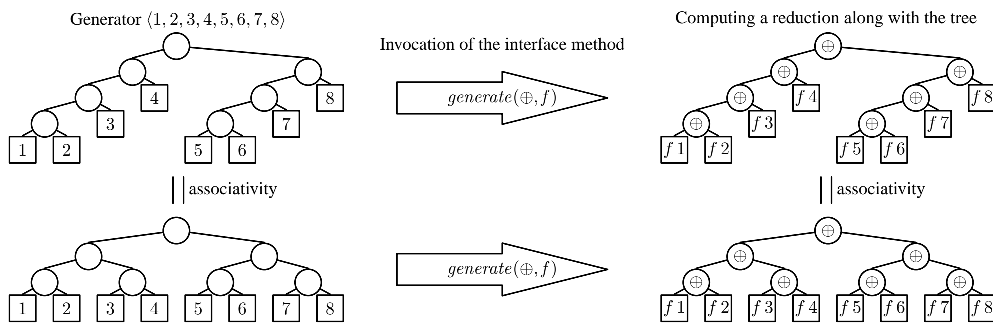


Figure 6.19. Generators abstract parallel computation where a set of elements (data) is generated and then consumed by a reduction. A generator forms an abstract computational tree. The associativity of reduction operators plays an important role in balancing the parallel computation.

has two (abstract) methods without their bodies. Object SumReduction extends two traits Reduction with the type parameter L = Number (a trait for numbers) and SomeCommutativeReduction. It also defines bodies of the abstract methods.

6.4.3 Generators for Reductions in Fortress

This section reviews the concept of generators in Fortress, which is the most important idea of high-level parallelism in Fortress. We will also shows its implementation briefly.

Basically, generators are objects that can generate a set of elements, and perform parallel reduction on them. In other words, generators can be seen as the combination of list data structures and list homomorphism on them. Thus, our developed theories can be applicable for them.

Later in the next section, we will extend generators to handle nested computation on nested structures.

Concept of Generators in Fortress

Generators in Fortress [ACH+08] are objects to abstract high-level parallel computation, in which a set of elements are generated and then consumed by a reduction like a summation. A generator is a container holding a set of elements, and can be seen as a higher order function in the sense that it provides a pattern of computation with function parameters. Actually, the computation pattern is the same as list homomorphism in Chapter 2.

Figure 6.19 shows how a generator abstracts such parallel computation. A generator forms an abstract tree structure for parallel computation, in which elements are held in leaves connected by internal nodes. For example, generator $\langle 1,2,3,4,5,6,7,8\rangle$ holds eight elements on the tree as shown in the top left of Figure 6.19, in which elements 1, 2, 3, and 4 are held in the left sub-tree and the others are in the right sub-tree.

It is easily seen that the abstract structure is the same as the structure of join lists. Also, the structure is the same as the computation structure of homomorphism.

A generator receives the pair of an associative binary operator and a function through the interface method *generate*, to perform the computation along with the abstract tree structure. The abstract tree structure is filled with the operator and the function, to become a concrete computational tree, in which a leaf becomes an application of the function to generate a new element, and an internal node is filled with the binary operator. For example, the top right tree in Figure 6.19 shows one filled with the pair $(\oplus, f)$. The computation performed through $\mathit{generate}(\oplus, f)$ of generator $\langle 1,2,3,4,5,6,7,8\rangle$ results in $f\,1 \oplus f\,2 \oplus f\,3 \oplus f\,4 \oplus f\,5 \oplus f\,6 \oplus f\,7 \oplus f\,8$, which is the same as homomorphism.

Changing the operator and the function to be supplied, we can compute various results from a generator. For example, the sum of squares of the elements are obtained by the pair of addition operator $+$ and function $f(x) = x^2$. Also, a product of all elements are computed by the product operator and the identity function.

A generator executes the computation in parallel. For example, two sub-trees in the computational tree shown in the top right of Figure 6.19 are executed in parallel, because their computations are completely independent. This parallelization corresponds to putting parenthesis as $(((f\,1 \oplus f\,2) \oplus f\,3) \oplus f\,4) \oplus (((f\,5 \oplus f\,6) \oplus f\,7) \oplus f\,8)$.

The parallelism of the computation is completely controlled by a generator, in which the associativity of operators plays an important role in balancing the parallel computation. For example, the abstract tree in the top left of Figure 6.19 can be changed to the tree in bottom left of the figure, so that the grandchild sub-trees can be evaluated in parallel evenly. The computation on the changed abstract tree shown in the bottom right results in the same value as the previous computation of the

```
generate⟦R⟧(red : Reduction⟦R⟧, body: E → R): R =
  if s ≤ 0 then red.empty()
  else loop′(lo : E, hi : E): R =
         if lo = hi then body(lo)
         else (∗ Identify power-of-2 boundary in region, split there ∗)
              split = partitionL((lo BITXOR hi) + 1)
              mid = hi BITAND (BITNOT(split − 1))
              red.join(loop′(lo, mid − 1), loop′(mid, hi))
         end
    loop′(b, b + s − 1)
  end
```

Figure 6.20. Simple parallel implementation of the method generate.

top right tree, since the operator has associativity, i.e., $(((f\,1 \oplus f\,2) \oplus f\,3) \oplus f\,4) = ((f\,1 \oplus f\,2) \oplus (f\,3 \oplus f\,4)$ and so on.

Comprehension notation is provided in Fortress for concise use of generators. Roughly speaking, the following comprehension with a reduction is equivalent to the invocation of $generate(\oplus, f)$ on a generator g.

$$\bigoplus \langle f\,a \mid a \leftarrow g \rangle$$

Here, the comprehension $\langle f\,a \mid a \leftarrow g \rangle$ specifies generation of elements by applying the function f to each element a held in the generator g, and big operator $\bigoplus$ specifies a reduction with $\oplus$ on the generated elements.

For example, to take the summation of squares of list $\langle 1, 2, 3, 4 \rangle$, we can use the following comprehension.

$$\sum \langle a^2 \mid a \leftarrow \langle 1, 2, 3, 4 \rangle \rangle$$

This comprehension is equivalent to the following skeleton program.

$$\mathsf{reduce}\ (+)\ (\mathsf{map}\ (\lambda a.a^2)\ [1, 2, 3, 4])$$

Comprehensions in Fortress can handle predicates to filter generated elements. For example, $\sum \langle a \mid a \leftarrow x, a > 0 \rangle$ results in a sum of positive elements by using predicate $a > 0$. This is equivalent to skeleton program $\mathsf{reduce}\ (+)\ (\mathsf{filter}\,(\lambda a.a > 0)\,x)$. However, we omit handling of predicates for simplicity, until Section 6.4.5.

Implementation of Generators in Fortress

A generator in Fortress is an object extending trait Generator with the interface method *generate*. Its core definition is shown in Figure 6.18. The method *generate* takes an associative binary operator *red* and a function *body*, to perform the generation and the reduction.

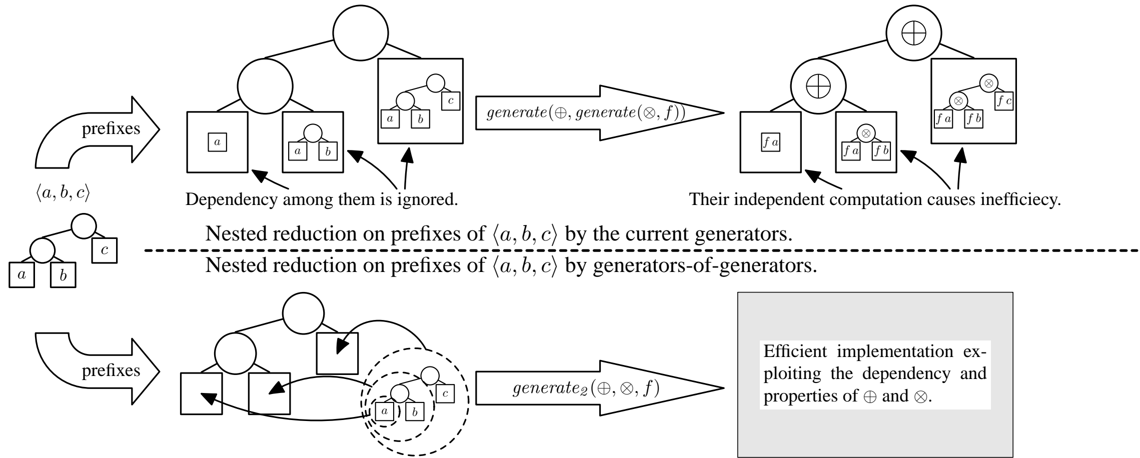


Figure 6.21. Computation of a nested reduction by the current generators and by the GoGs.

An associative binary operator is enclosed in an object extending trait Reduction. Trait Reduction has method *join* to provide an associative binary operator, and method *empty* to provide the identity of the operator.

For example, the usual addition operator $+$ is enclosed with object SumReduction shown in Figure 6.18, in which the method *join* adds two operands by the operator $+$, and the method *empty* returns 0, i.e., the identity of $+$.

Figure 6.20 shows an example parallel implementation of *generate* method of generator $(b \,\#\, s)$ that produces a sequence of s numbers from b, i.e., $b, b+1, \ldots, b+s-1$. When $s \leq 0$, i.e., the generator produces no elements, the method returns the identity $red.empty()$. Otherwise, it forms an abstract binary tree by recursive calls of the local function $loop'$ that performs partial computation on lo, $lo+1$, ..., hi. In the case of $lo = hi$, i.e., a leaf of lo, it generates a new element by $body(lo)$. In the other case, i.e., an internal node, it combines the results $loop'(lo, mid-1)$ and $loop'(mid, hi)$ of sub-trees by the given associative binary operator $red.join$. The results of sub-trees are computed in parallel, since operands are computed in parallel by default in Fortress. The structure of the abstract computational tree varies according to the number of elements to be generated. The associativity of the given operator is necessary to obtain a well-defined result of the computation.

Finally, we briefly mention desugaring of comprehensions in Fortress. Each use of comprehension is desugared into invocation of the method *generate* of a generator by the following desugaring algorithm $\mathcal{DS}$.

$$\begin{aligned}
&\mathcal{DS}(\textstyle\bigoplus\langle body \mid x \leftarrow g, gs\rangle) \\
&\qquad\qquad = g.generate(\oplus, \mathbf{fn}\ x \Rightarrow \mathcal{DS}(\textstyle\bigoplus\langle body \mid gs\rangle)) \\
&\mathcal{DS}(\textstyle\bigoplus\langle body \mid \rangle) = body
\end{aligned}$$

For example, the code $\sum\langle x^2 \mid x \leftarrow xs\rangle$ to compute a sum of squares is desugared by the compiler into the following invocation of *generate*. Here, the reduction operator

object SumReduction is given by the operator $\sum$.

$$xs.generate(\text{SumReduction}, \textbf{fn}\ x \Rightarrow x^2)$$

6.4.4 Generators-of-generators: the Core of Nested Reductions

We will introduce the concept of generators-of-generators (*GoG*s for short) to represent nested data structures of specific patterns, which is the core objects for nested reductions. Before introducing GoGs, we will first point out the problem of using the current simple generators.

Problem of Using the Simple Generators

The requirement for the core objects is that they can handle the whole computation on them, i.e., the whole computation of nested reductions, so that they can optimize the whole computation flexibly. They optimize the execution of the computation by replacing the naive algorithm to compute the result with efficient implementation. Thus, they have to handle the whole computation.

However, the current generators cannot satisfy the requirement. This is mainly because they are designed to handle only flat reductions on them. Even though their generating elements have dependencies on each other, and thus the reductions on them have the induced dependencies, these dependencies are completely ignored and the whole computation of nested reductions cannot be handled with one generator.

Consider computing nested reductions on dependent data structures such as prefixes of a sequence. With the existing generators, nested reductions on prefixes of generator $xs = \langle a, b, c\rangle$ with two operators $\oplus$ and $\otimes$ are computed by the following code.

$$\bigoplus\langle \bigotimes\langle f\ xs_i \mid i \leftarrow (0 \,\#\, s) \,\rangle \mid s \leftarrow (1 \,\#\, (|xs|))\,\rangle$$

Here, $|xs|$ is the size of xs. The top half of Figure 6.21 illustrates the computation, in which leaves of the big abstract computational tree are (virtually) abstract trees of prefixes $\langle a\rangle$, $\langle a, b\rangle$, and $\langle a, b, c\rangle$.

The problem is that the outer tree generates the inner abstract trees independently in its leaves, and thus inner computations are independently executed in the leaves, even though the inner trees have the dependency that they are prefixes of one generator xs.

To solve the problem, we will introduce generators-of-generators, which abstracts nested data structures.

Generators-of-Generators to Capture the Whole Computation of Nested Reductions

To solve the problem of the current generators ignoring inner generators, we will introduce generators-of-generators, which abstracts nested data structures. They

capture dependencies among generated generators such as prefixes, and abstract the whole structure of a nested reduction on them. Therefore, we can have a big chance to perform efficient computation exploiting the dependency and properties of operators.

GoGs are more-informed generators for efficient computation of nested reductions of the following form.

$$\bigoplus\langle \bigotimes\langle\, f\ y \mid y \leftarrow ys\rangle \mid ys \leftarrow gg\,\rangle$$

Here, gg is a GoG to produce a set of dependent generators (yss). In the rest of this chapter, we will especially focus on a GoG such that a generated generator is a subset of a given generator.

The computation of a nested reduction with a GoG is illustrated in the bottom of Figure 6.21. A GoG is also a generator, and forms a similar nested abstract computational tree. However, a GoG abstracts the whole structure of a nested reduction by *one* object, while the existing generators do by *multiple* objects (i.e., the outer generator and the inner generators).

GoGs have the new interface method $generate_2$ to execute nested reductions with the given operators, so that they can exploit mathematical properties between the given operators and the dependencies among inner generators for optimization based on theorems. The new interface receives two reduction operators at the same time, while the computation by the existing generators supply these operators separately to the outer generator and to the inner generators. This point is important, because GoGs are often required to handle these two operators simultaneously to exploit knowledge of theorems.

6.4.5 Design and Implementation of the Optimizing GoG Library

We will implement the growable optimizing library for nested reductions, based on the general design in Section 6.4.1.

The main features of the library are two folds.

- It provides a set of GoGs so that users can easily write programs for nested reductions with various patterns of dependencies. As demonstrated later, various interesting applications can be clearly specified with the GoGs the library provides.
- It, like other optimization techniques based on simple calculation rules [GLJ93, OHIT97], implements the optimization lightweightly deep analysis of program codes and can be implemented as a Fortress library. The simplicity and the nontrivial improvement are owing to the use of calculation theorems and the powerful dispatch mechanism of Fortress.

Our optimizing library for nested reductions has been integrated into the current Fortress interpreter [For08].

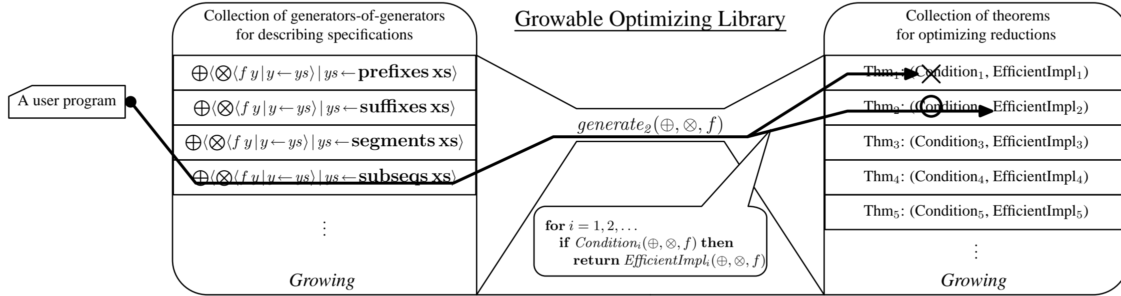


Figure 6.22. Two collections form a growable optimizing library. It optimizes specifications using knowledge of theorems on them.

Figure 6.22 shows the structure of the library, which is an instance of the design in Section 6.4.1. The core objects in the library are GoGs.

The library supports nested reductions of the following form with various GoG gg.

$$\bigoplus\langle \bigotimes\langle f\ y \mid y \leftarrow ys\rangle \mid ys \leftarrow gg \rangle$$

The behavior of programs written with our library is as follows. A nested reduction written a GoG is desugared into an invocation of the interface method $generate_2$ with the given parameters, i.e, two reduction operators and a function applied to each element. Then, the method $generate_2$ checks whether the given parameters satisfy the application conditions of theorems stored in the library. Once the condition of a theorem is found to be satisfied, the computation of the nested reduction is carried out with the efficient implementation the theorem provides. If no theorem is available, the nested reduction is carried out with the naive implementation, in which all dependent generators are actually generated independently and inner reductions on them are performed also independently. This mechanism of dispatching implementation enables users to exploit knowledge of theorems implicitly.

The optimization mechanism by dispatching needs to check mathematical properties of operators and functions to utilize calculation theorems. Generally, it is very difficult to prove a mathematical property at compile-time or, of course, run-time. Thus, we assume that operators and functions are beforehand annotated about their specific properties with specific types. The dispatching and checking mechanisms are explained in Section 6.4.5.

Mechanism of Growable Optimizing Library

We show the details of the implementation of our library, with examples of a GoG for prefixes.

We will first show the mechanism with the implementation for one-dimensional data structures, i.e., for lists. After the explanation of the mechanism, we will show the implementation for two-dimensional arrays, i.e., the implementation for the bide-tree homomorphism.

```
trait Generator2⟦E⟧ extends {Generator⟦Generator⟦E⟧⟧}
  getter seed() : Generator⟦E⟧
  generate_2⟦R⟧(q : Reduction⟦R⟧, r : Reduction⟦R⟧, f : E → R) : R
  theorems⟦R⟧() : List⟦((Reduction⟦R⟧, Reduction⟦R⟧, E → R) → Boolean,
                        (Reduction⟦R⟧, Reduction⟦R⟧, E → R) → R)⟧ = ⟨ ⟩
  naiveImpl⟦R⟧(q : Reduction⟦R⟧, r : Reduction⟦R⟧, f : E → R) : R =
                        generate⟦R⟧(q, (fn (x) ⇒ x.generate⟦R⟧(r, f)))
end
```

Figure 6.23. Base trait of GoGs

Base Trait of GoGs: Generator2

Figure 6.23 shows the base trait Generator2 of all GoGs. Since a GoG is a generator producing generators, it extends Generator⟦Generator⟦E⟧⟧, although it does not necessarily actually do the generation.

Since a GoG produces a set of generators that are subsets of a given generator, trait Generator2 has a getter *seed*() to hold the given generator. The given generator is used by efficient implementations to exploit the parallelism in the generator. Also, it is used by naive implementations to produce the naive nested computation structure.

The interface method $generate_2$ receives two objects of reduction operators and a function to be applied to every element, to perform the nested reduction with the given parameters. It implements the dispatching mechanism to optimize nested reductions using calculation theorems, which is explained in Section 6.4.5.

Knowledge of theorems on a GoG is stored as the list *theorems* of pairs of functions. Each pair of the list consists of (1) a function to check the application condition of a theorem, and (1) a function to execute efficient implementation provided by the theorem. These functions take the same arguments of the method $generate_2$, and return a Boolean value of satisfiability of the condition and the result of the nested reduction with the given parameters, respectively. The list is empty by default, and should be enriched by library implementers to utilize knowledge of theorems (see Section 6.4.5).

Method *naiveImpl* provides the naive implementation to perform the nested reduction when no theorems is available. It invokes the method *generate* with the outer reduction operator and a function that invokes the method *generate* of an inner generated generator with another operator and the function, which is the naive execution shown in the top half of Figure 6.21.

For example, Figure 6.24 shows the least implementation of a GoG for prefixes, which corresponds to inits in Section 5.3. The least requirement for defining a GoG is to implement the getter *seed*() and the method *generate*, which is an abstract method of trait Generator. The object Prefixes receives a generator of which prefixes are to be generated, and stores it in the getter *seed*(). The implementation of the method *generate* actually produces all prefixes by function *prefixesImpl*, and then

```
object Prefixes⟦E⟧(g : Generator⟦E⟧) extends Generator2⟦E⟧
  getter seed() : Generator⟦E⟧ = g
  generate⟦R⟧(r : Reduction⟦R⟧, body : Generator⟦E⟧ → R) : R
    = prefixesImpl⟦E⟧(g).generate⟦R⟧(r, body)
end
prefixesImpl⟦E⟧(x : Generator⟦E⟧) : Generator⟦Generator⟦E⟧⟧
  = ⟨ ⟨ x_i | i ← (0 # s) ⟩ | s ← (1 # (|x|)) ⟩
prefixes⟦E⟧(g : Generator⟦E⟧) : Prefixes⟦E⟧ = Prefixes⟦E⟧(g)
```

Figure 6.24. The least implementation of a GoG for prefixes.

```
generate_2⟦R⟧(q : Reduction⟦R⟧, r : Reduction⟦R⟧, f : E → R) : R
  = do ths = theorems⟦R⟧()
       i : ℤ32 := 0
       label dispatchingLoop
         while i < |ths| do
           (condition, efficientImpl) = ths_i
           if (condition(q, r, f)) then
             (* use efficient implementaion *)
             exit dispatchingLoop with efficientImpl(q, r, f)
           end
           i += 1
         end
         naiveImpl⟦R⟧(q, r, f)(* use naive implementaion *)
       end dispatchingLoop
     end
```

Figure 6.25. The implementation of the dispatching mechanism.

invokes the method *generate* of the generated prefixes. This is the required behavior of the GoG as a Fortress' generator to produce prefixes.

Function *prefixesImpl* implements the actual generation of prefixes, in which a list of prefixes is generated by the comprehension, where $\langle x_i \mid i \leftarrow (0 \# s)\rangle$ is the sth prefix of x. The last function *prefixes* is a short cut to make a GoG for prefixes of the given generator.

Dispatching Mechanism in Method $generate_2$

Method $generate_2$ implements the dispatching mechanism to optimize nested reductions with calculation theorems, which is the most important function of the library. Figure 6.25 shows implementation of the method, which takes the parameters of the nested reductions, i.e., two objects q and r of reduction operators and a function f applied to every element.

The main loop in method $generate_2$ checks each application condition *condition* of theorems stored in the list *theorems*. When the condition of a theorem is satisfied

```
trait DistributesOver⟦E⟧ end

distributes⟦Q, R⟧(q : Q, r : R) : Boolean =
  typecase (q, r) of
    (Q, DistributesOver⟦Q⟧) ⇒ true
    else⇒ false
  end
```

Figure 6.26. A trait for annotation about distributivity and a function to check the distributivity.

```
object SumReduction
  extends { Reduction⟦Number⟧, SomeCommutativeReduction,
  DistributesOver⟦MaxReduction⟧, DistributesOver⟦MinReduction⟧ }
    empty(): Number = 0
    join(a: Number, b: Number): Number = a + b
end
```

Figure 6.27. An annotated reduction object for summation.

by the given parameters q, r, and f, the dispatching loop carries out the nested reduction with the efficient implementation *efficientImpl* of the theorem, and exits the dispatching loop with the computed result. If there is no theorem available, the naive implementation *naiveImpl* is used to get the result. The current dispatching loop assumes that a theorem appearing earlier provides more efficient implementation.

The check on application conditions is performed by checking annotations on types of the objects. For example, Figure 6.26 shows a trait DistributesOver⟦E⟧ to annotate a reduction object about its distributivity. When an operator represented by an object of type R has distributivity over another operator represented by an object of type Q, R extends the trait DistributesOver⟦Q⟧ to inform the library of its possessing the property. Then, a function *distributes* checks possession of the distributivity by using `typecase` expression. Other properties can be annotated similarly using traits.

Figure 6.27 shows reduction object SumReduction annotated with the trait for the distributivity. Since the addition operator distributes over the maximum operator and the minimum operator, the object extends DistributesOver⟦MaxReduction⟧ for distributivity over the maximum, and DistributesOver⟦MinReduction⟧ for distributivity over the minimum.

Using such annotations on objects, we can successfully implement knowledge of theorems in the library. Figure 6.28 shows example implementation of the combination of Theorem 5.33 and Lemma 5.31 on prefixes. Its translation for comprehension notation is as follows.

```
conditionDistributive⟦R⟧(q : Reduction⟦R⟧, r : Reduction⟦R⟧, f : E → R) : Boolean
    = distributes(q, r)

efficientImplDistributive⟦R⟧(q : Reduction⟦R⟧, r : Reduction⟦R⟧, f : E → R) = do
  join(x : (R, R), y : (R, R)) : (R, R) = do (i1, s1) = x
                                             (i2, s2) = y
                                             (q.join(i1, r.join(s1, i2)), r.join(s1, s2))
                                           end
  zero1 = (q.empty(), r.empty())
  wrap(a : R) : (R, R) = (a, a)
  unwrap(a : (R, R)) = do (r1, r2) = a; r1 end
  unwrap(seed().generate⟦(R, R)⟧(makeReduction⟦(R, R)⟧(join, zero1), wrap ∘ f))
end

theorems⟦R⟧() = ⟨ (fn (r, q, f) ⇒ conditionDistributive⟦R⟧(r, q, f),
                   fn (r, q, f) ⇒ efficientImplDistributive⟦R⟧(r, q, f)) ⟩
```

Figure 6.28. A method to check the condition of Lemma 6.4 and a method implementing the provided efficient implementation. A pair of the two methods is added to the list *theorems* in the GoG object of prefixes, so that the library can exploit its knowledge.

Theorem 6.4. Provided that $\otimes$ distributes over $\oplus$, the following equation holds.

$$\begin{aligned}
&\bigoplus\langle \bigotimes\langle f\ y \mid y \leftarrow ys\rangle \mid ys \leftarrow \mathit{prefixes}\ xs\rangle \\
&\quad = \mathit{fst_2}\ (\bigodot_p\langle\ (f\ x, f\ x) \mid x \leftarrow xs\rangle) \\
&\qquad \textbf{where}\quad (i_1, s_1) \odot_p (i_2, s_2) = (i_1 \oplus (s_1 \otimes i_2), s_1 \otimes s_2) \\
&\qquad\qquad\qquad \mathit{fst_2}\ (a, b) = a \qquad \Box
\end{aligned}$$

The library implements the theorem as the pair of methods: *conditionDistributive* to check its application condition, and *efficientImpDistributive* to perform its providing efficient implementation. Since the condition is that the operator $\otimes$ is distributive over $\oplus$, the method *conditionDistributive* uses the function *distributes* to know whether the operator has the distributivity. The method *efficientImpDistributive* implements the efficient implementation straightforwardly, in which it uses direct invocation of method *generate* instead of the comprehension, and the function *makeReduction* encloses the new operator (*join*) and its identity (*zero₁*) in an object. Finally, to use the knowledge of the theorem in the library, the pair is added to the list *theorems*.

It is easily seen that the same functionality to dispatch efficient implementation based on theorems is implementable for other domains of computations.

Extension of Desugaring Process

We extend the desugaring process of comprehensions in Fortress to handle GoGs in nested reductions. The following rules are added to the existing desugaring algo-

Target specification of nested reductions	Application condition	Efficient implementation.
$\bigoplus\langle\bigotimes\langle f\,y \mid y \leftarrow ys\rangle \mid ys \leftarrow \mathit{prefixes}\ xs\rangle$	$\otimes$ distributes over $\oplus$	$\mathit{fst}_2\ (\bigodot_p\langle(f\,x, f\,x) \mid x \leftarrow xs\rangle)$
$\bigoplus\langle\bigotimes\langle f\,y \mid y \leftarrow ys\rangle \mid ys \leftarrow \mathit{suffixes}\ xs\rangle$	$\otimes$ distributes over $\oplus$	$\mathit{fst}_2\ (\bigodot_s\langle(f\,x, f\,x) \mid x \leftarrow xs\rangle)$
$\bigoplus\langle\bigotimes\langle f\,y \mid y \leftarrow ys\rangle \mid ys \leftarrow \mathit{segments}\ xs\rangle$	$\otimes$ distributes over $\oplus$, $\oplus$ is commutative	$\mathit{fst}_4\ (\bigodot_t\langle(f\,x, f\,x, f\,x, f\,x) \mid x \leftarrow xs\rangle)$
$\bigoplus\langle\bigotimes\langle f\,y \mid y \leftarrow ys\rangle \mid ys \leftarrow \mathit{prefixes}\ xs, p\ ys\rangle$	$\otimes$ distributes over $\oplus$, p is relational	$\mathit{fst}_4\ (\bigodot_{p'}\langle(f\,x, f\,x, x, x) \mid x \leftarrow xs\rangle)$
$\bigoplus\langle\bigotimes\langle f\,y \mid y \leftarrow ys\rangle \mid ys \leftarrow \mathit{suffixes}\ xs, p\ ys\rangle$	$\otimes$ distributes over $\oplus$, p is relational	$\mathit{fst}_4\ (\bigodot_{s'}\langle(f\,x, f\,x, x, x) \mid x \leftarrow xs\rangle)$
$\bigoplus\langle\bigotimes\langle f\,y \mid y \leftarrow ys\rangle \mid ys \leftarrow \mathit{segments}\ xs, p\ ys\rangle$	$\otimes$ distributes over $\oplus$, $\oplus$ is commutative, p is relational	$\mathit{fst}_6\ (\bigodot_{t'}\langle(fx, fx, fx, fx, x, x) \mid x \leftarrow xs\rangle)$

Table 6.4. A part of the collection of the composed theorems written in comprehension notation.

rithm.

$$\mathcal{DS}(\bigoplus\langle\bigotimes\langle f\ x \mid x \leftarrow xs\rangle \mid xs \leftarrow gg\rangle) = gg.\mathit{generate}_2(\oplus, \otimes, f)$$

$$\mathcal{DS}(\bigoplus\langle \mathit{body} \mid x \leftarrow gg, p\ x, gs\rangle) = \mathcal{DS}(\bigoplus\langle \mathit{body} \mid x \leftarrow gg.\mathit{filter}(p), gs\rangle)$$

The first rule desugars a nested reduction with a GoG into an invocation of the method *generate*$_2$ of the GoG. This rule connects the comprehension notation to the optimization mechanism of GoGs.

The second rule squeezes a filter with a predicate p into the preceding GoG gg, to result in another GoG $gg.\mathit{filter}(p)$ that generates only filtered generators of those the original produces. Each GoG has the method *filter* to take a predicate and returns such another GoG, although it is not discussed in the implementation of the GoG in the previous section for simplicity. The extension of the optimization mechanism to involve predicates is straightforward.

It is worth noting that the beta specification of Fortress supports library-level extensions of syntax for domain-specific languages. We would be able to use the feature to implement the desugaring process completely in our library, although the current extension is implemented in the interpreter. It would be noted that the desugaring process is independent from the optimization mechanism; it simply supports easy description of nested reductions.

Growing the Library

The library can improve its expressiveness and optimization power by enriching the collections of GoGs and theorems.

We have added GoGs corresponding to the generation functions shown in Section 5.3. Some functions have alternative familiar names: *prefixes* for inits, *suffixes* for tails, *segments* for *segs*, *subseqs* for subs, and *subrecs* for *rects*$'$.

Also, we have added theorems (lemmas) shown in Section 5.3, in which some theorems are implemented as combinations of two or more theorems. Some of implemented theorems are shown in Table 6.4.

```
trait GeneratorAT⟦E⟧
  generateAT⟦R⟧(op : Reduction⟦R⟧, ot : Reduction⟦R⟧, body: E → R) : R
end
```

Figure 6.29. The base trait of generators for two-dimensional data structures. The trait defines the interface method for the computation of the abide-tree homomorphism.

Implementation of GoGs for Two-dimensional Arrays

Now, we will briefly show the implementation of the abide-tree homomorphism in GoGs for two-dimensional arrays.

Figure 6.29 shows the base trait of generators for two-dimensional data structures. The trait defines the interface method for the computation of the abide-tree homomorphism. The interface method takes two reduction operators: one is for the vertical direction and the other is for the horizontal direction.

Figure 6.30 shows concrete implementation of the abide-tree homomorphism, in which the index spaces are divided into smaller ones in both horizontal and vertical directions to get the sub-results, and the sub-results are combined by the given operators. The wrapper object is prepared to perform the abide-tree homomorphism on usual arrays not extending the trait.

Figure 6.31 shows the base trait of GoGs for two-dimensional data structures. The new interface method receives four reduction operators: the first two for the outer reduction, and the last two for the inner reductions. The trait has the list of theorems, which is similar to the list in the trait Generator2, except for the number of reduction operators.

The dispatching process is implemented similar to that of the method $generate_2$.

Since comprehension notation for the abide-tree homomorphism is not prepared in Fortress, users are required to invoke the interface method directly to perform the abide-tree homomorphism. Fortunately, we can use the current comprehension notation if we use the same operators in both directions. For example, the maximum rectangle sum problem is one example of such computation. In this case, the interface method $generate_2$ is invoked, and we can dispatch efficient implementation in this method as shown in the previous section. It is an interesting task to design comprehension notation for the abide-tree homomorphism.

6.4.6 Programming with the Library

We will show example programs written with the library.

Figure 6.32 shows an example Fortress program to compute the maximum p-segment sum with the GoG library. The program imports the component Generator2 to use GoGs, and also imports List to use comprehensions. The main function *run* first makes a one-dimensional array *xs* of 400 elements, and two relational predicates

```
object GeneratorATWrapper⟦E, nat b₁, nat s₁, nat b₂, nat s₂⟧(x : Array2⟦E, b₁, s₁, b₂, s₂⟧)
  extends GeneratorAT⟦E⟧
  generateAT⟦R⟧(op : Reduction⟦R⟧, ot : Reduction⟦R⟧, body: E → R) : R = do
    loop′(lo₁ : E, hi₁ : E, lo₂ : E, hi₂ : E) =
      if lo₁ = hi₁ ∧ lo₂ = hi₂ then body(x[(lo₁, lo₂)])
      elif lo₁ = hi₁ then
          split₂ = partitionL((lo₂ BITXOR hi₂) + 1)
          mid₂ = hi₂ BITAND (BITNOT(split₂ − 1))
          ot.join(loop′(lo₁, hi₁, lo₂, mid₂ − 1), loop′(lo₁, hi₁, mid₂, hi₂))
      elif lo₂ = hi₂ then
          split₁ = partitionL((lo₁ BITXOR hi₁) + 1)
          mid₁ = hi₁ BITAND (BITNOT(split₁ − 1))
          op.join(loop′(lo₁, mid₁ − 1, lo₂, hi₂), loop′(mid₁, hi₁, lo₂, hi₂))
      else
          split₂ = partitionL((lo₂ BITXOR hi₂) + 1)
          mid₂ = hi₂ BITAND (BITNOT(split₂ − 1))
          split₁ = partitionL((lo₁ BITXOR hi₁) + 1)
          mid₁ = hi₁ BITAND (BITNOT(split₁ − 1))
          op.join(ot.join(loop′(lo₁, mid₁ − 1, lo₂, mid₂ − 1), loop′(lo₁, mid₁ − 1, mid₂, hi₂)),
                  ot.join(loop′(mid₁, hi₁, lo₂, mid₂ − 1), loop′(mid₁, hi₁, mid₂, hi₂)))
      end
    loop′(b₁, b₁ + s₁ − 1, b₂, b₂ + s₂ − 1)
  end
  end
```

Figure 6.30. Implementation of the abide-tree homomorphism with two operators. The wrapper object is prepared to perform the computation on the usual arrays.

flat_4 and *ascending* (Section 5.3.3) by specifying their relations to the predicate-generation function *relationalPredicate*. Then, it computes the maximum flat_4-*ascending*-segment sum of xs, generaing all segments by the provided function *segs*, filtering the segments by the predicates, taking sums of segments by the reduction $\sum$, and taking the maximum of sums by the reduction with BIG MAX. Here, the program explicitly specifies the static type parameters ⟦Number⟧ of the operators and the comprehension, and $\sum[\![\text{Number}]\!] ys$ is the abbreviation of $\sum[\![\text{Number}]\!]\langle y \mid y \leftarrow ys\rangle$.

It is worth noting that generators in Fortress is equipped with some fusion optimizations such that two consecutive maps are fused into one map, and two filter s are also fused into one filter . Delaying their computations by making computation objects instead of immediate execution of the computations, Fortress implements those fusions at the library level.

Similarly, in arbitrary points of Fortress programs, we can write various comprehensions with GoGs to perform nested reductions. Table 6.5 lists comprehensions written with GoGs for various applications, where $\downarrow$ is the minimum operator, and $\prod ys$ and $\sum ys$ are abbreviations of $\prod\langle y \mid y \leftarrow ys\rangle$ and $\sum\langle y \mid y \leftarrow ys\rangle$, respectively. Nested reductions with GoGs can describe various applications including typical problems in the functional community [Bir87, Gor97, SHTO00, Bir01], such as the

```
trait Generator2AT⟦E⟧ extends { Generator2⟦E⟧, GeneratorAT⟦GeneratorAT⟦E⟧⟧ }
  generate2AT⟦R⟧(op : Reduction⟦R⟧, ot : Reduction⟦R⟧, om : Reduction⟦R⟧, od : Reduction⟦R⟧, body: E → R) : R
  theoremsAT⟦R⟧() : List⟦((Reduction⟦R⟧, Reduction⟦R⟧, Reduction⟦R⟧, Reduction⟦R⟧, E → R) → Boolean,
                         (Reduction⟦R⟧, Reduction⟦R⟧, Reduction⟦R⟧, Reduction⟦R⟧, E → R) → R)⟧ = ⟨⟩
  naiveImplAT⟦R⟧(op : Reduction⟦R⟧, ot : Reduction⟦R⟧, om : Reduction⟦R⟧, od : Reduction⟦R⟧, f : E → R) : R =
                         generateAT⟦R⟧(op, ot, (fn (x) ⇒ x.generateAT⟦R⟧(om, od, f)))
end
```

Figure 6.31. The base trait of GoGs for two-dimensional data structures. The interface method receives four reduction operators: the first two for the outer reduction, and the last two for the inner reductions.

```
component ExampleProgram
import List.{...}
import Generator2.{...}
export Executable
run() : () = do
  xs = array⟦Number⟧(400).fill(fn (x : ℤ32) : Number ⇒ ⌊random(10) − 5⌋)
  flat_4 = relationalPredicate⟦Number⟧(fn (a, b) ⇒ |a − b| < 4)
  ascending = relationalPredicate⟦Number⟧(fn (a, b) ⇒ a < b)
  mpss = BIG MAX ⟦Number⟧⟨⟦Number⟧ ∑⟦Number⟧ys | ys ← segs xs, flat_4 ys, ascending ys⟩
  println("the maximum flat4-ascending-segment sum of xs is " mpss)
end
end
```

Figure 6.32. Ax example Fortress program to compute the maximum p-segment sum with the GoG library.

maximum prefix sum problem in the introduction, and the maximum segment sum problem that is a simplified problem to find a region of interest in a given sequence. Also, use of predicates can control targets of the inner reductions, which further broadens the application area of GoGs. The predicates *descending* and *equiv* used in examples are relational predicates of relations $x_{i-1} > x_i$ and $x_{i-1} = x_i$, respectively.

Most of the example comprehension shown above will be optimized at the runtime by executing the computation with efficient algorithms, although their naive computations are inefficient.

One feature of writing programs with GoGs is its easiness of writing nested reduction with various dependencies in clear and uniform expression. Writing nested reductions without GoGs usually requires explicit use of several or many indices generated by generators, which decreases productivity and maintainability of programs. For example, a naive reduction on prefixes or suffixes uses two indices, and that of segments needs three. Moreover, writing nested reductions on all subsequences is very difficult, because it requires infinite indices to write it in the same way.

Label	Fortress code with GoGs	Description
MPS	BIG MAX $\langle \sum ys \mid ys \leftarrow \mathit{prefixes}\ xs \rangle$	The maximum of sums of prefixes
MSP	BIG MIN $\langle \prod ys \mid ys \leftarrow \mathit{suffixes}\ xs \rangle$	The minimum of products of suffixes
MSS	BIG MAX $\langle \sum ys \mid ys \leftarrow \mathit{segments}\ xs \rangle$	The maximum of sums of all continuous subsequences
PSS	$\sum \langle \prod ys \mid ys \leftarrow \mathit{subseqs}\ xs \rangle$	The sum of products of all subsequences
pMPS	BIG MAX $\langle \sum ys \mid ys \leftarrow \mathit{prefixes}\ xs, \mathit{flat}_4\ ys \rangle$	The maximum sum of 4-flat prefixes in which every difference of neighboring elements is less than 4.
pMSP	BIG MIN $\langle \prod ys \mid ys \leftarrow \mathit{suffixes}\ xs, \mathit{descending}\ ys \rangle$	The minimum product of descendingly ordered suffixes.
pMSS	BIG MAX $\langle \sum ys \mid ys \leftarrow \mathit{segments}\ xs, \mathit{flat}_4\ ys, \mathit{ascending}\ ys \rangle$	The maximum sum of ascendingly ordered, 4-flat segments.
pPSS	$\sum \langle \prod ys \mid ys \leftarrow \mathit{subseqs}\ xs, \mathit{equiv}\ ys \rangle$	The sum of products of subsequences made of equivalents.
MRS	BIG MAX $\langle \sum ys \mid ys \leftarrow \mathit{subrecs}\ xs \rangle$	The maximum rectangle sum.

Table 6.5. Example comprehensions with GoGs.

Another advantage of writing programs with GoGs is that it reduces the risk of losing maintainability caused by hand-optimization. Performing optimization with high-level mathematical knowledge (i.e., calculation theorems) is not so hard for humans, but doing it by hand poses the risk of losing maintainability. A programmer cannot return an optimized program (i.e., an efficient implementation provided by a theorem) to the non-optimized program, to change the reduction operators to those not satisfying the application condition, unless he knows the original program.

Therefore, GoGs are very useful for easy and clear development of programs for nested reductions.

6.4.7 Experiment Results

We measured the execution time of micro-benchmarks listed in Table 6.5 with and without the optimization. We used the current Fortress interpreter (release 3294 from the subversion repository) [For08] run on a PC with two quadcore CPUs (Intel®Xeon®E5430, total 8 cores), 8GB memory, and Linux 2.6.24.

Table 6.6 lists measured execution time for several data sets. The sizes of input sequence (array) xs of the data sets (S, M, L, X) are (600, 1200, 12000, 120000) for MPS, MSP, pMPS and pMSP, (60, 120, 1200, 12000) for MSS and pMSS, (8, 16, —, —) for PSS and pPSS, (8×8, 16×16, 64×64, 128×128) for MRS. The table also lists asymptotic cost of the computation for the input of size n (the length of the input lists, and the width and the height of input arrays).

The figures of S and M without optimization show that the naive implementations are actually inefficient since their asymptotic cost is more than linear. Also, comparison of the results of M with and without optimization shows that the absolute execution time without optimization is much greater than that of efficient implementations dispatched by the library. The figures for L and X with optimization show that asymptotic cost of dispatched implementation is actually linear for MPS, MSP, MSS, pMPS, pMSP, and pMSS, and cubic for MRS. It is worth noting that codes dispatched by the library can achieve the same execution times as hand-coded solutions to the problems, since the library dispatches the hand-coded implementations given by calculation theorems, as long as they use generators for

	Without optimization			With optimization			
Label	Asym. cost	S	M	Asym.	M	L	X
MPS	O(n^2)	6.55	16.4	O(n)	0.253	0.576	2.02
MSP	O(n^2)	2.41	7.66	O(n)	0.21	0.772	1.85
MSS	O(n^3)	2.50	13.7	O(n)	0.053	0.110	0.34
PSS	O(2^n)	0.829	18.8	—	—	—	—
pMPS	O(n^2)	7.17	26.2	O(n)	0.257	0.666	4.29
pMSP	O(n^2)	4.88	18.0	O(n)	0.152	0.627	5.26
pMSS	O(n^3)	3.04	18.3	O(n)	0.0389	0.132	0.903
pPSS	O(2^n)	0.192	21.5	—	—	—	—
MRS	O(n^6)	5.52	615.12	O(n^3)	3.00	65.94	370.2

Table 6.6. Measured execution time (in seconds) of nested reductions in Table 6.5 with/without optimization. The sizes of input sequence (array) *xs* of the data sets (S, M, L, X) are (600, 1200, 12000, 120000) for MPS, MSP, pMPS and pMSP, (60, 120, 1200, 12000) for MSS and pMSS, (8, 16, —, —) for PSS and pPSS, (8×8, 16×16, 64×64, 128×128) for MRS. The table also lists asymptotic cost of the computation for the input of size n (the length of the input lists, and the width and the height of input arrays).

# of thm.	1	4	16	64	256	1024	4096
Time	3.55	3.18	4.3	9.6	28.7	103	395

Table 6.7. Overheads of the dispatching process for various numbers of theorems (in milli-seconds).

flat reductions. The results of PSS and pPSS with optimization are omitted since the library currently has no theorem for them. These results shows the library's optimization is dramatically effective for efficient computation of various nested reductions.

We also measured overhead of the dispatching process in the method *generate2*. We used dummy GoGs such that each of them has the list *theorems* of l dummy theorems. The condition of the dummy theorem checks distributivity and commutativity of operators (with functions in Figure 6.26) but returns always false. Thus, the dispatching process checks the dummy condition l times, and fails in finding an applicable theorem. It finally executes its dummy naive implementation that performs nothing. Therefore, we measured the execution time of the method *generate2* of dummy GoGs as the time of dispatching process.

Table 6.7 lists measured time of dispatching process for various sizes of theorem lists. The time of dispatching process is very small and ignorable against that of nested reductions, unless too many (more than hundreds) theorems are given. If so many theorems are given, we need to organize those theorems for efficient dispatching. It is a part of future work.

We mention parallelism of dispatched efficient implementations. Each of them

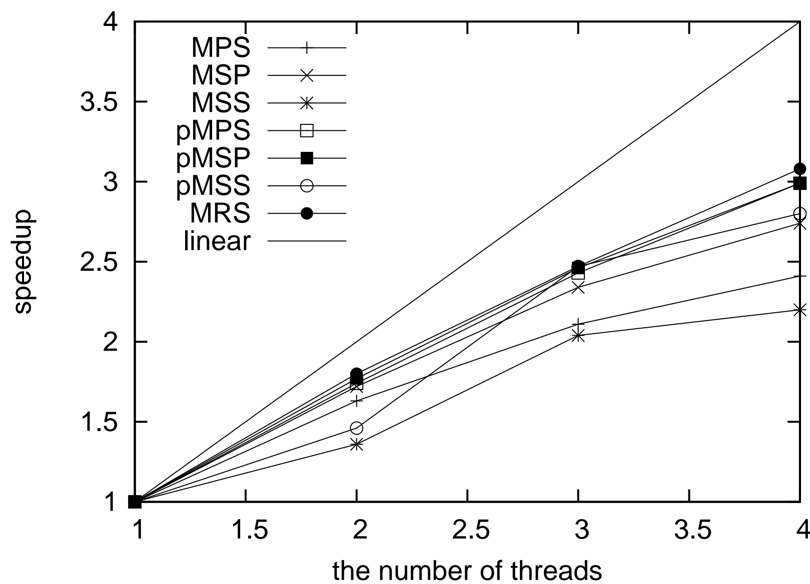


Figure 6.33. Speedup of programs listed in Table 6.5 for the data set X.

exploits parallelism provided by the existing generators in Fortress, since it just supplies newly constructed operators and functions to the seed generator held in the GoG. Therefore, parallelism of dispatched implementations is guaranteed by that of generators in Fortress. Figure 6.33 shows speedup of the optimized programs of Table 6.5. The figure shows good speedup of the programs. Unfortunately, the current Fortress interpreter has limitations on parallelism (less than 4 times speedup at the maximum for any programs), and thus the figure only shows the results of at most 4 threads. This limitation will be removed in the future Fortress interpreter or compiler, and thus the programs will be able to achieve better speedup, similar to the results on C++ implementation (see Section 6.2).

Therefore, programmers can receive the benefit that they can get the results by the optimized efficient parallel implementation by simply writing the naive programs. Without the library, they would suffer from writing complicated hundreds-line code for each program.

6.4.8 Discussion

Normalization of Comprehensions for Optimization

We show, with an example, that quite a lot of nested comprehensions can be systematically transformed into the form on which current our optimization mechanism (desugaring process) works. The summary of transformations steps follows the example transformation.

Consider the next nested comprehension as the example.

$$\texttt{BIG MAX}\,\langle\, \textstyle\sum\langle\, f(y,b,w) \mid y \leftarrow ys, \mathit{even}\ y \,\rangle \mid \\ ys \leftarrow \mathit{prefixes}\ xs, \mathit{ascending}\ ys, b \leftarrow bs \,\rangle$$

This example computes a variant of the maximum prefix sum, in which the maximum is considered only on ascending prefixes, the summation is taken only on even numbers, and the value is replaced with $f(y, b, w)$ instead of the number itself (y) within the summation.

The first step of the transformation is to fuse guards into their preceding generators, which is already shown in Section 6.4.5. For example, fusing guards in the example we get the following program.

$$\texttt{BIG MAX}\,\langle\, \textstyle\sum\langle\, f(y,b,w) \mid y \leftarrow ys.\mathit{filter}(\mathit{even}) \,\rangle \mid \\ ys \leftarrow (\mathit{prefixes}\ xs).\mathit{filter}(\mathit{ascending}), b \leftarrow bs \,\rangle$$

This transformation is applicable, when each predicate depends only on a variable in the left hand side of generators in the same comprehension.

The next step is to move depending generations to the edges. For example, the example has the pair of depending generations $y \leftarrow ys.\mathit{filter}(\mathit{even})$ and $ys \leftarrow (\mathit{prefixes}\ xs).\mathit{filter}(\mathit{ascending})$. Since the reduction operator `MAX` of the example is commutative, we can perform this transformation to get the following program.

$$\texttt{BIG MAX}\,\langle\, \textstyle\sum\langle\, f(y,b,w) \mid y \leftarrow ys.\mathit{filter}(\mathit{even}) \,\rangle \mid \\ b \leftarrow bs, ys \leftarrow (\mathit{prefixes}\ xs).\mathit{filter}(\mathit{ascending}) \,\rangle$$

Here, $ys \leftarrow (\mathit{prefixes}\ xs).\mathit{filter}(\mathit{ascending})$ is moved to the edge using the commutativity.

The third step is to restructure comprehensions to extract the form. The first stage of the restructuring is to strip generators from the outer comprehension, except for the edge generator. Applying the restructuring, we get the following program for the example, in which the generator $b \leftarrow bs$ is moved to a function h.

$$h(z) = \texttt{BIG MAX}\,\langle\, z(b) \mid b \leftarrow bs \,\rangle \\ h(\texttt{fn}\ \ b \Rightarrow \texttt{BIG MAX}\,\langle\, \textstyle\sum\langle\, f(y,b,w) \mid y \leftarrow ys.\mathit{filter}(\mathit{even}) \,\rangle \mid \\ ys \leftarrow (\mathit{prefixes}\ xs).\mathit{filter}(\mathit{ascending}) \,\rangle)$$

The second stage then strips generators and predicates from the inner comprehension. Making a new function f'', we get the following result for the example.

$$h(z) = \texttt{BIG MAX}\,\langle\, z(b) \mid b \leftarrow bs \,\rangle \\ h(\texttt{fn}\ \ b \Rightarrow \texttt{do}\ f''(y) = \texttt{if}\ \ \mathit{even}\ y\ \texttt{then}\ \ f(y,b,w)\ \texttt{else}\ 0\ \texttt{end} \\ \texttt{BIG MAX}\,\langle\, \textstyle\sum\langle\, f''y \mid y \leftarrow ys \,\rangle \mid \\ ys \leftarrow (\mathit{prefixes}\ xs).\mathit{filter}(\mathit{ascending}) \,\rangle \\ \texttt{end})$$

Here, the argument of h includes the nested reduction of the form, and it can be processed by our optimization mechanism.

Here is the summary of transformations.

Step. 1 Remove guards $p\ x$ by fusing it with generators $x \leftarrow g\ xs$.

$$\bigoplus[\, e \,|\, x \leftarrow g\ xs, p\ x, gs \,] \;\;\Rightarrow\;\; \bigoplus[\, e \,|\, x \leftarrow (g\ xs).\mathit{filter}(p), gs \,]$$

Step. 2 Move depending generations to the edges. If there is depending generations in generators of two comprehensions, move those depending generation to the edges of comprehensions as follows.

$$\bigoplus[\, \bigotimes[\, e \mid gs_1 \,] \mid gs_2 \,] \;\;\Rightarrow\;\; \bigoplus[\, \bigotimes[\, e \mid y \leftarrow \mathit{fg}\ ys, gs'_1 \,] \mid gs'_2, ys \leftarrow \mathit{fgg}\ xs \,]$$

Here, *fgg* is one of GoGs with filter, and *fg* is the identity function or filter. This transformation is valid if each operator of reductions is commutative and there is no dependency of gs'_2 to ys.

Step. 3 Restructure comprehensions to extract the form. The following is a rule used in this step.

$$\begin{array}{l}\bigoplus[\, \bigotimes[\, e \mid y \leftarrow \mathit{fg}\ ys, gs'_1 \,] \mid gs'_2, ys \leftarrow \mathit{fgg}\ xs \,] \\ \quad \Rightarrow \bigoplus[\, \bigoplus[\, \bigotimes[\, \bigotimes[\, e \,|\, gs'_1 \,] \mid y \leftarrow \mathit{fg}\ ys \,] \mid ys \leftarrow \mathit{fgg}\ xs \,] \,|\, gs'_2 \,]\end{array}$$

This transformation is always valid, since it is a combination of steps used in the usual desugaring process in Fortress. For readability, the result of this transformation can be written as the following form.

$$\begin{array}{l}h(\bigoplus[\, \bigotimes[\, f'(y) \mid y \leftarrow \mathit{fg}\ ys \,] \mid ys \leftarrow \mathit{fgg}\ xs \,]) \\ \quad \textbf{where} \quad h(z) = \bigoplus[\, z \,|\, gs'_2 \,] \\ \quad\quad\quad\quad\quad f'(y) = \bigotimes[\, e \,|\, gs'_1 \,]\end{array}$$

Now, the argument of h has the desired form.

It is an interesting part of future work to implement more power full desugaring process that can deal with the transformation.

It is worth noting that the transformation shown above has some restrictions on target comprehensions. However, we think many practical examples satisfy the restrictions.

Optimization of Three or More Nested Reductions

As far as we are aware, practical applications, such as nested queries on sequences, do not actually need optimization on 3+ level deep reductions (comprehensions). They can optimized by repeatedly applying optimizations for simply nested comprehensions (i.e., 2 level deep comprehensions), since optimization of simply nested reductions often results in another flat reduction. We can implement such successive optimizations on deeply nested reductions, if we let the dispatching process return a computation object to perform the flat reduction instead of immediately executing the optimized reduction. This also requires some modification on the current desugaring process.

Implementation in Languages Other Than Fortress

A library with the same functionality can be implemented similarly on other programming languages with subtyping and overloading, such as Java and C++. We chose Fortress because of its features: parallelism by default, generators, and expression branching based on types. Those features enables us to make a smart and clear library with interesting functionalities. For example, expressions branching based on types, such as `typecase`, enable clear and fine control over dispatching implementations based on types. The control over priorities of overloaded functions in C++ is far more complicated and it would result in no maintainable libraries. We also believe that the same functionality of our proposed library can be implemented in functional languages such as Haskell [Jon02, The08] and Objective Caml [LDG+08].

Dynamic Dispatching and Static Dispatching

Although our current implementation uses dynamic dispatching of implementations, we believe that the dispatching can also be done statically based on static types at compile time. For example, we can use C++ template specialization for the static dispatching. The static dispatching has the advantage that it has no dispatching overhead at runtime.

The experiment results, however, show that the overhead of the dynamic dispatching is reasonably small with respect to computation time of the reductions, and the overhead is ignorable. Also, the dynamic dispatching has another advantage that it can use runtime type information. Since it is generally finer than the static type information, the dynamic dispatching can apply better calculation rules in the optimization, and this advantage would be far better than the advantage of static dispatching.

Non-mathematical Properties

It is worth noting that our optimization mechanism can handle so-called "non-mathematical" properties, as long as we can make calculation theorems involving the properties. For example, the floating point addition does not satisfy the mathematical property "associativity," but it is approximately associative. We would be able to develop a theory involving the "approximated associativity," in which calculation theorems give efficient implementations to compute the equivalent results under the approximation. Actually, Fortress' generators change the computational structures under the approximated associativity of reduction operators. Also, parallelization with OpenMP [CDK+01, CJvdP07] does the same approximation to restructure computation of loops, and this approximation is reasonable and widely accepted.

6.5 Related Work

We review our work in context of comprehensions, nested reduction, and rewriting.

Comprehensions and Nested Reductions

Programming using comprehension has been considered a promising approach for concise parallelization, with a history of decades-long research [BS90,BHS+94,Ble96, BG96, CK00, CK01, CKLP01, LCK06, CLJ+07, FRR+07]. The research contributed to extracting data parallelism in aggregate computations they naturally express; these previous approaches do not leave to user programmers detailed controls on parallel computation, for example, how data is distributed among processors or how the systems carry out computation in parallel. Generators in Fortress allows flexibility in this respect, enabling sophisticated tuning of programs according to variety of their running circumstances. The previous work also studied optimization through flattening of nested comprehensions to effectively exploit a flat parallelism [BS90, LCK06] or fusion of successive operations to eliminate intermediate data passed among them [CK01]. These efforts stayed, however, in the problem of balancing computation tasks of nested reductions. Our work successfully goes one big step further to improve the complexity of computation without the hassles of programmers.

Library-level Optimizations

Another area of theoretical computer science, namely term rewriting, relates to our research. Glasgow Haskell Compiler (GHC) is one existing example to employ this technique: its RULES pragma [The08] enables users to implement library-level optimization by rewriting based on a set of rewriting rules. For example, the short cut fusion [GLJ93] is implemented by a set of rewriting rules in GHC. This approach, however, often suffers from inability to extract properties or surrounding contexts for applying correct and suitable rules, and the GHC's RULES pragma is not the exception. Our proposed library similarly performs optimization by rewriting, but with automatically examining specific mathematical properties of the input program to guarantee that the rewritten output program is actually a parallel program. Our approach therefore can be seen as an alternative to optimization by rewriting.

Broadway compiler [GL05] and the telescoping languages system [KBC+05] also employ library-level optimization by rewriting. Their rewriting rules concern mathematical properties of operand values of operations, while ours mainly concern properties of operators and functions given to higher-order functions.

Skeleton Libraries and Systems

Many skeleton libraries and systems have been proposed so far, which includes P3L [BDO+95,DPP97,Pel98], SCL [DFH+93,DGTY95], eSkel [Col04,BC05,BCHG05],

MuesLi [Kuc02], and QUAFF [FSCL06] .

The P3L [Pel98,BDO+95,DPP97] system adopts the idea of separating the higher skeletal parallel part from the lower, sequential part in parallel programs. In a P3L program, skeletal part is written in a functional notation, while the base part is described in the C language. From these descriptions, the P3L compiler generates a C code that calls MPI library functions. The system supports data parallel skeletons and communication skeletons for distributed lists. It also supports control parallel skeletons. The selection of the set of data parallel skeletons is similar to our framework.

In Darlington's framework SCL [DFH+93, DGTY95], the user writes a parallel program in two-layer structure: higher skeleton level and lower base language level. The user writes the higher level of the program with skeletons, abstracting its parallel behavior using the SCL (Structured Coordination Language). Its syntax is some kind of functional notation. The lower sequential part of the program is described in the base language. The system supports data parallel skeletons and communication skeletons for distributed arrays (one or two dimensional), and control parallel skeletons.

eSkel [Col04, BC05, BCHG05] is a library of C functions, also implemented on top of MPI. The latest version of eSkel supports control parallel skeletons, putting emphasis on addressing the issues of nesting of skeletons and interaction between parallel activities.

Muesli, developed by Kuchen [Kuc02], is a C++ library that works using MPI. It supports data parallel skeletons for distributed list and matrix, and task parallel skeletons. The system has two tier model: a parallel computation consists of a sequence of independent task parallel computations where each computation may nest task parallel skeletons arbitrarily, and an atomic (i.e. non-nested) task parallel computation can contain data parallelism. However, nesting constructors for the corresponding skeletons to generate a process topology is sometimes hard for the user because of low abstraction of task parallel skeletons.

QUAFF [FSCL06] is a skeleton-based parallel programming library on C++. It supports three task parallel skeletons. Its main originality is to rely on C++ template meta-programming techniques to achieve high efficiency. In particular, by performing most of skeleton instantiation and optimization at compile-time, QUAFF can keep the overhead very small.

APL [Ive62, FI73, Ber93] is a pioneer language that supports operators to manipulate arrays, namely array operators. APL's array operators supports element-wise computations, reductions to collapse arrays with binary operators, and manipulation of the layout of arrays such as shift and rotate. More complex computations on arrays can be made by composing the array operators. The idea of APL is the same as skeletal parallel programming, in the sense that both provide users with a set of basic patterns of array computations, and let users make sophisticated computations by compositions of these basic patterns. Also, both array operators and skeletons conceal complicated parallelism from users.

Chapter 7

Conclusion

7.1 Summary of the Thesis

This thesis has studied homomorphism-based structured parallel programming (also known as skeletal parallel programming), in which the skeletons to organize parallel programs have been designed based on homomorphisms of algebras of data structures. Structured by homomorphisms, the designed skeletons have good composability and good optimizability.

In the first part of the thesis, we have studied the homomorphism-based design of parallel skeletons for lists, two-dimensional arrays, and trees. Structured by homomorphisms, the designed skeletons have good composability and good optimizability. Especially, we have successfully designed skeletons for two-dimensional arrays, which had remained as a challenging problem. We have used the abide-tree algebra to represent two-dimensional arrays, which has nice freedom for parallelism owing to the abide property of constructors. The abide-tree algebra can be seen as an extension of the monoid algebra. We have shown that the designed skeletons are well composable with each other to build parallel programs for various problems.

In the second part, we have studied optimization of skeleton programs to solve the inefficiency problem of the compositional-style programming of skeletons. Based on the nice fusion laws of homomorphisms, we have successfully developed domain-independent fusion optimization for the skeletons, in which consecutive skeletons are fused into one skeleton to eliminate the redundant intermediate data structures between them. We have shown a derivation of non-trivial efficient programs for the maximum rectangle sum problem from naively-composed skeleton programs, as well as a general strategy for the derivation. The result shows that we can systematically develop non-trivial efficient algorithms from naive compositions of our skeletons. Also, we have proposed studied domain-specific optimizations for skeleton programs, to solve problems of the domain-independent fusion optimizations. We have concentrated on optimization of skeleton programs for computation involving neighbor elements, which can be categorized into two types: that involving a finite number of neighbor elements, such as filtering of sequences and images, the finite

difference method, and some matrix-vector operations; and that involving an infinite number of neighbor elements, such as queries of interesting segments on lists and rectangles (sub-arrays) on two-dimensional arrays. We have developed domain-specific fusion rules for the former type, proposing a new strategy for developing domain-specific fusion optimization of skeleton programs. Then, for the latter type that have been formalized as nested reductions, we have developed shortcut theorems to provide efficient algorithms to nested reductions by fusion. We have succeeded in developing the domain-specific fusions and the optimization theorems, owing to the solid theories of carefully designed homomorphism-based skeletons. Also, we have succeeded in widening the application area of our skeleton programming.

In the last part of the thesis, we have reported implementations of designed skeletons and optimization mechanisms. We have shown implementations of skeletons for distributed parallel machines, in which the parallelism of homomorphism guarantees their parallelism. We have shown small systems for domain-independent and domain-specific fusions, which have been implemented as source-to-source translators. Experiment results show the success of the implementations and the base theories. We have also proposed the general design of libraries with optimization capabilities based on optimization theorems in our theories. The design has been demonstrated with the implementation of the GoG library for nested reductions in Fortress, in which the useful comprehension notation is linked to our skeleton theories. The implementation of the optimizing libraries is lightweight in the sense that it does not need deep analysis of program codes. The key point is the fact that application conditions of theorems are described with parameters' possession of mathematical properties, which is due to the uniform structure of skeletons based on homomorphisms, i.e., higher-order functions. The implemented libraries can be seen as active dictionaries of algorithms, in the sense that efficient implementations are indexed in the dictionaries by naive programs.

7.2 Future Work

Although we have shown a strategy to make efficient parallel algorithms from naive compositions of skeletons, there is still a demand for methodologies to construct the initial naive compositions. Especially, there is a demand for a strong methodology to derive operators with the abide property, to derive naive parallel programs for two-dimensional arrays. For skeletal parallel programs on one-dimensional data structures, some techniques have been proposed to derive skeleton programs (associative operators). The key of one technique is the use of the associativity of function compositions as the seed of associative operators. We might be able to apply the key idea to derivation of *abide* operators. We have found two pairs of such seed operations: the pair of function composition and function tupling, and the pair of matrix multiplication and tensor product. Unfortunately, neither of seed pairs results in useful operator pairs, since one of the derived pair operators builds

big structures, in other words, it simply delays computations of its direction. Thus, investigation of good seed pairs is one part of future work.

Extension of parallel skeletons to multi-dimensional data structures is also one direction of future work. We have not found many application problems that essentially need multi-dimensional data structures, though.

Automatic discovery of mathematical properties of program objects is very useful for automatic optimization with calculation theorems. Also, a technique to solve priority problems in applying calculation theorems is still an open problem [PP96], since a greedy selection of a sequence of theorems does not necessarily result in the best implementation. We have avoided such priority problem by implementing only big-step theorems. However, the problem would arise again when the number of big-steps becomes large. Study of these techniques is one direction of future work.

It is also interesting future work to study the integration of data-parallel skeletons and task-parallel skeletons. The computation provided by task-parallel skeletons distributes tasks among processors, while the computation by data-parallel skeletons distributes data structures among processors, and carries out almost the same process on each processor. This thesis have studied data-parallel skeletons structured by homomorphisms on data structures, since the center of their computation is the data structure. However, it is an open problem what should be used to structure task-parallel skeletons. It is also an open problem whether we can integrate the theory of structured task-parallel skeletons into the theory of structured data-parallel skeletons.

Although we have shown that some matrix operations can be described with our skeletons, some matrix operations are not so easily written with the designed skeletons. Also, conditions for optimizing these operations are often described with properties of input matrices, while the optimization of our skeletons have conditions on operators (functions). Therefore, we might have to take another approach to structuring and optimizing those operations. Some results are found in a technical report [EHK+09].

Bibliography

[ACH+08] Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele Jr., and Sam Tobin-Hochstadt. The Fortress language specification version 1.0. `http://research.sun.com/projects/plrg/fortress.pdf`, 2008.

[ADKP89] Karl R. Abrahamson, N. Dadoun, David G. Kirkpatrick, and Teresa M. Przytycka. A simple parallel tree contraction algorithm. *Journal of Algorithms*, 10(2):287–302, June 1989.

[BBC+06] Anders Berglund, Scott Boag, Don Chamberlin, Mary F. Fernández, Michael Kay, Jonathan Robie, and Jérôme Siméon, editors. *XML Path Language (XPath) 2.0*. W3C Candidate Recommendation, June 2006. available from `http://www.w3.org/TR/xpath20/`.

[BC05] Anne Benoit and Murray Cole. Two fundamental concepts in skeletal parallel programming. In *International Conference on Computational Science (2)*, volume 3515 of *Lecture Notes in Computer Science*, pages 764–771. Springer-Verlag, 2005.

[BCHG05] Anne Benoit, Murray Cole, J. Hillston, and S. Gilmore. Flexible skeletal programming with eskel. In *Proceedings of 11th International Euro-Par Conference (Euro-Par'05)*, volume 3648 of *Lecture Notes in Computer Science*, pages 761–770. Springer-Verlag, 2005.

[BdM96] Richard S. Bird and Oege de Moor. *Algebra of Programming*. International Series in Computer Science. Prentice Hall, September 1996.

[BDO+95] Bruno Bacci, Marco Danelutto, Salvatore Orlando, Susanna Pelagatti, and Marco Vanneschi. P3L: A Structured High Level Programming Language and its Structured Support. *Concurrency Practice and Experience*, 7(3):225–255, 1995.

[Ben84a] Jon Bentley. Programming Pearls: Algorithm Design Techniques. *Communications of the ACM*, 27(9):865–873, 1984.

[Ben84b] Jon Bentley. Programming Pearls: Perspective on Performance. *Communications of the ACM*, 27(11):1087–1092, 1984.

[Ber93] R. Bernecky. The role of APL and J in high-performance computation. *APL Ouote Ouad*, 24(1):17–32, 1993.

[BG96] Guy E. Blelloch and John Greiner. A provable time and space efficient implementation of nesl. In *ICFP '96: Proceedings of the first ACM SIGPLAN international conference on Functional programming*, pages 213–225, New York, NY, USA, 1996. ACM.

[BGH+06] Ganesh Bikshandi, Jia Guo, Daniel Hoeflinger, Gheorghe Almasi, Basilio B. Fraguela, Maria J. Garzaran, David Padua, and Christoph von Praun. Programming for Parallelism and Locality with Hierarchically Tiled Arrays. In *Proceedings of 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'06)*, pages 48–57, New York, NY, USA, 2006. ACM Press.

[BHS+94] Guy E. Blelloch, Jonathan C. Hardwick, Jay Sipelstein, Marco Zagha, and Siddhartha Chatterjee. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing*, 21(1):4–14, 1994.

[Bir87] Richard S. Bird. An introduction to the theory of lists. In *Proceedings of the NATO Advanced Study Institute on Logic of programming and calculi of discrete design*, pages 5–42, New York, NY, USA, 1987. Springer-Verlag New York, Inc.

[Bir88] Richard S. Bird. Lectures on constructive functional programming. Technical Report Technical Monograph PRG-69, Oxford University Computing Laboratory, 1988.

[Bir98] Richard S. Bird. *Introduction to Functional Programming using Haskell.* Prentice Hall Series in Computer Science. Prentice Hall, 2nd edition, April 1998.

[Bir01] Richard S. Bird. Functional pearls maximum marking problems. *Journal of Functional Programming*, 11(4):411–424, 2001.

[Ble90] Guy E. Blelloch. Prefix sums and their applications. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, November 1990.

[Ble96] Guy E. Blelloch. Programming parallel algorithms. *Communication ACM*, 39(3):85–97, 1996.

[BS90] Guy E. Blelloch and Gary W. Sabot. Compiling collection-oriented languages onto massively parallel computers. *Journal of Parallel and Distributed Computing*, 8(2):119–134, 1990.

[BSWB02] David A. Bader, Sukanya Sreshta, and Nina R. Weisse-Bernstein. Evaluating arithmetic expressions using tree contraction: A fast and scalable parallel implementation for symmetric multiprocessors (SMPs) (extended abstract). In Sartaj Sahni, Viktor K. Prasanna, and Uday Shukla, editors, *High Performance Computing — HiPC 2002, 9th International Conference, Bangalore, India, December 18–21, 2002, Proceedings*, volume 2552 of *Lecture Notes in Computer Science*, pages 63–78. Springer, 2002.

[CDK+01] Robit Chandra, Leonardo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, and Ramesh Menon. *Parallel programming in OpenMP*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[CJvdP07] Barbara Chapman, Gabriele Jost, and Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007.

[CK00] Manuel M. T. Chakravarty and Gabriele Keller. More types for nested data parallel programming. In *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 94–105, New York, NY, USA, 2000. ACM.

[CK01] Manuel M. T. Chakravarty and Gabriele Keller. Functional array fusion. In *ICFP '01: Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*, pages 205–216, New York, NY, USA, 2001. ACM.

[CKLP01] Manuel M. T. Chakravarty, Gabriele Keller, Roman Lechtchinsky, and W. Pfannenstiel. Nepal - nested data parallelism in haskell. In *Euro-Par '01: Proceedings of the 7th International Euro-Par Conference Manchester on Parallel Processing*, pages 524–534, London, UK, 2001. Springer-Verlag.

[CLJ+07] Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, Gabriele Keller, and Simon Marlow. Data parallel haskell: a status report. In *DAMP '07: Proceedings of the 2007 workshop on Declarative aspects of multicore programming*, pages 10–18, New York, NY, USA, 2007. ACM.

[Col89] Murray Cole. *Algorithmic Skeletons : A Structured Approach to the Management of Parallel Computation*. Research Monographs in Parallel and Distributed Computing, Pitman, London, 1989.

[Col95] Murray Cole. Parallel programming with list homomorphisms. *Parallel Processing Letters*, 5(2):191–203, June 1995.

[Col02] Murray Cole. eSkel Home Page. `http://homepages.inf.ed.ac.uk/mic/eSkel/`, 2002.

[Col04] Murray Cole. Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. *Parallel Comput.*, 30(3):389–406, 2004.

[CTH98] Wei-Ngan Chin, Akihiko Takano, and Zhenjiang Hu. Parallelization via context preservation. In *Proceedings of the 1998 International Conference on Computer Languages, ICCL '98, May 14–16, 1998, Chicago, IL, USA*, pages 153–162. IEEE Computer Society, 1998.

[CV88] Richard Cole and Uzi Vishkin. The accelerated centroid decomposition technique for optimal parallel tree evaluation in logarithmic time. *Algorithmica*, 3:329–346, March 1988.

[DFH+93] John Darlington, A. J. Field, Peter G. Harrison, Paul H. J. Kelly, D. W. N. Sharp, and Q. Wu. Parallel programming using skeleton functions. In *PARLE '93: Proceedings of the 5th International PARLE Conference on Parallel Architectures and Languages Europe*, pages 146–160, London, UK, 1993. Springer-Verlag.

[DGTY95] John Darlington, Yi-ke Guo, Hing Wing To, and Jin Yang. Parallel skeletons for structured composition. *SIGPLAN Not.*, 30(8):19–28, 1995.

[DH98] Krzysztof Diks and Torben Hagerup. More general parallel tree contraction: Register allocation and broadcasting in a tree. *Theoretical Computer Science*, 203(1):3–29, August 1998.

[DPP97] M. Danelutto, F. Pasqualetti, and S. Pelagatti. Skeletons for data parallelism in P3L. In *Proceedings of 3rd International Euro-Par Conference (Euro-Par'97)*, volume 1300 of *Lecture Notes in Computer Science*, pages 619–628. Springer-Verlag, 1997.

[EGJK04] Erik Elmroth, Fred Gustavson, Isak Jonsson, and Bo Kagstroom. Recursive Blocked Algorithms and Hybrid Data Structures for Dense Matrix Library Software. *SIAM Review*, 46(1):3–45, 2004.

[EHK+09] Kento Emoto, Zhenjiang Hu, Kazuhiko Kakehi, Kiminori Matsuzaki, and Masato Takeichi. A generative matrix library in fortress relieves programmers' headache! In *Proceedings of the 25th JSSST domestic conference*, 2009.

[FI73] Adin D. Falkoff and Kenneth E. Iverson. The design of APL. *IBM Journal of Research and Development*, 17(4):324–334, 1973.

[For08] Project Fortress. The reference interpreter for the Fortress language. `http://projectfortress.sun.com/Projects/Community`, 2008.

[FPS01] Paul F. Fischer, Franco P. Preparata, and John E. Savage. Generalized scans and tridiagonal systems. *Theoretical Computer Science*, 255(1–2):423–436, 2001.

[FRR+07] Matthew Fluet, Mike Rainey, John Reppy, Adam Shaw, and Yingqi Xiao. Manticore: a heterogeneous parallel language. In *DAMP '07: Proceedings of the 2007 workshop on Declarative aspects of multicore programming*, pages 37–44, New York, NY, USA, 2007. ACM.

[FSCL06] J. Falcou, J. Sérot, T. Chateau, and J. T. Lapresté. Quaff: efficient c++ design for parallel skeletons. *Parallel Comput.*, 32(7):604–615, 2006.

[FW03] Jeremy D. Frens and David S. Wise. QR Factorization with Morton-Ordered Quadtree Matrices for Memory Re-use and Parallelism. In *Proceedings of 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'03)*, pages 144–154, 2003.

[GCS94] Jeremy Gibbons, Wentong Cai, and David B. Skillicorn. Efficient Parallel Algorithms for Tree Accumulations. *Science of Computer Programming*, 23(1):1–18, 1994.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional, 2005.

[GL05] Samuel Zev Guyer and Calvin Lin. Broadway: A compiler for exploiting the domain-specific semantics of software libraries. *Proceedings of the IEEE*, 93(2):342–357, 2005.

[GLJ93] Andrew Gill, John Launchbury, and Simon L. Peyton Jones. A short cut to deforestation. In *FPCA '93: Proceedings of the conference on Functional programming languages and computer architecture*, pages 223–232, New York, NY, USA, 1993. ACM.

[GLS99] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI (2nd ed.): portable parallel programming with the message-passing interface.* MIT Press, Cambridge, MA, USA, 1999.

[Gor96] Sergei Gorlatch. Systematic efficient parallelization of scan and other list homomorphisms. In Luc Bougé, Pierre Fraigniaud, Anne Mignotte, and Yves Robert, editors, *Euro-Par '96 Parallel Processing, Second International Euro-Par Conference, Lyon, France, August 26–29, 1996, Proceedings, Volume II*, volume 1124 of *Lecture Notes in Computer Science*, pages 401–408. Springer, 1996.

[Gor97] Sergei Gorlatch. Optimizing compositions of scans and reductions in parallel program derivation. Technical Report MPI–9711, Universität Passau, 1997.

[GS06] Clemens Grelck and Sven-Bodo Scholz. Merging compositions of array skeletons in SaC. *Parallel Computing*, 32(7–8):507–522, 2006.

[GW06] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing (3rd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006.

[GWL99] Sergei Gorlatch, Christoph Wedler, and Christian Lengauer. Optimization rules for programming with collective operations. In *13th International Parallel Processing Symposium / 10th Symposium on Parallel and Distributed Processing (IPPS / SPDP '99), 12-16 April 1999, San Juan, Puerto Rico, Proceedings*, pages 492–499. IEEE Computer Society, 1999.

[HIT97] Zhenjiang Hu, Hideya Iwasaki, and Masato Takeichi. Formal Derivation of Efficient Parallel Programs by Construction of List Homomorphisms. *ACM Transactions on Programming Langauges and Systems*, 19(3):444–461, 1997.

[HIT02] Zhenjiang Hu, Hideya Iwasaki, and Masato Takeichi. An Accumulative Parallel Skeleton for All. In *Proceedings of 11st European Symposium on Programming (ESOP 2002), LNCS 2305*, pages 83–97. Springer-Verlag, April 2002.

[HTC98] Zhenjiang Hu, Masato Takeichi, and Wei-Ngan Chin. Parallelization in calculational forms. In *POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, January 19–21, 1998, San Diego, CA, USA*, pages 316–328. ACM Press, 1998.

[HTI99] Zhenjiang Hu, Masato Takeichi, and Hideya Iwasaki. Diffusion: Calculating efficient parallel programs. In Olivier Danvy, editor, *Proceedings of the 1999 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, San Antonio, Texas, January 22–23, 1999*, pages 85–94. University of Aarhus, 1999. Technical report BRICS-NS-99-1.

[Ive62] Kenneth E. Iverson. *A Programming Language*. John Wiley and Sons, 1962.

[Jai89] Anil K. Jain. *Fundamentals of Digital Image Processing*. Prentice Hall, 1989.

[Jeu93] Johan Theodoor Jeuring. *Theories for Algorithm Calculation*. PhD thesis, Faculty of Science, Utrecht University, 1993.

[Jon02] Simon Peyton Jones. Haskell 98 language and libraries: The revised report. PDF, September 2002.

[KBC+05] Ken Kennedy, Bradley Broom, Arun Chauhan, Rob Fowler, John Garvin, Charles Koelbel, Cheryl McCosh, and John Mellor-Crummey. Telescoping languages: A system for automatic generation of domain languages. *Proceedings of the IEEE*, 93(2):387–408, Feb. 2005.

[Kuc02] Herbert Kuchen. A Skeleton Library. In *Proceedings of 8th International Euro-Par Conference (Euro-Par'02)*, volume 2400 of *Lecture Notes in Computer Science*, pages 620–629. Springer-Verlag, 2002.

[LCK06] Roman Leshchinskiy, Manuel M. T. Chakravarty, and Gabriele Keller. Higher order flattening. In Vassil N. Alexandrov, G. Dick van Albada, Peter M. A. Sloot, and Jack Dongarra, editors, *Computational Science - ICCS 2006, 6th International Conference, Reading, UK, May 28-31, 2006, Proceedings, Part II*, volume 3992 of *Lecture Notes in Computer Science*, pages 920–928. Springer, 2006.

[LDG+08] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Remy, and Jerome Vouillon. The Objective Caml system release 3.11. `http://caml.inria.fr/`, November 2008.

[LM88] Charles E. Leiserson and Bruce M. Maggs. Communication-efficient parallel algorithms for distributed random-access machines. *Algorithmica*, 3:53–77, 1988.

[Mat07] Kiminori Matsuzaki. *Parallel Programming with Tree Skeletons*. PhD thesis, Graduate School of Information Science and Technology, University of Tokyo, 2007.

[Mee88] Lambert Meertens. First steps towards the theory of rose trees. CWI, Amsterdam; IFIP Working Group 2.1 working paper 592 ROM-25, 1988.

[MFP91] Erik Meijer, Maarten M. Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*, pages 124–144, London, UK, 1991. Springer-Verlag.

[MIEH06] Kiminori Matsuzaki, Hideya Iwasaki, Kento Emoto, and Zhenjiang Hu. A library of constructive skeletons for sequential style of parallel programming. In *InfoScale '06: Proceedings of the 1st international conference on Scalable information systems*, volume 152 of *ACM International Conference Proceeding Series*, page 13. ACM Press, 2006.

[Mil94] Richard Miller. *Two Approaches to Architecture-Independent Parallel Computation*. PhD thesis, Computing Laboratory, Oxford University, 1994.

[MKI+04] Kiminori Matsuzaki, Kazuhiko Kakehi, Hideya Iwasaki, Zhenjiang Hu, and Yoshiki Akashi. A Fusion-Embedded Skeleton Library. In *Proceedings of 10th International Euro-Par Conference (Euro-Par'04)*, volume 3149 of *Lecture Notes in Computer Science*, pages 644–653. Springer-Verlag, 2004.

[MR85] Gary L. Miller and John H. Reif. Parallel tree contraction and its application. In *26th Annual Symposium on Foundations of Computer Science, 21–23 October 1985, Portland, Oregon, USA*, pages 478–489. IEEE Computer Society, 1985.

[MW97] Ernst W. Mayr and Ralph Werchner. Optimal tree contraction and term matching on the hypercube and related networks. *Algorithmica*, 18(3):445–460, July 1997.

[OHIT97] Yoshiyuki Onoue, Zhenjiang Hu, Hideya Iwasaki, and Masato Takeichi. A calculational fusion system hylo. In *Proceedings of the IFIP TC 2 WG 2.1 international workshop on Algorithmic languages and calculi*, pages 76–106, London, UK, UK, 1997. Chapman & Hall, Ltd.

[Pel98] Susanna Pelagatti. *Structured development of parallel programs.* Taylor & Francis, Inc., Bristol, PA, USA, 1998.

[PP96] Alberto Pettorossi and Maurizio Proietti. Rules and strategies for transforming functional and logic programs. *ACM Computing Surveys*, 28(2):360–414, 1996.

[RG02] Fethi A. Rabhi and Sergei Gorlatch, editors. *Patterns and Skeletons for Parallel and Distributed Computing.* Springer-Verlag, 2002.

[RT94] John H. Reif and Stephen R. Tate. Dynamic parallel tree contraction (extended abstract). In Lawrence Snyder and Charles E. Leiserson, editors, *SPAA '94: Proceedings of the 6th Annual ACM Symposium on Parallel Algorithms and Architectures, June 27–29, 1994, Cape May, New Jersey, USA*, pages 114–121. ACM Press, 1994.

[Rus06] John C. Russ. *The Image Processing Handbook, Fifth Edition (Image Processing Handbook).* CRC Press, Inc., Boca Raton, FL, USA, 2006.

[SG02] Jocelyn Serot and Dominique Ginhac. Skeletons for Parallel Image Processing: an Overview of the SKIPPER Project. *Parallel Computing*, 28(12):1685–1708, 2002.

[SHT01] Isao Sasano, Zhenjiang Hu, and Masato Takeichi. Generation of efficient programs for solving maximum multi-marking problems. In *SAIG 2001: Proceedings of the Second International Workshop on Semantics, Applications, and Implementation of Program Generation*, pages 72–91, London, UK, 2001. Springer-Verlag.

[SHTO00] Isao Sasano, Zhenjiang Hu, Masato Takeichi, and Mizuhito Ogawa. Make it practical: a generic linear-time algorithm for solving maximum-weightsum problems. In *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 137–149, New York, NY, USA, 2000. ACM.

[Ski94] David B. Skillicorn. *Foundations of Parallel Programming*, volume 6 of *Cambridge International Series on Parallel Computation*. Cambridge University Press, 1994.

[Ski96] David B. Skillicorn. Parallel implementation of tree skeletons. *Journal of Parallel and Distbributed Computing*, 39(2):115–125, December 1996.

[Ski97] David B. Skillicorn. Structured parallel computation in structured documents. *Journal of Univarsal Computer Science*, 3(1):42–68, January 1997.

[SO98] Marc Snir and Steve Otto. *MPI-The Complete Reference: The MPI Core*. MIT Press, Cambridge, MA, USA, 1998.

[Sto73] Harold S. Stone. An efficient parallel algorithm for the solution of a tridiagonal linear system of equations. *Journal of the ACM*, 20(1):27–38, 1973.

[Sve02] Josef Svenningsson. Shortcut fusion for accumulating parameters & zip-like functions. In *ICFP '02: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 124–132, New York, NY, USA, 2002. ACM.

[Tak02] Tadao Takaoka. Efficient Algorithms for the Maximum Subarray Problem by Distance Matrix Multiplication. In *Proceedings of Computing: The Australasian Theory Symposium (CATS'02)*, pages 189–198, 2002.

[The08] The GHC Team. The glorious glasgow haskell compilation system user's guide, version 6.10.1. `http://www.haskell.org/ghc/`, 2008.

[TM95] Akihiko Takano and Erik Meijer. Shortcut deforestation in calculational form. In *FPCA '95: Proceedings of the seventh international conference on Functional programming languages and computer architecture*, pages 306–313, New York, NY, USA, 1995. ACM.

[Wad88] Philip Wadler. Deforestation: Transforming programs to eliminate trees. In Harald Ganzinger, editor, *ESOP '88, 2nd European Symposium on Programming, Nancy, France, March 21-24, 1988, Proceedings*, volume 300 of *Lecture Notes in Computer Science*, pages 344–358. Springer, 1988.

[Wis84] David S. Wise. Representing Matrices as Quadtrees for Parallel Processors. *Information Processing Letters*, 20(4):195–199, 1984.

[Wis99] David S. Wise. Undulant Block Elimination and Integer-Preserving Matrix Inversion. *Science of Computer Programming*, 22(1):29–85, 1999.

[WL98] Christoph Wedler and Christian Lengauer. On linear list recursion in parallel. *Acta Informatica*, 35(10):875–909, October 1998.

[XKH04] Dana N. Xu, Siau-Cheng Khoo, and Zhenjiang Hu. PType system: A featherweight parallelizability detector. In Wei-Ngan Chin, editor, *Programming Languages and Systems: Second Asian Symposium, APLAS 2004, Taipei, Taiwan, November 4–6, 2004. Proceedings*, volume 3302 of *Lecture Notes in Computer Science*, pages 197–212. Springer, 2004.

[Zan92] Hans Zantema. Longest segment problems. *Science of Computer Programming*, 18(1):39–66, 1992.

[Zha02] Haiyan Zhao. *A Compositional Approach to Mining Optimal Ranges.* PhD thesis, Department of Information Engineering, University of Tokyo, 2002.

Appendix A

Auxiliary Rules for Skeletons on Two-dimensional Arrays

Rule I

$$
\begin{array}{l}
\mathsf{map}\ f\ (\mathsf{zipwith}(\oplus)\ x\ y) = \mathsf{zipwith}(\oplus')\ (\mathsf{map}\ f\ x)\ (\mathsf{map}\ f\ y) \\
\Leftarrow \forall a, b\ \ f\ (a \oplus b) = f\ a \oplus' f\ b
\end{array}
$$

Proof: It is proved by induction on the structure of abide trees.

$$
\begin{array}{ll}
 & \mathsf{map}\ f\ (\mathsf{zipwith}(\oplus)\ |a|\ |b|) \\
= & \qquad \{\ \text{def. of}\ \mathsf{zipwith}, \mathsf{map}\ \} \\
 & |f\ (a \oplus b)| \\
= & \qquad \{\ \text{hypo.}\ \} \\
 & |f\ a \oplus' f\ b| \\
= & \qquad \{\ \text{def. of}\ \mathsf{zipwith}, \mathsf{map}\ \} \\
 & \mathsf{zipwith}(\oplus')\ (\mathsf{map}\ f\ |a|)\ (\mathsf{map}\ f\ |b|)
\end{array}
$$

$$
\begin{array}{ll}
 & \mathsf{map}\ f\ (\mathsf{zipwith}(\oplus)\ (x \ominus y)\ (u \ominus v)) \\
= & \qquad \{\ \text{def. of}\ \mathsf{zipwith}, \mathsf{map}\ \} \\
 & \mathsf{map}\ f\ (\mathsf{zipwith}(\oplus)\ x\ u) \ominus \mathsf{map}\ f\ (\mathsf{zipwith}(\oplus)\ y\ v) \\
= & \qquad \{\ \text{hypo. of induction}\ \} \\
 & \mathsf{zipwith}(\oplus')\ (\mathsf{map}\ f\ x)\ (\mathsf{map}\ f\ u) \ominus \mathsf{zipwith}(\oplus')\ (\mathsf{map}\ f\ y)\ (\mathsf{map}\ f\ v) \\
= & \qquad \{\ \text{def. of}\ \mathsf{zipwith}, \mathsf{map}\ \} \\
 & \mathsf{zipwith}(\oplus')\ (\mathsf{map}\ f\ (x \ominus y))\ (\mathsf{map}\ f\ (u \ominus v))
\end{array}
$$

$$
\begin{array}{ll}
 & \mathsf{map}\ f\ (\mathsf{zipwith}(\oplus)\ (x \phi y)\ (u \phi v)) \\
= & \qquad \{\ \text{similar to}\ \ominus\ \} \\
 & \mathsf{zipwith}(\oplus')\ (\mathsf{map}\ f\ (x \phi y))\ (\mathsf{map}\ f\ (u \phi v))
\end{array}
$$

Rule II

$$
\begin{array}{l}
\mathsf{map}\ (\mathsf{reduce}(\oplus, \otimes))\ (\mathsf{zipwith}(\ominus)\ x\ y) \\
\qquad = \mathsf{zipwith}(\oplus)\ (\mathsf{map}\ (\mathsf{reduce}(\oplus, \otimes))\ x)\ (\mathsf{map}\ (\mathsf{reduce}(\oplus, \otimes))\ y)
\end{array}
$$

Proof: Rule I and the following calculation with $f = \mathsf{reduce}(\oplus, \otimes), \oplus = \ominus, \oplus' = \oplus$.

$$\mathsf{reduce}(\oplus, \otimes)\ (a \ominus b) = \mathsf{reduce}(\otimes, \oplus)\ a \oplus \mathsf{reduce}(\otimes, \oplus)\ b$$

Rule III

$$\begin{array}{l} \mathsf{map}\ f\ (gemm(\oplus, \otimes)\ x\ y) = gemm(\oplus', \otimes')\ (\mathsf{map}\ f\ x)\ (\mathsf{map}\ f\ y) \\ \Leftarrow \forall a, b\ \ f\ (a \oplus b) = f\ a \oplus' f\ b,\ f\ (a \otimes b) = f\ a \otimes' f\ b \end{array}$$

Proof: It is proved by induction on the structure of abide trees.

$$\begin{array}{cl} & \mathsf{map}\ f\ (gemm(\oplus, \otimes)\ |a|\ |b|) \\ = & \quad \{\ \text{def. of } gemm, \mathsf{map}\ \} \\ & |f\ (a \otimes b)| \\ = & \quad \{\ \text{hypo.}\ \} \\ & |f\ a \otimes' f\ b| \\ = & \quad \{\ \text{def. of } gemm, \mathsf{map}\ \} \\ & gemm(\oplus', \otimes')\ (\mathsf{map}\ f\ |a|)\ (\mathsf{map}\ f\ |b|) \end{array}$$

$$\begin{array}{cl} & \mathsf{map}\ f\ (gemm(\oplus, \otimes)\ (x \ominus y)\ z) \\ = & \quad \{\ \text{def. of } gemm, \mathsf{map}\ \} \\ & \mathsf{map}\ f\ (gemm(\oplus, \otimes)\ x\ z) \ominus \mathsf{map}\ f\ (gemm(\oplus, \otimes)\ y\ z) \\ = & \quad \{\ \text{hypo. of induction}\ \} \\ & gemm(\oplus', \otimes')\ (\mathsf{map}\ f\ x)\ (\mathsf{map}\ f\ z) \ominus gemm(\oplus', \otimes')\ (\mathsf{map}\ f\ y))\ (\mathsf{map}\ f\ z) \\ = & \quad \{\ \text{def. of } gemm, \mathsf{map}\ \} \\ & gemm(\oplus', \otimes')\ (\mathsf{map}\ f\ (x \ominus y))\ (\mathsf{map}\ f\ z) \end{array}$$

$$\begin{array}{cl} & \mathsf{map}\ f\ (gemm(\oplus, \otimes)\ x\ (y \obar z)) \\ = & \quad \{\ \text{similar to above}\ \} \\ & gemm(\oplus', \otimes')\ (\mathsf{map}\ f\ x)\ (\mathsf{map}\ f\ (y \obar z)) \end{array}$$

$$\begin{array}{cl} & \mathsf{map}\ f\ (gemm(\oplus, \otimes)\ (x \obar y)\ (u \ominus v)) \\ = & \quad \{\ \text{def. of } gemm, \mathsf{map}\ \} \\ & \mathsf{map}\ f\ (\mathsf{zipwith}(\oplus)\ (gemm(\oplus, \otimes)\ x\ u)\ (gemm(\oplus, \otimes)\ y\ v)) \\ = & \quad \{\ \text{I}\ \} \\ & \mathsf{zipwith}(\oplus')(\mathsf{map}\ f\ (gemm(\oplus, \otimes)\ x\ u))\ (\mathsf{map}\ f\ (gemm(\oplus, \otimes)\ y\ v)) \\ = & \quad \{\ \text{hypo. of induction}\ \} \\ & \mathsf{zipwith}(\oplus')(gemm(\oplus', \otimes')\ (\mathsf{map}\ f\ x)\ (\mathsf{map}\ f\ u)) \\ & \qquad\qquad\qquad (gemm(\oplus', \otimes')\ (\mathsf{map}\ f\ y)\ (\mathsf{map}\ f\ v)) \\ = & \quad \{\ \text{def. of } gemm, \mathsf{map}\ \} \\ & gemm(\oplus', \otimes')\ (\mathsf{map}\ f\ (x \ominus y))\ (\mathsf{map}\ f\ z) \end{array}$$

Rule IV

$$\begin{array}{l} \mathsf{map}\ f\ (\mathsf{map}(\oplus x)\ y) = \mathsf{map}\ (\otimes'(f\ x))(\mathsf{map}\ f\ y) \\ \Leftarrow \forall a, b\ \ f\ (a \oplus b) = f\ a \oplus' f\ b \end{array}$$

Proof: It is proved by induction on the structure of abide trees.

$$
\begin{array}{ll}
 & \mathsf{map}\ f\ (\mathsf{map}(\oplus x)\ |a|) \\
= & \quad \{\ \text{def. of } \mathsf{map}\ \} \\
 & |f\ (x \oplus a)| \\
= & \quad \{\ \text{hypo.}\ \} \\
 & |f\ x \oplus' f\ a)| \\
= & \quad \{\ \text{def. of } \mathsf{map}\ \} \\
 & \mathsf{map}\ (\otimes'(f\ x))(\mathsf{map}\ f\ y)
\end{array}
$$

$$
\begin{array}{ll}
 & \mathsf{map}\ f\ (\mathsf{map}(\oplus x)\ (y \phi z)) \\
= & \quad \{\ \text{def. of } \mathsf{map}\ \} \\
 & \mathsf{map}\ f\ (\mathsf{map}(\oplus x)\ y) \phi \mathsf{map}\ f\ (\mathsf{map}(\oplus x)\ z) \\
= & \quad \{\ \text{hypo. of induction}\ \} \\
 & \mathsf{map}\ (\otimes'(f\ x))(\mathsf{map}\ f\ y) \phi \mathsf{map}\ (\otimes'(f\ x))(\mathsf{map}\ f\ z) \\
= & \quad \{\ \text{def. of } \mathsf{map}\ \} \\
 & \mathsf{map}\ (\otimes'(f\ x))(\mathsf{map}\ f\ (y \phi z))
\end{array}
$$

The inductive case for $\ominus$ is proved similarly.

The following is an instance of this rule:

$$\mathsf{map}\ \mathit{sum}\ (\mathsf{zipwith}\ (\ominus)\ a\ b) = \mathsf{zipwith}(+)\ (\mathsf{map}\ \mathit{sum}\ a)\ (\mathsf{map}\ \mathit{sum}\ b)$$

Rule V

$$\mathsf{map}\ f\ (\mathit{right}'\ x) = \mathit{right}'(\mathsf{map}\ (\mathsf{map}\ f)\ x)$$

Proof:

$$
\begin{array}{ll}
 & \mathsf{map}\ f \circ \mathit{right}' \\
= & \quad \{\ \text{def. of } \mathit{right}'\ \} \\
 & \mathsf{map}\ f \circ \mathsf{the} \circ \mathit{right} \\
= & \quad \{\ \text{def. of } \mathit{right}\ \} \\
 & \mathsf{map}\ f \circ \mathsf{the} \circ \mathsf{reduce}(\ominus, \gg) \circ \mathsf{map}\ |\cdot| \\
= & \quad \{\ \text{def. of } \mathsf{the}, \mathsf{map}\ \} \\
 & \mathsf{the} \circ \mathsf{map}\ (\mathsf{map}\ f) \circ \mathsf{reduce}(\ominus, \gg) \circ \mathsf{map}\ |\cdot| \\
= & \quad \{\ \text{VI}\ \} \\
 & \mathsf{the} \circ \mathsf{reduce}(\ominus, \gg) \circ \mathsf{map}\ (\mathsf{map}\ (\mathsf{map}\ f))\ \ \mathsf{map}\ |\cdot| \\
= & \quad \{\ \text{def. of } |\cdot|, \mathsf{map}\ \} \\
 & \mathsf{the} \circ \mathsf{reduce}(\ominus, \gg) \circ \mathsf{map}\ |\cdot| \circ \mathsf{map}\ (\mathsf{map}\ f)) \\
= & \quad \{\ \text{def. of } \mathit{right}'\ \} \\
 & \mathit{right}' \circ \mathsf{map}\ (\mathsf{map}\ f)
\end{array}
$$

This rule for top' holds similarly.

Rule VI

$$
\begin{array}{l}
\mathsf{map}\ f\ \circ \mathsf{reduce}(\oplus, \otimes) = \mathsf{reduce}(\oplus, \otimes) \circ \mathsf{map}\ (\mathsf{map}\ f) \\
\Leftarrow \oplus, \otimes \in \{\ominus, \phi, \ll, \gg\}
\end{array}
$$

Proof:

$$
\begin{array}{ll}
 & \mathsf{map}\ f\ (\mathsf{reduce}(\oplus,\otimes)\ |a|) \\
= & \quad \{ \text{ def. of } \mathsf{reduce} \} \\
 & \mathsf{map}\ f\ a \\
= & \quad \{ \text{ def. of } \mathsf{reduce} \} \\
 & \mathsf{reduce}(\oplus,\otimes)\ |\mathsf{map}\ f\ a| \\
= & \quad \{ \text{ def. of } \mathsf{map} \} \\
 & \mathsf{reduce}(\oplus,\otimes)\ (\mathsf{map}\ (\mathsf{map}\ f)\ |a|)
\end{array}
$$

$$
\begin{array}{ll}
 & \mathsf{map}\ f\ (\mathsf{reduce}(\oplus,\otimes)\ (x \phi y)) \\
= & \quad \{ \text{ def. of } \mathsf{reduce} \} \\
 & \mathsf{map}\ f\ (\mathsf{reduce}(\oplus,\otimes)\ x \otimes \mathsf{reduce}(\oplus,\otimes) y) \\
= & \quad \{ \text{ below } \} \\
 & \mathsf{map}\ f\ (\mathsf{reduce}(\oplus,\otimes)\ x) \otimes \mathsf{map}\ f\ (\mathsf{reduce}(\oplus,\otimes) y) \\
= & \quad \{ \text{ hypo. of induction } \} \\
 & \mathsf{reduce}(\oplus,\otimes)\ (\mathsf{map}\ (\mathsf{map}\ f)\ x) \otimes \mathsf{reduce}(\oplus,\otimes)\ (\mathsf{map}\ (\mathsf{map}\ f)\ y) \\
= & \quad \{ \text{ def. of } \mathsf{map}, \mathsf{reduce} \} \\
 & \mathsf{reduce}(\oplus,\otimes)\ (\mathsf{map}\ (\mathsf{map}\ f)\ (x \phi y))
\end{array}
$$

The inductive case for $\ominus$ is proved similarly.

$$\mathsf{map}\ f\ (x \oplus y) = \mathsf{map}\ f\ x \oplus \mathsf{map}\ f\ y \Leftarrow \oplus \in \{\ominus, \phi, \ll, \gg\}$$

Proof:

$$
\begin{array}{l}
\mathsf{map}\ f\ (x \phi y) = \mathsf{map}\ f\ x \phi \mathsf{map}\ f\ y \\
\mathsf{map}\ f\ (x \ominus y) = \mathsf{map}\ f\ x \ominus \mathsf{map}\ f\ y \\
\mathsf{map}\ f\ (x \gg y) = \mathsf{map}\ f\ y = \mathsf{map}\ f\ x \gg \mathsf{map}\ f\ y \\
\mathsf{map}\ f\ (x \ll y) = \mathsf{map}\ f\ x = \mathsf{map}\ f\ x \ll \mathsf{map}\ f\ y
\end{array}
$$

Rule VII

$$
\begin{array}{l}
\mathsf{map}\ f\ (\mathsf{zipwith}_4\ g\ x\ u\ w\ a) \\
\qquad = \mathsf{zipwith}_4\ g'\ (\mathsf{map}\ f_1\ x)\ (\mathsf{map}\ f_2\ u)\ (\mathsf{map}\ f_3\ w)\ (\mathsf{map}\ f_4\ a) \\
\Leftarrow f\ (g\ x\ u\ w\ a) = g'\ (f_1\ x)\ (f_2\ u)\ (f_3\ w)\ (f_4\ a)
\end{array}
$$

Proof: It is proved by induction on the structure of abide trees.

$$
\begin{array}{ll}
 & \mathsf{map}\ f\ (\mathsf{zipwith}_4\ g\ |a|\ |b|\ |c|\ |d|) \\
= & \quad \{ \text{ def. of } \mathsf{zipwith}, \mathsf{map} \} \\
 & |f\ (g\ a\ b\ c\ d)| \\
= & \quad \{ \text{ hypo. } \} \\
 & |g'\ (f_1\ a)\ (f_2\ b)\ (f_3\ c)\ (f_4\ d)| \\
= & \quad \{ \text{ def. of } \mathsf{zipwith}, \mathsf{map} \} \\
 & \mathsf{zipwith}_4\ g'\ (\mathsf{map}\ f_1\ |a|)\ (\mathsf{map}\ f_2\ |b|)\ (\mathsf{map}\ f_3\ |c|)\ (\mathsf{map}\ f_4\ |d|)
\end{array}
$$

$$
\begin{array}{ll}
 & \mathsf{map}\ f\ (\mathsf{zipwith}_4\ g\ (a \phi x)\ (b \phi y)\ (c \phi z)\ (d \phi w)) \\
= & \quad \{ \text{ def. of } \mathsf{zipwith}, \mathsf{map} \} \\
 & map\ f\ (\mathsf{zipwith}_4\ g\ a\ b\ c\ d) \phi \mathsf{map}\ f\ (\mathsf{zipwith}_4\ g\ x\ y\ z\ w) \\
= & \quad \{ \text{ hypo. of induction } \} \\
 & \mathsf{zipwith}_4\ g'\ (\mathsf{map}\ f_1\ a)\ (\mathsf{map}\ f_2\ b)\ (\mathsf{map}\ f_3\ c)\ (\mathsf{map}\ f_4\ d) \\
 & \phi\ \mathsf{zipwith}_4\ g'\ (\mathsf{map}\ f_1\ x)\ (\mathsf{map}\ f_2\ y)\ (\mathsf{map}\ f_3\ z)\ (\mathsf{map}\ f_4\ w) \\
= & \quad \{ \text{ def. of } \mathsf{zipwith}, \mathsf{map} \} \\
 & \mathsf{zipwith}_4\ g'\ (\mathsf{map}\ f_1\ (a \phi x))\ (\mathsf{map}\ f_2\ (b \phi y))\ (\mathsf{map}\ f_3\ (c \phi z))\ (\mathsf{map}\ f_4\ (d \phi w))
\end{array}
$$

The inductive case for $\ominus$ is proved similarly.

Rule VIII

$$sum \circ (\phi x) = (+(sum\ x)) \circ sum$$

Proof:

$$(sum \circ (\phi x))\ y = sum\ (y \phi x) = sum\ y + sum\ x = ((+(sum\ x)) \circ sum)\ y$$

Rule IX

$$\begin{array}{ll}
 & \mathsf{map}\ \ sum(\mathsf{map}\ (\phi\ \ top'\ tr_2)\ tr_1 \ominus tr_2) \\
= & \qquad \{ \text{ def. of } \mathsf{map} \ \} \\
 & \mathsf{map}\ \ sum(\mathsf{map}\ (\phi\ \ top'\ tr_2)\ tr_1) \ominus \mathsf{map}\ \ sum tr_2 \\
= & \qquad \{ \text{ def. of } \mathsf{map} \ \} \\
 & \mathsf{map}\ (sum \circ (\phi\ \ top'\ tr_2))\ tr_1 \ominus \mathsf{map}\ \ sum tr_2 \\
= & \qquad \{ \text{ V, VIII } \} \\
 & \mathsf{map}\ (+\ top'\ (\mathsf{map}\ \ sum\ tr_2))\ (\mathsf{map}\ \ sum\ tr_1) \ominus \mathsf{map}\ \ sum tr_2
\end{array}$$

Rule X

$$\begin{array}{l}
\mathsf{reduce}(\oplus,\oplus)(\mathsf{zipwith}(\otimes)\ (\mathsf{zipwith}(\oplus)\ a\ b)\ \ (\mathsf{zipwith}(\oplus)\ c\ d)) \\
= \mathsf{reduce}(\oplus,\oplus)(\mathsf{zipwith}(\otimes)\ a\ c) \oplus \mathsf{reduce}(\oplus,\oplus)(\mathsf{zipwith}(\otimes)\ a\ d) \\
\ \ \oplus \mathsf{reduce}(\oplus,\oplus)(\mathsf{zipwith}(\otimes)\ b\ c) \oplus \mathsf{reduce}(\oplus,\oplus)(\mathsf{zipwith}(\otimes)\ b\ d) \\
\Leftarrow (a \oplus b) \otimes (c \oplus d) = (a \otimes c) \oplus (a \otimes d) \oplus (b \otimes c) \oplus (b \otimes d)
\end{array}$$

Proof: It is proved by induction on the structure of abide trees.

$$\begin{array}{ll}
 & \mathsf{reduce}(\oplus,\oplus)(\mathsf{zipwith}(\otimes)\ (\mathsf{zipwith}(\oplus)\ |a|\ |b|))\ (\mathsf{zipwith}(\otimes)\ (\mathsf{zipwith}(\oplus)\ |c|\ |d|)) \\
= & \qquad \{ \text{ def. of } \mathsf{zipwith}, \mathsf{reduce} \ \} \\
 & (a \oplus b) \otimes (c \oplus d) \\
= & \qquad \{ \text{ hypo. } \} \\
 & (a \otimes c) \oplus (a \otimes d) \oplus (b \otimes c) \oplus (b \otimes d) \\
= & \qquad \{ \text{ def. of } \mathsf{zipwith}, \mathsf{reduce} \ \} \\
 & \mathsf{reduce}(\oplus,\oplus)(\mathsf{zipwith}(\otimes)\ |a|\ |c|) \oplus \mathsf{reduce}(\oplus,\oplus)(\mathsf{zipwith}(\otimes)\ |a|\ |d|) \\
 & \qquad \oplus \mathsf{reduce}(\oplus,\oplus)(\mathsf{zipwith}(\otimes)\ |b|\ |c|) \oplus \mathsf{reduce}(\oplus,\oplus)(\mathsf{zipwith}(\otimes)\ |b|\ |d|)
\end{array}$$

$$
\begin{aligned}
&\quad \mathsf{reduce}(\oplus,\oplus)\\
&\quad\ (\mathsf{zipwith}(\otimes)\,(\mathsf{zipwith}(\oplus)\,(a_1 \phi a_2)\,(b_1 \phi b_2)))\,(\mathsf{zipwith}(\otimes)\,(\mathsf{zipwith}(\oplus)\,(c_1 \phi c_2)\,(d_1 \phi d_2)))\\
= &\qquad \{\ \text{def. of } \mathsf{zipwith}, \mathsf{reduce}\ \}\\
&\quad \mathsf{reduce}(\oplus,\oplus)(\mathsf{zipwith}(\otimes)\ (\mathsf{zipwith}(\oplus)\ a_1\ b_1))\ (\mathsf{zipwith}(\otimes)\ (\mathsf{zipwith}(\oplus)\ c_1\ d_1))\\
&\qquad \oplus \mathsf{reduce}(\oplus,\oplus)(\mathsf{zipwith}(\otimes)\ (\mathsf{zipwith}(\oplus)\ a_2\ b_2))\ (\mathsf{zipwith}(\otimes)\ (\mathsf{zipwith}(\oplus)\ c_2\ d_2))\\
= &\qquad \{\ \text{hypo. of induction}\ \}\\
&\quad \mathsf{reduce}(\oplus,\oplus)(\mathsf{zipwith}(\otimes)\ a_1\ c_1) \oplus \mathsf{reduce}(\oplus,\oplus)(\mathsf{zipwith}(\otimes)\ a_1\ d_1)\\
&\qquad \oplus \mathsf{reduce}(\oplus,\oplus)(\mathsf{zipwith}(\otimes)\ b_1\ c_1) \oplus \mathsf{reduce}(\oplus,\oplus)(\mathsf{zipwith}(\otimes)\ b_1\ d_1)\\
&\qquad\quad \oplus \mathsf{reduce}(\oplus,\oplus)(\mathsf{zipwith}(\otimes)\ a_2\ c_2) \oplus \mathsf{reduce}(\oplus,\oplus)(\mathsf{zipwith}(\otimes)\ a_2\ d_2)\\
&\qquad\qquad \oplus \mathsf{reduce}(\oplus,\oplus)(\mathsf{zipwith}(\otimes)\ b_2\ c_2) \oplus \mathsf{reduce}(\oplus,\oplus)(\mathsf{zipwith}(\otimes)\ b_2\ d_2)\\
= &\qquad \{\ \text{def. of } \mathsf{zipwith}, \mathsf{reduce}\ \}\\
&\quad \mathsf{reduce}(\oplus,\oplus)(\mathsf{zipwith}(\otimes)\ (a_1 \phi a_2)\ (c_1 \phi c_2))\\
&\quad\ \oplus \mathsf{reduce}(\oplus,\oplus)(\mathsf{zipwith}(\otimes)\ (a_1 \phi a_2)\ (d_1 \phi d_2))\\
&\qquad \oplus \mathsf{reduce}(\oplus,\oplus)(\mathsf{zipwith}(\otimes)\ (b_1 \phi b_2)\ (c_1 \phi c_2))\\
&\qquad\ \oplus \mathsf{reduce}(\oplus,\oplus)(\mathsf{zipwith}(\otimes)\ (b_1 \phi b_2)\ (d_1 \phi d_2))
\end{aligned}
$$

The inductive case for $\ominus$ is proved similarly.

Rule XI

$$
\begin{aligned}
&max\ (\mathsf{map}\ \ max\ (gemm(_,\mathsf{zipwith}(+))\ b\ t))\\
&= max\ (\mathsf{zipwith}(+)\ (\mathsf{reduce}\ (\mathsf{zipwith}(\uparrow),_)\ b)\ (\mathsf{reduce}\ (_,\mathsf{zipwith}(\uparrow))\ t))\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \Leftarrow width\ b = 1, height\ t = 1
\end{aligned}
$$

Proof:It is proved by induction on the structure of abide trees.

$$
\begin{aligned}
&\quad max\ (\mathsf{map}\ \ max\ (gemm(_,\mathsf{zipwith}(+))\ |b|\ |t|))\\
= &\qquad \{\ \text{def. of } gemm\ \}\\
&\quad max\ (\mathsf{map}\ \ max\ (|\,\mathsf{zipwith}(+)\ b\ t|))\\
= &\qquad \{\ \text{def. of } \mathsf{zipwith}, \mathsf{map}, max\ \}\\
&\quad max\ (\mathsf{zipwith}(+)\ b\ t)\\
= &\qquad \{\ \text{def. of } \mathsf{reduce}\ \}\\
&\quad max\ (\mathsf{zipwith}(+)\ (\mathsf{reduce}\ (\mathsf{zipwith}(\uparrow),_)\ |b|)\ (\mathsf{reduce}\ (_,\mathsf{zipwith}(\uparrow))\ |t|))
\end{aligned}
$$

$$
\begin{aligned}
&\quad max\ (\mathsf{map}\ \ max\ (gemm(_,\mathsf{zipwith}(+))\ (b_1 \ominus b_2)\ (t_1 \phi t_2)))\\
= &\qquad \{\ \text{def. of } gemm\ \}\\
&\quad max\ (\mathsf{map}\ \ max\ (((gemm(_,\mathsf{zipwith}(+))\ b_1\ t_1) \phi (gemm(_,\mathsf{zipwith}(+))\ b_1\ t_2))\\
&\qquad \ominus((gemm(_,\mathsf{zipwith}(+))\ b_2\ t_1) \phi (gemm(_,\mathsf{zipwith}(+))\ b_2\ t_2))))\\
= &\qquad \{\ \text{def. of } max\ \}\\
&\quad max\ (\mathsf{map}\ \ max\ (gemm(_,\mathsf{zipwith}(+))\ b_1\ t_1)) \uparrow\\
&\qquad max\ (\mathsf{map}\ \ max\ (gemm(_,\mathsf{zipwith}(+))\ b_1\ t_2))\\
&\qquad\ \uparrow max\ (\mathsf{map}\ \ max\ (gemm(_,\mathsf{zipwith}(+))\ b_2\ t_1))\\
&\qquad\quad \uparrow max\ (\mathsf{map}\ \ max\ (gemm(_,\mathsf{zipwith}(+))\ b_2\ t_2))\\
= &\qquad \{\ \text{hypo. of induction}\ \}\\
&\quad max\ (\mathsf{zipwith}(+)\ (\mathsf{reduce}\ (\mathsf{zipwith}(\uparrow),_)\ b_1)\ (\mathsf{reduce}\ (_,\mathsf{zipwith}(\uparrow))\ t_1))\\
&\qquad \uparrow max\ (\mathsf{zipwith}(+)\ (\mathsf{reduce}\ (\mathsf{zipwith}(\uparrow),_)\ b_1)\ (\mathsf{reduce}\ (_,\mathsf{zipwith}(\uparrow))\ t_2))\\
&\qquad\quad \uparrow max\ (\mathsf{zipwith}(+)\ (\mathsf{reduce}\ (\mathsf{zipwith}(\uparrow),_)\ b_2)\ (\mathsf{reduce}\ (_,\mathsf{zipwith}(\uparrow))\ t_1))\\
&\qquad\qquad \uparrow max\ (\mathsf{zipwith}(+)\ (\mathsf{reduce}\ (\mathsf{zipwith}(\uparrow),_)\ b_2)\ (\mathsf{reduce}\ (_,\mathsf{zipwith}(\uparrow))\ t_2))\\
= &\qquad \{\ \text{X with } \oplus = \uparrow, \otimes = +\ \}\\
&\quad max\ (\mathsf{zipwith}(+)\ (\mathsf{reduce}\ (\mathsf{zipwith}(\uparrow),_)\ (b_1 \ominus b_2))\ (\mathsf{reduce}\ (_,\mathsf{zipwith}(\uparrow))\ (t_1 \phi t_2)))
\end{aligned}
$$

Rule XII

$$
\begin{array}{l}
max(\mathsf{zipwith}_4\ \ f_s\ s_1\ s_2\ r_1\ l_2)\\
= max\ s_1 \uparrow max\ s_2 \uparrow\ max\ (\mathsf{zipwith}(+)\ (\mathsf{map}\ \ \mathsf{reduce}(\uparrow, _)\ r_1)\ (\mathsf{map}\ \ \mathsf{reduce}(_, \uparrow)\ l_2))\\
\quad \mathbf{where}\ \ f_s\ s_1\ s_2\ r_1\ l_2 = s_1 \uparrow max\ (gemm(_, +)\ r_1\ l_2) \uparrow s_2\\
\Leftarrow width \text{ of elements of } r_1 = 1, height \text{ of elements of } l_2 = 1
\end{array}
$$

Proof: First, we prove the following equation by the induction on the structure of abide trees.

$$
\begin{array}{l}
max(\mathsf{zipwith}_4\ \ f_s\ s_1\ s_2\ r_1\ l_2) = max\ s_1 \uparrow max\ s_2 \uparrow max\ (\mathsf{zipwith}\ \ f'_s\ r_1\ l_2)\\
\mathbf{where}\ \ f'_s\ r_1\ l_2 = max\ (gemm(_, +)\ r_1\ l_2)
\end{array}
$$

Proof:

$$
\begin{array}{ll}
 & max(\mathsf{zipwith}_4\ \ f_s\ |s_1|\ |s_2|\ |r_1|\ |l_2|)\\
= & \quad \{ \text{ def. of } f_s, \mathsf{zipwith}\ \}\\
 & s_1 \uparrow max\ (gemm(_, +)\ r_1\ l_2) \uparrow s_2\\
= & \quad \{ \text{ def. of } f'_s, max, \text{ associativity of } \uparrow\ \}\\
 & max\ |s_1| \uparrow max\ |s_2| \uparrow max\ (\mathsf{zipwith}\ \ f'_s\ |r_1|\ |l_2|)
\end{array}
$$

$$
\begin{array}{ll}
 & max(\mathsf{zipwith}_4\ \ f_s\ (s_1^1 \phi\ s_1^2)\ (s_2^1 \phi\ s_2^2)\ (r_1^1 \phi\ r_1^2)\ (l_2^1 \phi\ l_2^2))\\
= & \quad \{ \text{ def. of } max, \mathsf{zipwith}\ \}\\
 & max(\mathsf{zipwith}_4\ \ f_s\ s_1^1\ s_2^1\ r_1^1\ l_2^1) \uparrow max(\mathsf{zipwith}_4\ \ f_s\ s_1^2\ s_2^2\ r_1^2\ l_2^2)\\
= & \quad \{ \text{ hypo. of induction } \}\\
 & max\ s_1^1 \uparrow max\ s_2^1 \uparrow max\ (\mathsf{zipwith}\ \ f'_s\ r_1^1\ l_2^1)\\
 & \quad \uparrow max\ s_1^2 \uparrow max\ s_2^2 \uparrow max\ (\mathsf{zipwith}\ \ f'_s\ r_1^2\ l_2^2)\\
= & \quad \{ \text{ def. of } f'_s, max, \text{ associativity of } \uparrow\ \}\\
 & max\ (s_1^1 \phi\ s_1^2) \uparrow max\ (s_2^1 \phi\ s_2^2) \uparrow max\ (\mathsf{zipwith}\ \ f'_s\ (r_1^1 \phi\ r_1^2)\ (l_2^1 \phi\ l_2^2))
\end{array}
$$

The inductive case for $\ominus$ is proved similarly.

Then, we prove the following equation by the induction on the structure of abide trees.

$$
max\ (\mathsf{zipwith}\ \ f'_s\ r_1\ l_2) = max\ (\mathsf{zipwith}(+)\ (\mathsf{map}\ \ \mathsf{reduce}(\uparrow, _)\ r_1)\ (\mathsf{map}\ \ \mathsf{reduce}(_, \uparrow)\ l_2))
$$

Proof:

$$
\begin{array}{ll}
 & max(\mathsf{zipwith}\ \ f'_s\ |r_1|\ |l_2|)\\
= & \quad \{ \text{ def. of } max, \mathsf{zipwith}, f'_s\ \}\\
 & max\ |\max\ (gemm(_, +)\ r_1\ l_2)|\\
= & \quad \{ \text{ below } \}\\
 & max\ (|\,\mathsf{reduce}(\uparrow, _)\ r_1 + \mathsf{reduce}(\uparrow, _)\ l_2|)\\
= & \quad \{ \text{ def. of } \mathsf{zipwith}, \mathsf{map}\ \}\\
 & max\ (\mathsf{zipwith}(+)\ (\mathsf{map}\ \ \mathsf{reduce}(\uparrow, _)\ |r_1|)\ (\mathsf{map}\ \ \mathsf{reduce}(_, \uparrow)\ |l_2|))
\end{array}
$$

$$
\begin{array}{ll}
 & max(\mathsf{zipwith}\ \ f'_s\ (r_1^1 \phi\ r_1^2)\ (l_2^1 \phi\ l_2^2))\\
= & \quad \{ \text{ def. of } max, \mathsf{zipwith}\ \}\\
 & max(\mathsf{zipwith}\ \ f'_s\ r_1^1\ l_2^1) \uparrow max(\mathsf{zipwith}\ \ f'_s\ r_1^2\ l_2^2)\\
= & \quad \{ \text{ hypo. of induction } \}\\
 & max\ (\mathsf{zipwith}(+)\ (\mathsf{map}\ \ \mathsf{reduce}(\uparrow, _)\ r_1^1)\ (\mathsf{map}\ \ \mathsf{reduce}(_, \uparrow)\ l_2^1))\\
 & \quad \uparrow max\ (\mathsf{zipwith}(+)\ (\mathsf{map}\ \ \mathsf{reduce}(\uparrow, _)\ r_1^2)\ (\mathsf{map}\ \ \mathsf{reduce}(_, \uparrow)\ l_2^2))\\
= & \quad \{ \text{ def. of } max, \mathsf{zipwith}, \mathsf{map}\ \}\\
 & max\ (\mathsf{zipwith}(+)\ (\mathsf{map}\ \ \mathsf{reduce}(\uparrow, _)\ (r_1^1 \phi\ r_1^2))\ (\mathsf{map}\ \ \mathsf{reduce}(_, \uparrow)\ (l_2^1 \phi\ l_2^2)))
\end{array}
$$

To complete the proof of the base case, we prove the next equation by the induction on the structure of abide trees.

$$max\ (gemm(_,+)\ r_1\ l_2) = \mathsf{reduce}(\uparrow,_)\ r_1 + \mathsf{reduce}(\uparrow,_)\ l_2$$
$$\Leftarrow width\ r_1 = 1, height\ l_2 = 1$$

Proof:

$$\begin{array}{ll} & max\ (gemm(_,+)\ |r_1|\ |l_2|) \\ = & \quad \{\ \text{def. of } gemm, max\ \} \\ & r_1 + l_2 \\ = & \quad \{\ \text{def. of } \mathsf{reduce}\ \} \\ & \mathsf{reduce}(\uparrow,_)\ |r_1| + \mathsf{reduce}(\uparrow,_)\ |l_2| \end{array}$$

$$\begin{array}{ll} & max\ (gemm(_,+)\ (r_1^1 \ominus r_1^2)\ (l_2^1 \phi\ l_2^2)) \\ = & \quad \{\ \text{def. of } gemm, max\ \} \\ & max\ (gemm(_,+)\ r_1^1\ l_2^1) \uparrow max\ (gemm(_,+)\ r_1^1\ l_2^2) \\ & \quad \uparrow max\ (gemm(_,+)\ r_1^2\ l_2^1) \uparrow max\ (gemm(_,+)\ r_1^2\ l_2^2) \\ = & \quad \{\ \text{hypo. of induction}\ \} \\ & (\mathsf{reduce}(\uparrow,_)\ r_1^1 + \mathsf{reduce}(\uparrow,_)\ l_2^1) \uparrow (\mathsf{reduce}(\uparrow,_)\ r_1^1 + \mathsf{reduce}(\uparrow,_)\ l_2^2) \\ & \quad \uparrow (\mathsf{reduce}(\uparrow,_)\ r_1^2 + \mathsf{reduce}(\uparrow,_)\ l_2^1) \uparrow (\mathsf{reduce}(\uparrow,_)\ r_1^2 + \mathsf{reduce}(\uparrow,_)\ l_2^2) \\ = & \quad \{\ \text{associativity and distributivity}\ \} \\ & (\mathsf{reduce}(\uparrow,_)\ r_1^1 \uparrow \mathsf{reduce}(\uparrow,_)\ r_1^2) + (\mathsf{reduce}(\uparrow,_)\ l_2^1 \uparrow \mathsf{reduce}(\uparrow,_)\ l_2^2) \\ = & \quad \{\ \text{def. of } \mathsf{reduce}\ \} \\ & \mathsf{reduce}(\uparrow,_)\ (r_1^1 \ominus r_1^2) + \mathsf{reduce}(\uparrow,_)\ (l_2^1 \phi\ l_2^2) \end{array}$$

Rule XIII

$$\mathsf{reduce}(\oplus,\otimes)\ (\mathsf{map}\ \ f\ x)$$
$$= f\ (\mathsf{reduce}(\oplus,\otimes)\ x) \Leftarrow f\ a \otimes f\ b = f\ (a \otimes b), f\ a \oplus f\ b = f\ (a \oplus\ b)$$

Proof: It is proved by induction on the structure of abide trees.

$$\begin{array}{ll} & \mathsf{reduce}(\oplus,\otimes)\ (\mathsf{map}\ \ f\ |x|) \\ = & \quad \{\ \text{def. of } \mathsf{reduce}, \mathsf{map}\ \} \\ & f\ x \\ = & \quad \{\ \text{def. of } \mathsf{reduce}\ \} \\ & f\ (\mathsf{reduce}(\oplus,\otimes)\ |x|) \end{array}$$

$$\begin{array}{ll} & \mathsf{reduce}(\oplus,\otimes)\ (\mathsf{map}\ \ f\ (x \ominus y)) \\ = & \quad \{\ \text{def. of } \mathsf{reduce}, \mathsf{map}\ \} \\ & \mathsf{reduce}(\oplus,\otimes)\ (\mathsf{map}\ \ f\ x)) \oplus \mathsf{reduce}(\oplus,\otimes)\ (\mathsf{map}\ \ f\ y) \\ = & \quad \{\ \text{hypo. of induction}\ \} \\ & f\ (\mathsf{reduce}(\oplus,\otimes)\ x) \oplus f\ (\mathsf{reduce}(\oplus,\otimes)\ y) \\ = & \quad \{\ \text{hypo.}\ \} \\ & f\ ((\mathsf{reduce}(\oplus,\otimes)\ x) \oplus (\mathsf{reduce}(\oplus,\otimes)\ y)) \\ = & \quad \{\ \text{def. of } \mathsf{reduce}\ \} \\ & f\ (\mathsf{reduce}(\oplus,\otimes)\ (x \phi\ y)) \end{array}$$

The inductive case for $\ominus$ is proved similarly.

For instance, $\oplus = _$(don't care), $\otimes = \mathsf{zipwith}(\uparrow)$ and $f = \mathsf{zipwith}(+)\ c_1$ satisfy the condition $f\ a \otimes f\ b = f\ (a \otimes b)$.

Rule XIV

$$
\begin{aligned}
&\mathsf{reduce}(\oplus,\otimes)\ (\mathsf{zipwith}_4\ \ f\ x\ y\ z\ w)\\
&\quad = f'\ (\mathsf{reduce}(\oplus_1,\otimes_1)\ x)\ (\mathsf{reduce}(\oplus_2,\otimes_2)\ y)\ (\mathsf{reduce}(\oplus_3,\otimes_3)\ z)\ (\mathsf{reduce}(\oplus_4,\otimes_4)\ w)\\
&\quad \Leftarrow f\ a\ b\ c\ d = f'\ a\ b\ c\ d,\\
&\qquad f'\ a\ b\ c\ d \oplus f'\ x\ y\ z\ w = f'\ (a \oplus_1 x)\ (b \oplus_2 y)\ (c \oplus_3 z)\ (d \oplus_4 w)\\
&\qquad f'\ a\ b\ c\ d \otimes f'\ x\ y\ z\ w = f'\ (a \otimes_1 x)\ (b \otimes_2 y)\ (c \otimes_3 z)\ (d \otimes_4 w)
\end{aligned}
$$

Proof: It is proved by induction on the structure of abide trees.

$$
\begin{aligned}
&\quad \mathsf{reduce}(\oplus,\otimes)\ (\mathsf{zipwith}_4\ \ f\ |x|\ |y|\ |z|\ |w|)\\
&= \qquad \{\text{ def. of } \mathsf{reduce}, \mathsf{zipwith}\ \}\\
&\quad f\ x\ y\ z\ w\\
&= \qquad \{\text{ hypo. }\}\\
&\quad f'\ x\ y\ z\ w\\
&= \qquad \{\text{ def. of } \mathsf{reduce}\ \}\\
&\quad f'\ (\mathsf{reduce}(\oplus_1,\otimes_1)\ |x|)\ (\mathsf{reduce}(\oplus_2,\otimes_2)\ |y|)\ (\mathsf{reduce}(\oplus_3,\otimes_3)\ |z|)\ (\mathsf{reduce}(\oplus_4,\otimes_4)\ |w|)
\end{aligned}
$$

$$
\begin{aligned}
&\quad \mathsf{reduce}(\oplus,\otimes)\ (\mathsf{zipwith}_4\ \ f\ (x_1 \ominus x_2)\ (y_1 \ominus y_2)\ (z_1 \ominus z_2)\ (w_1 \ominus w_2))\\
&= \qquad \{\text{ def. of } \mathsf{reduce}, \mathsf{zipwith}\ \}\\
&\quad \mathsf{reduce}(\oplus,\otimes)\ (\mathsf{zipwith}_4\ \ f\ x_1\ y_1\ z_1\ w_1) \oplus \mathsf{reduce}(\oplus,\otimes)\ (\mathsf{zipwith}_4\ \ f\ x_2\ y_2\ z_2\ w_2)\\
&= \qquad \{\text{ hypo. of induction }\}\\
&\quad f'\ (\mathsf{reduce}(\oplus_1,\otimes_1)\ x_1)\ (\mathsf{reduce}(\oplus_2,\otimes_2)\ y_1)\ (\mathsf{reduce}(\oplus_3,\otimes_3)\ z_1)\ (\mathsf{reduce}(\oplus_4,\otimes_4)\ w_1)\\
&\quad\ \oplus f'\ (\mathsf{reduce}(\oplus_1,\otimes_1)\ x_2)\ (\mathsf{reduce}(\oplus_2,\otimes_2)\ y_2)\ (\mathsf{reduce}(\oplus_3,\otimes_3)\ z_2)\ (\mathsf{reduce}(\oplus_4,\otimes_4)\ w_2)\\
&= \qquad \{\text{ hypo. }\}\\
&\quad f'\ (\mathsf{reduce}(\oplus_1,\otimes_1)\ x_1 \oplus_1 \mathsf{reduce}(\oplus_1,\otimes_1)\ x_2)\ (\mathsf{reduce}(\oplus_2,\otimes_2)\ y_1 \oplus_2 \mathsf{reduce}(\oplus_2,\otimes_2)\ y_2)\\
&\qquad (\mathsf{reduce}(\oplus_3,\otimes_3)\ z_1 \oplus_3 \mathsf{reduce}(\oplus_3,\otimes_3)\ z_3)\ (\mathsf{reduce}(\oplus_4,\otimes_4)\ w_1 \oplus_4 \mathsf{reduce}(\oplus_4,\otimes_4)\ w_2)\\
&= \qquad \{\text{ def. of } \mathsf{reduce}\ \}\\
&\quad f'\ (\mathsf{reduce}(\oplus_1,\otimes_1)\ (x_1 \ominus x_2))\ (\mathsf{reduce}(\oplus_2,\otimes_2)\ (y_1 \ominus y_2))\\
&\qquad\qquad\qquad\qquad (\mathsf{reduce}(\oplus_3,\otimes_3)\ (z_1 \ominus z_2))\ (\mathsf{reduce}(\oplus_4,\otimes_4)\ (w_1 \ominus w_2))
\end{aligned}
$$

The inductive case for ϕ is proved similarly.

For instance, $f'\ a\ b\ c\ d = (a \phi\ gemm(\uparrow,+)\ c\ d) \ominus (NIL \phi\ b)$, $\otimes_1 = \mathsf{zipwith}(\uparrow)$, $\otimes_2 = \mathsf{zipwith}(\uparrow)$, $\otimes_3 = \phi$ and $\otimes_4 = \ominus$ satisfy the condition for $f\ a\ b\ c\ d = (a \phi\ gemm(_,+)\ c\ d) \ominus (NIL \phi\ b)$, $\otimes = \mathsf{zipwith}(\uparrow)$ and $\oplus = _$.

Rule XV

$$
\begin{aligned}
&\mathsf{map}\ (\mathsf{reduce}(\oplus,_))\ (gemm(_,\mathsf{zipwith}(\otimes))\ x\ y)\\
&= gemm(\oplus,\otimes)\ (tr\ (\mathsf{reduce}\ (\phi,_))\ x)\ (\mathsf{reduce}\ (_,\phi)\ y)\\
&\ \Leftarrow width \text{ of } x \text{ and its elements} = 1, width \text{ of } y\text{'s elements} = 1, height\ y = 1
\end{aligned}
$$

Proof: It is proved by induction on the structure of abide trees.

$$
\begin{aligned}
&\quad \mathsf{map}\ (\mathsf{reduce}(\oplus,_))\ (gemm(_,\mathsf{zipwith}(\otimes))\ |x|\ |y|)\\
&= \qquad \{\text{ def. of } \mathsf{map}, gemm\ \}\\
&\quad |\,\mathsf{reduce}(\oplus,_)\ x\ y|\\
&= \qquad \{\text{ below }\}\\
&\quad gemm\ (\oplus,\otimes)\ (tr\ x)\ y\\
&= \qquad \{\text{ def. of } \mathsf{reduce}\ \}\\
&\quad gemm\ (\oplus,\otimes)\ (tr\ (\mathsf{reduce}\ (\phi,_)\ |x|))\ (\mathsf{reduce}\ (_,\phi)\ |y|)
\end{aligned}
$$

$$
\begin{array}{ll}
& \mathsf{map}\ (\mathsf{reduce}(\oplus, _))\ (gemm(_, \mathsf{zipwith}(\otimes))\ (x_1 \ominus x_2)\ (y_1 \phi\ y_2)) \\
= & \qquad \{ \text{ def. of } \mathsf{map}, gemm \ \} \\
& (\mathsf{map}\ (\mathsf{reduce}(\oplus, _))(gemm(_, \mathsf{zipwith}(\otimes))\ x_1\ y_1) \\
& \quad \phi\ \mathsf{map}\ (\mathsf{reduce}(\oplus, _))\ (gemm(_, \mathsf{zipwith}(\otimes))\ x_1\ y_2)) \\
& \qquad \ominus(\mathsf{map}\ (\mathsf{reduce}(\oplus, _))\ (gemm(_, \mathsf{zipwith}(\otimes))\ x_2\ y_1) \\
& \qquad\quad \phi\ \mathsf{map}\ (\mathsf{reduce}(\oplus, _))\ (gemm(_, \mathsf{zipwith}(\otimes))\ x_2\ y_2)) \\
= & \qquad \{ \text{ hypo. of induction } \} \\
& (gemm\ (\oplus, \otimes)\ (tr\ (\mathsf{reduce}\ (\phi, _)\ x_1))\ (\mathsf{reduce}\ (_, \phi)\ y_1) \\
& \quad \phi\, gemm\ (\oplus, \otimes)\ (tr\ (\mathsf{reduce}\ (\phi, _)\ x_1))\ (\mathsf{reduce}\ (_, \phi)\ y_2)) \\
& \qquad \ominus(gemm\ (\oplus, \otimes)\ (tr\ (\mathsf{reduce}\ (\phi, _)\ x_2))\ (\mathsf{reduce}\ (_, \phi)\ y_1) \\
& \qquad\quad \phi\, gemm\ (\oplus, \otimes)\ (tr\ (\mathsf{reduce}\ (\phi, _)\ x_2))\ (\mathsf{reduce}\ (_, \phi)\ y_2)) \\
= & \qquad \{ \text{ def. of } gemm \ \} \\
& gemm\ (\oplus, \otimes)\ (tr\ (\mathsf{reduce}\ (\phi, _)\ x_1)\ \phi\ tr\ (\mathsf{reduce}\ (\phi, _)\ x_2))\ (\mathsf{reduce}\ (_, \phi)\ y_1\ \phi\ \mathsf{reduce}\ (_, \phi)\ y_2)) \\
= & \qquad \{ \text{ def. of } tr, \mathsf{reduce} \ \} \\
& gemm\ (\oplus, \otimes)\ (tr\ (\mathsf{reduce}\ (\phi, _)\ (x_1 \ominus x_2)))\ (\mathsf{reduce}\ (_, \phi)\ (y_1 \phi\ y_2))
\end{array}
$$

To complete the proof, we prove the following equation by the induction on the structure of abide trees.

$$
\begin{array}{l}
|\, \mathsf{reduce}(\oplus, _)\ (\mathsf{zipwith}(\otimes)\ x\ y)| = gemm(\oplus, \otimes)\ (tr\ x)\ y \\
\Leftarrow width\ x = 1, width\ y = 1
\end{array}
$$

Proof:

$$
\begin{array}{ll}
& |\, \mathsf{reduce}(\oplus, _)\ (\mathsf{zipwith}(\otimes)\ |x|\ |y|)| \\
= & \qquad \{ \text{ def. of } \mathsf{zipwith}, \mathsf{reduce} \ \} \\
& |x \otimes y| \\
= & \qquad \{ \text{ def. of } gemm, tr \ \} \\
& gemm(\oplus, \otimes)\ (tr\ |x|)\ |y|
\end{array}
$$

$$
\begin{array}{ll}
& |\, \mathsf{reduce}(\oplus, _)\ (\mathsf{zipwith}(\otimes)\ (x_1 \ominus x_2)\ (y_1 \ominus y_2))| \\
= & \qquad \{ \text{ def. of } \mathsf{zipwith}, \mathsf{reduce} \ \} \\
& |\, \mathsf{reduce}(\oplus, _)\ (\mathsf{zipwith}(\otimes)\ x_1\ y_1) \oplus \mathsf{reduce}(\oplus, _)\ (\mathsf{zipwith}(\otimes)\ x_2\ y_2)| \\
= & \qquad \{ \text{ def. of } \mathsf{zipwith} \ \} \\
& \mathsf{zipwith}(\oplus)\ |\, \mathsf{reduce}(\oplus, _)\ (\mathsf{zipwith}(\otimes)\ x_1\ y_1)|\ |\, \mathsf{reduce}(\oplus, _)\ (\mathsf{zipwith}(\otimes)\ x_2\ y_2)| \\
= & \qquad \{ \text{ hypo. of induction } \} \\
& \mathsf{zipwith}(\oplus)\ (gemm(\oplus, \otimes)\ (tr\ x_1)\ y_1)\ (gemm(\oplus, \otimes)\ (tr\ x_2)\ y_2) \\
= & \qquad \{ \text{ def. of } gemm, tr \ \} \\
& gemm(\oplus, \otimes)\ (tr\ (x_1 \ominus x_2))\ (y_1 \ominus y_2)
\end{array}
$$

Rule XVI

$$
\begin{array}{l}
\mathsf{map}\ (\mathsf{reduce}(_, \oplus))\ (gemm(_, \mathsf{zipwith}(\otimes))\ x\ y) \\
= gemm(\oplus, \otimes)\ (\mathsf{reduce}\ (\ominus, _)\ x)\ (tr(\mathsf{reduce}\ (_, \ominus)\ y)) \\
\Leftarrow width\ x = 1, height \text{ of } x\text{'s elements} = 1, height \text{ of } y \text{ and its elements} = 1
\end{array}
$$

Proof: Similar to Rule XV.

Rule XVII

$$\begin{array}{l} \mathsf{map}(\mathsf{reduce}(\uparrow,_))\ (\mathsf{zipwith}_3\ \ f_r\ r_1\ r_2\ ro_2) \\ \quad = \mathsf{zipwith}_3\ \ f'_r\ (\mathsf{reduce}\ (\uparrow)\ r_1)\ ro_2\ (\mathsf{reduce}\ (\uparrow,_)r_2) \\ \quad \textbf{where}\ \ f_r\ r_1\ r_2\ ro_2 = \mathsf{map}\ (+ro_2)\ r_1 \ \phi\ r_2 \\ \qquad\quad f'_r\ r_1\ ro_2\ r_2 = (r_1 + ro_2) \uparrow\ r_2 \end{array}$$

Proof: Rule VII and following calculation.

$$\begin{array}{ll} & \mathsf{reduce}(\uparrow,_)\ (f_r\ r_1\ r_2\ ro_2) \\ = & \qquad \{\ \text{def. of } f_r\ \} \\ & \mathsf{reduce}(\uparrow,_)\ ((\mathsf{map}\ (+ro_2)r_1)\ \phi\ r_2) \\ = & \qquad \{\ \text{def. of } \mathsf{reduce}\ \} \\ & \mathsf{reduce}(\uparrow,_)\ (\mathsf{map}\ (+ro_2)r_1) \uparrow r_2 \\ = & \qquad \{\ + \text{ distributes over } \uparrow\ \} \\ & ((\mathsf{reduce}(\uparrow,_)\ r_1) + ro_2) \uparrow r_2 \end{array}$$

Rule XVIII

$$\begin{array}{l} \mathsf{reduce}(_,\phi)\ (\mathsf{map}\ (\mathsf{zipwith}(+)\ (right'\ x))\ y) \\ \ = \mathsf{map}_c\ (\mathsf{zipwith}(+)\ (right\ (\mathsf{reduce}(_,\phi)\ x)))\ (\mathsf{reduce}(_,\phi)\ y) \\ \Leftarrow height\ x = 1, width \text{ of } x\text{'s elements} = 1 \end{array}$$

Proof: First, we prove the next equation by the induction on the structure of abide trees.

$$\begin{array}{l} \mathsf{reduce}(_,\phi)\ (\mathsf{map}\ \ f\ x) = \mathsf{map}_c\ \ f\ (\mathsf{reduce}(_,\phi)\ x) \\ \Leftarrow height\ x = 1, width \text{ of } x\text{'s elements} = 1 \end{array}$$

Proof:

$$\begin{array}{ll} & \mathsf{reduce}(_,\phi)\ (\mathsf{map}\ \ f\ |x|) \\ = & \qquad \{\ \text{def. of } \mathsf{reduce}, \mathsf{map}\ \} \\ & f\ x \\ = & \qquad \{\ \text{def. of } \mathsf{map}_c,\ height\ x = 1\ \} \\ & \mathsf{map}_c\ \ f\ x \\ = & \qquad \{\ \text{def. of } \mathsf{reduce}\ \} \\ & \mathsf{map}_c\ \ f\ (\mathsf{reduce}(_,\phi)\ |x|) \end{array}$$

$$\begin{array}{ll} & \mathsf{reduce}(_,\phi)\ (\mathsf{map}\ \ f\ (x_1\ \phi\ x_2)) \\ = & \qquad \{\ \text{def. of } \mathsf{reduce}, \mathsf{map}\ \} \\ & \mathsf{map}\ \ f\ x_1\ \phi\ \mathsf{map}\ \ f\ x_2 \\ = & \qquad \{\ \text{hypo. of induction}\ \} \\ & \mathsf{map}_c\ \ f\ (\mathsf{reduce}(_,\phi)\ x_1)\ \phi\ \mathsf{map}_c\ \ f\ (\mathsf{reduce}(_,\phi)\ x_2) \\ = & \qquad \{\ \text{def. of } \mathsf{map}_c\ \} \\ & \mathsf{map}_c\ \ f\ (\mathsf{reduce}(_,\phi)\ (x_1\ \phi\ x_2)) \end{array}$$

To complete the proof, we prove the next equation by the induction on the structure of abide trees.

$$\begin{array}{l} right'\ x = right\ (\mathsf{reduce}(_,\phi)\ x) \\ \Leftarrow height\ x = 1, width \text{ of } x\text{'s elements} = 1 \end{array}$$

Proof:

$$
\begin{array}{ll}
 & \mathit{right}' \ |x| \\
= & \quad \{ \text{ def. of } \mathit{right}' \ \} \\
 & x \\
= & \quad \{ \text{ def. of } \mathit{right}, \ \mathit{width} \ x = 1 \ \} \\
 & \mathit{right} \ x \\
= & \quad \{ \text{ def. of } \mathsf{reduce} \ \} \\
 & \mathit{right} \ (\mathsf{reduce}(_, \phi) \ |x|)
\end{array}
$$

$$
\begin{array}{ll}
 & \mathit{right}' \ (x_1 \phi x_2) \\
= & \quad \{ \text{ def. of } \mathit{right}' \ \} \\
 & \mathit{right}' \ x_2 \\
= & \quad \{ \text{ hypo. of induction } \} \\
 & \mathit{right} \ (\mathsf{reduce}(_, \phi) \ x_2) \\
= & \quad \{ \text{ def. of } \mathit{right} \ \} \\
 & \mathit{right} \ (\mathsf{reduce}(_, \phi) \ x_1 \phi \ \mathsf{reduce}(_, \phi) \ x_2) \\
= & \quad \{ \text{ def. of } \mathsf{reduce} \ \} \\
 & \mathit{right} \ (\mathsf{reduce}(_, \phi) \ (x_1 \phi x_2))
\end{array}
$$

Rule XIX

$$
\begin{array}{l}
\mathsf{reduce}(\phi, _) \ (\mathsf{map} \ (\mathsf{zipwith}(+) \ (\mathit{top}' \ x)) \ y) \\
\ = \mathsf{map}_c \ (\mathsf{zipwith}(+) \ (\mathit{top} \ (\mathsf{reduce}(\phi, _) \ x))) \ (\mathsf{reduce}(\phi, _) \ y) \\
\Leftarrow \mathit{width} \ x = 1, \mathit{width} \text{ of } x\text{'s elements} = 1
\end{array}
$$

Proof: Similar to Rule XVIII.

Rule XX

$$
\begin{array}{l}
\mathsf{reduce}(_, \phi) \ (\mathsf{zipwith} \ \ f \ x \ y) \\
\ = \mathsf{map}_r \ (\mathsf{zipwith}(+) \ (\mathit{top} \ (\mathsf{reduce}(_, \phi) \ y))) \ (\mathsf{reduce}(\phi, _) \ x) \ominus (\mathsf{reduce}(\phi, _) \ y) \\
\Leftarrow \mathit{height} \ x = 1, \mathit{height} \ y = 1, \mathit{width} \text{ of } x \text{ and } y\text{'s elements} = 1 \\
\quad f \ x \ y = \mathsf{map} \ (+(\mathit{top}' \ y)) \ x \ominus y
\end{array}
$$

Proof: Rule XIV with $f' \ a \ b = \mathsf{map}_r \ (\mathsf{zipwith}(+) \ (\mathit{top} \ b)) \ a \ominus \ b$ and $\otimes = \phi$, $\otimes_1 = \phi$, $\otimes_2 = \phi$.

Rule XXI

$$
\begin{array}{l}
\mathsf{reduce}(_, \phi) \ (\mathsf{zipwith} \ \ f \ x \ y) \\
\ = \mathsf{map}_r \ (\mathsf{zipwith}(+) \ (\mathit{top} \ (\mathsf{reduce}(_, \phi) \ y))) \ (\mathsf{reduce}(\phi, _) \ x) \ominus (\mathsf{reduce}(\phi, _) \ y) \\
\Leftarrow \mathit{width} \ x = 1, \mathit{width} \ y = 1, \mathit{width} \text{ of } x \text{ and } y\text{'s elements} = 1 \\
\quad f \ x \ y = \mathsf{map} \ (+(\mathit{top}' \ y)) \ x \ominus y
\end{array}
$$

Proof: Rule XIV with $f' \ a \ b = \mathsf{map}_r \ (\mathsf{zipwith}(+) \ (\mathit{top} \ b)) \ a \ominus \ b$ and $\oplus = \phi$, $\oplus_1 = \phi$, $\oplus_2 = \phi$.

Appendix B

Complete Proof of Theorem 5.1

In the following, we use an abbreviation of the index accessing for readability: $\langle x \rangle_i = \mathsf{at}\ x\ i$. We assume that the length of an input list is n. The goal of this proof is to show the equation $eval_P\ prog = eval\ (compile\ prog)$ for any $prog$.

Base Case

What we have to prove is the following equation for a list x.

$$eval_P\ x \quad = \quad eval\ [\![\ [], Leaf_c\ id\ (Var\ x\ 0), []\ ]\!] \tag{B.1}$$

The ith element of the left-hand sides is as follows.

$$\begin{aligned} \langle eval_P\ x \rangle_i \quad = & \quad \{\ \text{definition of } eval_P\ \} \\ & \langle eval_P\ x \rangle_i \end{aligned}$$

The ith element of the right-hand sides is as follows.

$$\begin{aligned} \langle eval\ [\![\ [], Leaf_c\ id\ (Var\ x\ 0), []\ ]\!] \rangle_i \quad = & \quad \{\ \text{definition of } eval\ \} \\ & \langle \mathsf{map}\ \ eval_T\ (Leaf_c\ id\ (Var\ x\ 0))\ [0..n-1] \rangle_i \\ = & \quad \{\ i\text{th element}\ \} \\ & eval_T\ (Leaf_c\ id\ (Var\ x\ 0))\ i \\ = & \quad \{\ \text{definition of } eval_T \text{ and the identity function}\ \} \\ & eval_V\ (Var\ x\ 0)\ i \\ = & \quad \{\ \text{definition of } eval_V \text{ and } i\text{th element}\ \} \\ & \langle x \rangle_i \end{aligned}$$

Thus, the following equation holds.

$$\langle eval_P\ x \rangle_i \quad = \quad \langle eval\ [\![\ [], Leaf_c\ id\ (Var\ x\ 0), []\ ]\!] \rangle_i$$

Since this equation holds for $i \in [0..n-1]$, the equation (B.1) holds.

Inductive Case for $\underline{\mathsf{map}}$

What we have to prove is the following equation for a function f and a program $prog$. Here, $compile\ prog = [\![\ ls, zms, rs\ ]\!]$.

$$eval_P\ (\underline{\mathsf{map}}\ f\ prog) \quad = \quad eval\ [\![\ \mathsf{map}\ \ (comp\ f)\ ls, comp\ f\ zms, \mathsf{map}\ \ (comp\ f)\ rs\ ]\!] \tag{B.2}$$

To prove this equation, we first show a lemma.

Lemma B.3. Let f be a function, t be a computational tree, and i be an index. Then, the following equation holds.

$$eval_T\ (comp\ f\ t)\ i \quad = \quad f\ (eval_T\ t\ i)$$

Proof. This is shown by induction on *Tree*.

The *Node* case:

$$\begin{aligned}
eval_T\ (comp\ f\ (Node\ g\ l\ r))\ i &= \quad \{\ \text{definition of } comp\ \} \\
&\quad eval_T\ (Node\ (f \circ g)\ l\ r)\ i \\
&= \quad \{\ \text{definition of } eval_T\ \} \\
&\quad (f \circ g)\ (eval_T\ l\ i, eval_T\ r\ i) \\
&= \quad \{\ \text{function composition}\ \} \\
&\quad f\ (g\ (eval_T\ l\ i, eval_T\ r\ i)) \\
&= \quad \{\ \text{definition of } eval_T\ \} \\
&\quad f\ (eval_T\ (Node\ g\ l\ r)\ i)
\end{aligned}$$

The $Leaf_v$ case:

$$\begin{aligned}
eval_T\ (comp\ f\ (Leaf_v\ g\ v))\ i &= \quad \{\ \text{definition of } comp\ \} \\
&\quad eval_T\ (Leaf_v\ (f \circ g)\ v)\ i \\
&= \quad \{\ \text{definition of } eval_T\ \} \\
&\quad (f \circ g)\ (eval_V\ v\ i) \\
&= \quad \{\ \text{function composition}\ \} \\
&\quad f\ (g\ (eval_V\ v\ i)) \\
&= \quad \{\ \text{definition of } eval_T\ \} \\
&\quad f\ (eval_T\ (Leaf_v\ g\ v)\ i)
\end{aligned}$$

The $Leaf_c$ case:

$$\begin{aligned}
eval_T\ (comp\ f\ (Leaf_c\ c))\ i &= \quad \{\ \text{definition of } comp\ \} \\
&\quad eval_T\ (Leaf_c\ (f\ c))\ i \\
&= \quad \{\ \text{definition of } eval_T\ \} \\
&\quad f\ c \\
&= \quad \{\ \text{definition of } eval_T\ \} \\
&\quad f\ (eval_T\ (Leaf_c\ c)\ i)
\end{aligned}$$

□ □

Now, we show the equation (B.2). In the following, $l = length\ ls$, $r = length\ rs$, and $idces = [l..(n-r-1)]$.

$$\begin{aligned}
&\quad eval_P\ (\underline{\mathsf{map}}\ f\ prog) \\
= &\quad \{\ \text{definition of } eval_P\ \} \\
&\quad \mathsf{map}\ f\ (eval_P\ prog) \\
= &\quad \{\ \text{induction hypothesis}\ \} \\
&\quad \mathsf{map}\ f\ (eval\ [\![\, ls, zms, rs\,]\!]) \\
= &\quad \{\ \text{definition of } eval\ \} \\
&\quad \mathsf{map}\ f\ (\mathsf{map}\ eval_{T0}\ ls +\!\!+ \mathsf{map}\ (eval_T\ zms)\ idces +\!\!+ \mathsf{map}\ eval_{T0}\ rs) \\
= &\quad \{\ \text{definition of } \mathsf{map} \text{ and its distributivity: } \mathsf{map}\ h \circ \mathsf{map}\ g = \mathsf{map}\ (h \circ g)\ \} \\
&\quad \mathsf{map}\ (f \circ eval_{T0})\ ls +\!\!+ \mathsf{map}\ (f \circ eval_T\ zms)\ idces +\!\!+ \mathsf{map}\ (f \circ eval_{T0})\ rs \\
= &\quad \{\ \text{Lemma B.3 and definition of } eval_{T0}\ \} \\
&\quad \mathsf{map}\ (eval_{T0} \circ comp\ f)\ ls +\!\!+ \mathsf{map}\ (eval_T (comp\ f\ zms))\ idces +\!\!+ \mathsf{map}\ (eval_{T0} \circ comp\ f)\ rs \\
= &\quad \{\ \text{distributivity of } \mathsf{map} \text{ and definition of } eval\ \} \\
&\quad eval\ [\![\, \mathsf{map}\ (comp\ f)\ ls, comp\ f\ zms, \mathsf{map}\ (comp\ f)\ rs\,]\!]
\end{aligned}$$

Thus, the equation (B.2) holds.

Inductive Cases for $\underline{\mathsf{shift}}_{\ll}$ and $\underline{\mathsf{shift}}_{\gg}$

What we have to prove is the following equations for an element e and a program $prog$. Here, $compile\ prog = [\![\, ls, zms, rs \,]\!]$.

$$eval_P\ (\underline{\mathsf{shift}}_{\ll}\ e\ prog) = eval\ [\![\, \mathsf{tail}\ ls, slide\ (-1)\ zms, rs \mathbin{+\!\!+} [Leaf_c\ e] \,]\!] \tag{B.4}$$
$$eval_P\ (\underline{\mathsf{shift}}_{\gg}\ e\ prog) = eval\ [\![\, [Leaf_c\ e] \mathbin{+\!\!+} ls, slide\ 1\ zms, \mathsf{init}\ rs \,]\!] \tag{B.5}$$

To prove this equation, we first show a lemma.

Lemma B.6. Let d be an integer, i be an index, and t be a computational tree. Then, the following equation holds.

$$eval_T\ (slide\ d\ t)\ i \quad = \quad eval_T\ t\ (i-d) \tag{B.7}$$

Proof. This is shown by induction on $Tree$ and Var.

The $Node$ case:

$$\begin{aligned}
eval_T\ (slide\ d\ (Node\ f\ l\ r))\ i \ &= \quad \{\ \text{definition of } slide\ \} \\
&\quad eval_T\ (Node\ f\ (slide\ d\ l)\ (slide\ d\ r)))\ i \\
&= \quad \{\ \text{definition of } eval_T\ \} \\
&\quad f\ (eval_T\ (slide\ d\ l)\ i, eval_T\ (slide\ d\ r)\ i) \\
&= \quad \{\ \text{induction hypothesis}\ \} \\
&\quad f\ (eval_T\ l\ (i-d), eval_T\ r\ (i-d)) \\
&= \quad \{\ \text{definition of } eval_T\ \} \\
&\quad eval_T\ (Node\ f\ l\ r)\ (i-d)
\end{aligned}$$

The $Leaf_v$ with Var case:

$$\begin{aligned}
eval_T\ (slide\ d\ (Leaf_v\ f\ (Var\ x\ s)))\ i \ &= \quad \{\ \text{definition of } slide\ \} \\
&\quad eval_T\ (Leaf_v\ f\ (Var\ x\ (s+d)))\ i \\
&= \quad \{\ \text{definition of } eval_T \text{ and } eval_V\ \} \\
&\quad f\ (\mathsf{at}\ x\ (i-(s+d))) \\
&= \quad \{\ \text{arithmetic}\ \} \\
&\quad f\ (\mathsf{at}\ x\ ((i-d)-s)) \\
&= \quad \{\ \text{definition of } eval_T \text{ and } eval_V\ \} \\
&\quad eval_T\ (Leaf_v\ f\ (Var\ x\ s))\ (i-d)
\end{aligned}$$

Since other cases of $Leaf_v$ and the case of $Leaf_c$ ignore the index i, the equation (B.7) holds in these cases.

Thus, the equation (B.7) holds. □ □

Now, we show the equation (B.4). In the following, $l = length\ ls$ and $r = length\ rs$. First, we

assume ls is not empty, i.e. $l > 0$.

$$
\begin{aligned}
& \mathit{eval}_P\ (\underline{\mathsf{shift}}_{\ll}\ e\ \mathit{prog}) \\
= & \quad \{\ \text{definition of } \mathit{eval}_P\ \} \\
& \mathsf{shift}_{\ll}\ e\ (\mathit{eval}_P\ \mathit{prog}) \\
= & \quad \{\ \text{induction hypothesis}\ \} \\
& \mathsf{shift}_{\ll}\ e\ (\mathit{eval}\ [\![\, \mathit{ls}, \mathit{zms}, \mathit{rs} \,]\!]) \\
= & \quad \{\ \text{definition of } \mathit{eval}\ \} \\
& \mathsf{shift}_{\ll}\ e\ (\mathsf{map}\ \ \mathit{eval}_{T0}\ \mathit{ls} +\!\!+ \mathsf{map}\ \ (\mathit{eval}_T\ \mathit{zms})\ [l..(n-r-1)] +\!\!+ \mathsf{map}\ \ \mathit{eval}_{T0}\ \mathit{rs}) \\
= & \quad \{\ \text{the relation } \mathsf{shift}_{\ll}\ e\ x = \mathsf{tail}\ x +\!\!+ [e] \text{, and } \mathit{ls} \text{ being not empty}\ \} \\
& \mathsf{tail}\ (\mathsf{map}\ \ \mathit{eval}_{T0}\ \mathit{ls}) +\!\!+ \mathsf{map}\ \ (\mathit{eval}_T\ \mathit{zms})\ [l..(n-r-1)] +\!\!+ \mathsf{map}\ \ \mathit{eval}_{T0}\ \mathit{rs} +\!\!+ [e] \\
= & \quad \{\ \text{definition of } \mathsf{map} \text{ and } \mathit{eval}_{T0}\ \} \\
& \mathsf{map}\ \ \mathit{eval}_{T0}\ (\mathsf{tail}\ \mathit{ls}) +\!\!+ \mathsf{map}\ \ (\mathit{eval}_T\ \mathit{zms})\ (\mathsf{map}\ \ (+1)\ [(l-1)..(n-(r+1)-1)]) \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad +\!\!+ \mathsf{map}\ \ \mathit{eval}_{T0}\ (\mathit{rs} +\!\!+ [\mathit{Leaf}_c\ e]) \\
= & \quad \{\ \text{distributivity of } \mathsf{map} \text{, and Lemma B.6}\ \} \\
& \mathsf{map}\ \ \mathit{eval}_{T0}\ (\mathsf{tail}\ \mathit{ls}) +\!\!+ \mathsf{map}\ \ (\mathit{eval}_T\ (\mathit{slide}\ (-1)\ \mathit{zms}))\ [l'..(n-r'-1)] \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad +\!\!+ \mathsf{map}\ \ \mathit{eval}_{T0}\ (\mathit{rs} +\!\!+ [\mathit{Leaf}_c\ e]) \\
& \qquad \mathbf{where}\ l' = \mathit{length}\ \mathsf{tail}\ \mathit{ls},\ \ r' = \mathit{length}\ (\mathit{rs} +\!\!+ [\mathit{Leaf}_c\ e]) \\
= & \quad \{\ \text{definition of } \mathit{eval}\ \} \\
& \mathit{eval}\ [\![\, \mathsf{tail}\ \mathit{ls}, \mathit{slide}\ (-1)\ \mathit{zms}, \mathit{rs} +\!\!+ [\mathit{Leaf}_c\ e] \,]\!]
\end{aligned}
$$

Next, we assume $ls = [\,]$.

$$
\begin{aligned}
& \mathit{eval}_P\ (\underline{\mathsf{shift}}_{\ll}\ e\ \mathit{prog}) \\
= & \quad \{\ \text{the same as the previous calculation}\ \} \\
& \mathsf{shift}_{\ll}\ e\ (\mathsf{map}\ \ \mathit{eval}_{T0}\ \mathit{ls} +\!\!+ \mathsf{map}\ \ (\mathit{eval}_T\ \mathit{zms})\ [l..(n-r-1)] +\!\!+ \mathsf{map}\ \ \mathit{eval}_{T0}\ \mathit{rs}) \\
= & \quad \{\ \text{the relation } \mathsf{shift}_{\ll}\ e\ x = \mathsf{tail}\ x +\!\!+ [e] \text{, and } \mathit{ls} \text{ being empty}\ \} \\
& \mathsf{tail}\ (\mathsf{map}\ \ (\mathit{eval}_T\ \mathit{zms})\ [0..(n-r-1)]) +\!\!+ \mathsf{map}\ \ \mathit{eval}_{T0}\ \mathit{rs} +\!\!+ [e] \\
= & \quad \{\ \text{definition of } \mathsf{tail}\ \} \\
& \mathsf{map}\ \ (\mathit{eval}_T\ \mathit{zms})\ ([1..(n-r-1)]) +\!\!+ \mathsf{map}\ \ \mathit{eval}_{T0}\ \mathit{rs} +\!\!+ [e] \\
= & \quad \{\ \text{definition of } \mathsf{map} \text{ and } \mathit{eval}_{T0}\ \} \\
& \mathsf{map}\ \ (\mathit{eval}_T\ \mathit{zms})\ (\mathsf{map}\ \ (+1)\ [0..(n-(r+1)-1)]) +\!\!+ \mathsf{map}\ \ \mathit{eval}_{T0}\ (\mathit{rs} +\!\!+ [\mathit{Leaf}_c\ e]) \\
= & \quad \{\ \text{distributivity of } \mathsf{map} \text{, Lemma B.6, and } \mathsf{tail}\ \mathit{ls} = [\,]\ \} \\
& \mathsf{map}\ \ \mathit{eval}_{T0}\ (\mathsf{tail}\ \mathit{ls}) +\!\!+ \mathsf{map}\ \ (\mathit{eval}_T\ (\mathit{slide}\ (-1)\ \mathit{zms}))\ [l'..(n-r'-1)] \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad +\!\!+ \mathsf{map}\ \ \mathit{eval}_{T0}\ (\mathit{rs} +\!\!+ [\mathit{Leaf}_c\ e]) \\
& \qquad \mathbf{where}\ l' = \mathsf{tail}\ \mathit{ls},\ \ r' = \mathit{length}\ (\mathit{rs} +\!\!+ [\mathit{Leaf}_c\ e]) \\
= & \quad \{\ \text{definition of } \mathit{eval}\ \} \\
& \mathit{eval}\ [\![\, \mathsf{tail}\ \mathit{ls}, \mathit{slide}\ (-1)\ \mathit{zms}, \mathit{rs} +\!\!+ [\mathit{Leaf}_c\ e] \,]\!]
\end{aligned}
$$

Thus, the equation (B.4) holds.

The equation (B.5) is shown similarly.

Inductive Case for $\underline{\mathsf{zip}}$

What we have to prove is the following equation for two programs prog_1 and prog_2. Here, $\mathit{compile}\ \mathit{prog}_1 = [\![\, \mathit{ls}_1, \mathit{zms}_1, \mathit{rs}_1 \,]\!]$ and $\mathit{compile}\ \mathit{prog}_2 = [\![\, \mathit{ls}_2, \mathit{zms}_2, \mathit{rs}_2 \,]\!]$.

$$
\begin{aligned}
& \mathit{eval}_P\ (\underline{\mathsf{zip}}\ \mathit{prog}_1\ \mathit{prog}_2) = \mathit{eval}\ [\![\, \mathit{ls}, \mathit{zms}, \mathit{reverse}\ \mathit{rs} \,]\!] \\
& \quad \mathbf{where}\ \ \mathit{zms} = \mathit{Node}\ \mathit{id}\ \mathit{zms}_1\ \mathit{zms}_2 \\
& \qquad\qquad\ \ \mathit{ls} = \mathit{trim}\ \mathit{FromL}\ \mathit{ls}_1\ \mathit{ls}_2\ \mathit{zms}_1\ \mathit{zms}_2 \\
& \qquad\qquad\ \ \mathit{rs} = \mathit{trim}\ \mathit{FromR}\ (\mathit{reverse}\ \mathit{rs}_1)\ (\mathit{reverse}\ \mathit{rs}_2)\ \mathit{zms}_1\ \mathit{zms}_2
\end{aligned}
\tag{B.8}
$$

To prove this equation, we first show a lemma.

Lemma B.9. Let i be an index, and t be a computational tree. Then, the following equation holds.

$$eval_{T0}\ (inst\ FromL\ i) \quad = \quad eval_T\ t\ i \tag{B.10}$$

Proof. This is shown by induction on *Tree* and *Var*.

The *Node* case:

$$\begin{aligned}
eval_{T0}\ (inst\ FromL\ (Node\ f\ l\ r)\ i) &= \quad \{\ \text{definition of } inst\ \} \\
&\quad\ eval_{T0}\ (Node\ f\ (inst\ FromL\ l\ i)\ (inst\ FromL\ r\ i)) \\
&= \quad \{\ \text{definition of } eval_{T0}\ \} \\
&\quad\ f\ (eval_{T0}\ (inst\ FromL\ l\ i), eval_{T0}\ (inst\ FromL\ r\ i)) \\
&= \quad \{\ \text{induction hypothesis}\ \} \\
&\quad\ f\ (eval_T\ l\ i, eval_T\ r\ i) \\
&= \quad \{\ \text{definition of } eval_T\ \} \\
&\quad\ eval_T\ (Node\ f\ l\ r)\ i
\end{aligned}$$

The $Leaf_v$ with *Var* case:

$$\begin{aligned}
eval_{T0}\ (inst\ FromL\ (Leaf_v\ f\ (Var\ x\ s))\ i) &= \quad \{\ \text{definition of } inst\ \} \\
&\quad\ eval_{T0}\ (Leaf_v\ f\ (Fix\ x\ (-s+i)\ FromL)) \\
&= \quad \{\ \text{definition of } eval_{T0}\ \} \\
&\quad\ f\ (\mathsf{at}\ x\ (-s+i)) \\
&= \quad \{\ \text{definition of } eval_T\ \} \\
&\quad\ eval_T\ (Var\ x\ s)\ i
\end{aligned}$$

Since other cases of $Leaf_v$ and the case of $Leaf_c$ merely return the given tree and this tree ignores the index, the equation (B.10) holds in these cases.

Thus, the equation (B.10) holds. □ □

We also use the following lemma.

Lemma B.11. Let i be an index, and t be a computational tree. Then, the following equation holds.

$$eval_{T0}\ (inst\ FromR\ i) \quad = \quad eval_T\ t\ (n-1-i)$$

Proof. Similar to the proof of Lemma B.9. □ □

In the following, $l_1 = length\ ls_1$, $r_1 = length\ rs_1$, $l_2 = length\ ls_2$ and $r_2 = length\ rs_2$. First,

we assume $l_1 \geq l_2$ and $r_1 \geq r_2$.

$$
\begin{array}{ll}
& eval_P\ (\underline{\mathsf{zip}}\ prog_1\ prog_2) \\
= & \quad \{\ \text{definition of } eval_P\ \} \\
& \mathsf{zip}\ (eval_P\ prog_1, eval_P\ prog_2) \\
= & \quad \{\ \text{induction hypothesis}\ \} \\
& \mathsf{zip}\ (eval\ [\![\, ls_1, zms_1, rs_1 \,]\!])\ (eval\ [\![\, ls_2, zms_2, rs_2 \,]\!]) \\
= & \quad \{\ \text{definition of } eval \text{ and } \mathsf{zip}\text{, letting } ls_1 = ls_{11} +\!\!+ ls_{12} \text{ and } rs_1 = rs_{11} +\!\!+ rs_{12}\ \} \\
& \mathsf{zip}\ (\mathsf{map}\ eval_{T0}\ ls_{11})\ (\mathsf{map}\ eval_{T0}\ ls_2) \\
& +\!\!+ \mathsf{zip}\ (\mathsf{map}\ eval_{T0}\ ls_{11})\ (\mathsf{map}\ (eval_T\ zms_2)\ [l_2..(l_1-1)]) \\
& +\!\!+ \mathsf{zip}\ (\mathsf{map}\ (eval_T\ zms_1)\ [l_1..(n-r_1-1)])\ (\mathsf{map}\ (eval_T\ zms_2)\ [l_1..(n-r_1-1)]) \\
& +\!\!+ \mathsf{zip}\ (\mathsf{map}\ eval_{T0}\ rs_{11})\ (\mathsf{map}\ (eval_T\ zms_2)\ [(n-r_1)..(n-r_2-1)]) \\
& +\!\!+ \mathsf{zip}\ (\mathsf{map}\ eval_{T0}\ rs_{12})\ (\mathsf{map}\ eval_{T0}\ rs_2) \\
= & \quad \{\ \text{Lemma B.9 and Lemma B.11}\ \} \\
& \mathsf{zip}\ (\mathsf{map}\ eval_{T0}\ ls_{11})\ (\mathsf{map}\ eval_{T0}\ ls_2) \\
& +\!\!+ \mathsf{zip}\ (\mathsf{map}\ eval_{T0}\ ls_{11})\ (\mathsf{map}\ eval_{T0}\ (\mathsf{map}\ (inst\ FromL\ zms_2)\ [l_2..(l_1-1)]) \\
& +\!\!+ \mathsf{zip}\ (\mathsf{map}\ (eval_T\ zms_1)\ [l_1..(n-r_1-1)])\ (\mathsf{map}\ (eval_T\ zms_2)\ [l_1..(n-r_1-1)]) \\
& +\!\!+ \mathsf{zip}\ (\mathsf{map}\ eval_{T0}\ rs_{11})\ (\mathsf{map}\ eval_{T0}\ (\mathsf{map}\ (inst\ FromR\ zms_2)\ [(r_1-1)..r_2]) \\
& +\!\!+ \mathsf{zip}\ (\mathsf{map}\ eval_{T0}\ rs_{12})\ (\mathsf{map}\ eval_{T0}\ rs_2) \\
= & \quad \{\ \text{introducing } (Node\ id)\text{, and definition of } eval \text{ and } eval_{T0}\ \} \\
& \mathsf{map}\ eval_{T0}\ (\mathsf{zipwith}\ (Node\ id)\ ls_{11}\ ls_2) \\
& +\!\!+ \mathsf{map}\ eval_{T0}\ (\mathsf{zipwith}\ (Node\ id)\ ls_{11}\ (\mathsf{map}\ (inst\ FromL\ zms_2)\ [l_2..(l_1-1)])) \\
& +\!\!+ \mathsf{map}\ (eval_T\ (Node\ id\ zms_1\ zms_2))\ [l_1..(n-r_1-1)] \\
& +\!\!+ \mathsf{map}\ eval_{T0}\ (\mathsf{zipwith}\ (Node\ id)\ rs_{11}\ (\mathsf{map}\ (inst\ FromR\ zms_2)\ [(r_1-1)..r_2])) \\
& +\!\!+ \mathsf{map}\ eval_{T0}\ (\mathsf{zipwith}\ (Node\ id)\ rs_{12}\ rs_2)
\end{array}
$$

$$
\begin{array}{ll}
= & \quad \{\ \text{combining edge elements}\ \} \\
& \mathsf{map}\ eval_{T0}\ (\mathsf{zipwith}\ (Node\ id)\ ls_1\ (ls_2 +\!\!+ \mathsf{map}\ (inst\ FromL\ zms_2)\ [l_2..(l_1-1)])) \\
& +\!\!+ \mathsf{map}\ (eval_T\ (Node\ id\ zms_1\ zms_2))\ [l_1..(n-r_1-1)] \\
& +\!\!+ \mathsf{map}\ eval_{T0}\ (\mathsf{zipwith}\ (Node\ id)\ rs_1\ (\mathsf{map}\ (inst\ FromR\ zms_2)\ [(r_1-1)..r_2] +\!\!+ rs_2)) \\
= & \quad \{\ \text{definition of } eval \text{ and } trim\ \} \\
& eval\ [\![\, ls, zms, reverse\ rs \,]\!] \\
& \quad \textbf{where}\ \ zms = Node\ id\ zms_1\ zms_2 \\
& \qquad\qquad ls = trim\ FromL\ ls_1\ ls_2\ zms_1\ zms_2 \\
& \qquad\qquad rs = trim\ FromR\ (reverse\ rs_1)\ (reverse\ rs_2)\ zms_1\ zms_2
\end{array}
$$

The other cases ($l_1 \geq l_2 \wedge r_1 < r_2$, $l_1 < l_2 \wedge r_1 \geq r_2$ and $l_1 < l_2 \wedge r_1 < r_2$) are similarly shown.

Thus, the equation (B.8) holds.

Appendix C

Auxiliary Rules for the Proof of Theorem 5.44

Rule I

$$\mathsf{shift}_{\ll}\ e = \mathit{flatten} \circ \mathsf{shift}_{\ll}\ [e] \circ \mathsf{map}\ [\cdot]$$

Proof: It is shown by the following calculation. Some where-clauses are omitted for readability.

$$
\begin{array}{ll}
 & \mathsf{shift}_{\ll}\ e \\
= & \{ \text{ definition of } \mathsf{shift}_{\ll} \ \} \\
 & \mathsf{map}\ \pi_1 \circ \mathsf{scanr}'\ (\oplus)\ (_, e, _) \circ \mathsf{map}\ f \\
 & \textbf{where}\ \ f\ a = (a, a, \mathsf{True}) \\
 & \qquad\quad (_, c, \mathsf{True}) \oplus (_, p, _) = (p, c, \mathsf{False}) \\
 & \qquad\quad (p, c, \mathsf{False}) \oplus (_, _, _) = (p, c, \mathsf{False}) \\
= & \{\ \ \mathit{flatten} \circ \mathsf{map}\ [\cdot] = \mathit{id}\ \ \} \\
 & \mathit{flatten} \circ \mathsf{map}\ [\cdot] \circ \mathsf{map}\ \pi_1 \circ \mathsf{scanr}'\ (\oplus)\ (_, e, _) \circ \mathsf{map}\ f \\
= & \{\ \ [\cdot] \circ \pi_1 = \pi_1 \circ ([\cdot] \times [\cdot] \times \mathit{id})\ \ \} \\
 & \mathit{flatten} \circ \mathsf{map}\ \pi_1 \circ \mathsf{map}\ ([\cdot] \times [\cdot] \times \mathit{id}) \circ \mathsf{scanr}'\ (\oplus)\ (_, e, _) \circ \mathsf{map}\ f \\
= & \{\ \ ([\cdot]\times[\cdot]\times \mathit{id})\ x \oplus ([\cdot]\times[\cdot]\times \mathit{id})\ y = ([\cdot]\times[\cdot]\times \mathit{id})\ (x \oplus y), \text{ and fusion of } \mathsf{map} \text{ and } \mathsf{scanr}'\ \} \\
 & \mathit{flatten} \circ \mathsf{map}\ \pi_1 \circ \mathsf{scanr}'\ (\oplus)\ (_, [e], _) \circ \mathsf{map}\ ([\cdot] \times [\cdot] \times \mathit{id}) \circ \mathsf{map}\ f \\
= & \{\ \ ([\cdot] \times [\cdot] \times \mathit{id}) \circ f = f \circ [\cdot]\ \} \\
 & \mathit{flatten} \circ \mathsf{map}\ \pi_1 \circ \mathsf{scanr}'\ (\oplus)\ (_, [e], _) \circ \mathsf{map}\ f \circ \mathsf{map}\ [\cdot]) \\
= & \{ \text{ definition of } \mathsf{shift}_{\ll} \ \} \\
 & \mathit{flatten} \circ \mathsf{shift}_{\ll}\ [e] \circ \mathsf{map}\ [\cdot]
\end{array}
$$

Rule II

$$\mathit{flatten} \circ \mathsf{shift}_{\ll}\ y \circ \mathsf{map}\ [\cdot] = (+\!\!+ y) \circ \mathit{flatten} \circ \mathsf{shift}_{\ll}\ [\,] \circ \mathsf{map}\ [\cdot]$$

Proof: It is shown by the following calculation. Some where-clauses are omitted for readability.

$$
\begin{aligned}
& (+\!\!+ y) \circ \mathit{flatten} \circ \mathsf{shift}_{\ll}\ [\,] \circ \mathsf{map}\ [\cdot] \\
= & \quad \{ \text{ definition of } \mathsf{shift}_{\ll} \ \} \\
& (+\!\!+ y) \circ \mathit{flatten}\mathsf{map}\ \pi_1 \circ \circ \mathsf{map}\ (\oplus(_, [\,], _)) \circ \mathsf{scanr}\ (\oplus) \circ \mathsf{map}\ f \circ \mathsf{map}\ [\cdot] \\
& \mathbf{where}\ \ f\ a = (a, a, \mathsf{True}) \\
& \qquad\quad (_, c, \mathsf{True}) \oplus (_, p, _) = (p, c, \mathsf{False}) \\
& \qquad\quad (p, c, \mathsf{False}) \oplus (_, _, _) = (p, c, \mathsf{False}) \\
= & \quad \{ \ (+\!\!+ y) \circ \mathit{flatten} = \mathit{flatten} \circ (+\!\!+ [y]) \ \} \\
& \mathit{flatten} \circ (+\!\!+ [y]) \circ \mathsf{map}\ \pi_1 \circ \circ \mathsf{map}\ (\oplus(_, [\,], _)) \circ \mathsf{scanr}\ (\oplus) \circ \mathsf{map}\ f \circ \mathsf{map}\ [\cdot] \\
= & \quad \{ \text{ only the last element in the result of } \mathsf{scanr}\ (\oplus) \text{ has } \mathsf{True} \text{ in the third component } \} \\
& \mathit{flatten}\mathsf{map}\ \pi_1 \circ \mathsf{map}\ (\oplus(_, y, _)) \circ \mathsf{scanr}\ (\oplus) \circ \mathsf{map}\ f \circ \mathsf{map}\ [\cdot] \\
= & \quad \{ \text{ definition of } \mathsf{shift}_{\ll} \ \} \\
& \mathit{flatten} \circ \mathsf{shift}_{\ll}\ y \circ \mathsf{map}\ [\cdot]
\end{aligned}
$$

Rule III

$$\mathsf{shift}_{\ll}\ e\ (x +\!\!+ y) = \mathsf{shift}_{\ll}\ (\mathit{head}\ y)\ x +\!\!+ \mathsf{shift}_{\ll}\ e\ y$$

Proof: It is shown by the following calculation. Some where-clauses are omitted for readability.

$$
\begin{aligned}
& \mathsf{shift}_{\ll}\ e\ (x +\!\!+ y) \\
= & \quad \{ \text{ definition of } \mathsf{shift}_{\ll} \ \} \\
& (\mathsf{map}\ \pi_1 \circ \mathsf{map}\ (\oplus(_, e, _)) \circ \mathsf{scanr}\ (\oplus) \circ \mathsf{map}\ f)\ (x +\!\!+ y) \\
& \mathbf{where}\ \ f\ a = (a, a, \mathsf{True}) \\
& \qquad\quad (_, c, \mathsf{True}) \oplus (_, p, _) = (p, c, \mathsf{False}) \\
& \qquad\quad (p, c, \mathsf{False}) \oplus (_, _, _) = (p, c, \mathsf{False}) \\
= & \quad \{ \text{ definition of } \mathsf{map} \text{ and } \mathsf{scanr} \ \} \\
& (\mathsf{map}\, \pi_1 \circ \mathsf{map}\, (\oplus(_, e, _)) \circ \mathsf{map}\, (\oplus(\mathsf{reduce}\, (\ll)\, (\mathsf{scanr}\, (\oplus)\, (\mathsf{map}\, f\, y))) \circ \mathsf{scanr}\, (\oplus) \circ \mathsf{map}\, f)\, x \\
& \quad +\!\!+ (\mathsf{map}\ \pi_1 \circ \mathsf{map}\ (\oplus(_, e, _)) \circ \mathsf{scanr}\ (\oplus) \circ \mathsf{map}\ f)\ y \\
= & \quad \{ \text{ definition of } \oplus \ \} \\
& (\mathsf{map}\ \pi_1 \circ \mathsf{map}\ (\oplus(_, \mathsf{reduce}\ (\ll)\ y, _)) \circ \mathsf{scanr}\ (\oplus) \circ \mathsf{map}\ f)\ x \\
& \quad +\!\!+ (\mathsf{map}\ \pi_1 \circ \mathsf{map}\ (\oplus(_, e, _)) \circ \mathsf{scanr}\ (\oplus) \circ \mathsf{map}\ f)\ y \\
= & \quad \{ \text{ definition of } \mathsf{shift}_{\ll} \text{ and } \mathit{head} \ \} \\
& \mathsf{shift}_{\ll}\ (\mathit{head}\ y)\ x +\!\!+ \mathsf{shift}_{\ll}\ e\ y
\end{aligned}
$$

Rule IV

$$\mathsf{tail}\ (x +\!\!+ y) = \mathsf{tail}\ x +\!\!+ y$$

Proof: It is shown by the following induction on y. Some where-clauses are omitted for readability.

The base case is shown as follows.

$$
\begin{aligned}
& \mathsf{tail}\ (x \mathbin{+\!\!+} [a]) \\
= & \quad \{ \text{ definition of } \mathsf{tail} \text{ and } \mathsf{shift}_{\ll} \} \\
& (\mathit{flatten} \circ \mathsf{map}\ \pi_1 \circ \mathsf{map}\ (\oplus(_, [\,], _)) \circ \mathsf{scanr}\ (\oplus) \circ \mathsf{map}\ f \circ \mathsf{map}\ [\cdot])\ (x \mathbin{+\!\!+} [a]) \\
& \textbf{where}\ \ f\ a = (a, a, \mathsf{True}) \\
& \qquad\quad (_, c, \mathsf{True}) \oplus (_, p, _) = (p, c, \mathsf{False}) \\
& \qquad\quad (p, c, \mathsf{False}) \oplus (_, _, _) = (p, c, \mathsf{False}) \\
= & \quad \{ \text{ definition of } \mathsf{map} \} \\
& (\mathit{flatten} \circ \mathsf{map}\ \pi_1 \circ \mathsf{map}\ (\oplus(_, [\,], _)))\ (\mathsf{scanr}\ (\oplus)\ (\mathsf{map}\ f\ (\mathsf{map}\ [\cdot]\ x)) \mathbin{+\!\!+} [([a], [a], \mathsf{True})]) \\
= & \quad \{ \text{ definition of } \mathsf{scan} \} \\
& (\mathit{flatten} \circ \mathsf{map}\,\pi_1 \circ \mathsf{map}\,(\oplus(_, [\,], _)))\,((\mathsf{map}\,(\oplus([a], [a], \mathsf{True}))\,(\mathsf{scanr}\,(\oplus)\,(\mathsf{map}\,f\,(\mathsf{map}\,[\cdot]\,x)))) \\
& \quad \mathbin{+\!\!+} [([a], [a], \mathsf{True})]) \\
= & \quad \{ ([a], [a], \mathsf{True}) \oplus (_, [\,], _) = ([\,], [a], \mathsf{False}) \} \\
& (\mathit{flatten} \circ \mathsf{map}\ \pi_1)\ ((\mathsf{map}\ (\oplus(_, [a], _))\ (\mathsf{scanr}\ (\oplus)\ (\mathsf{map}\ f\ (\mathsf{map}\ [\cdot]\ x)))) \mathbin{+\!\!+} [([\,], [a], \mathsf{False})]) \\
= & \quad \{ \text{ definition of } \mathsf{map} \text{ and } \pi_1 \} \\
& (\mathit{flatten} \circ \mathsf{map}\ \pi_1)\ ((\mathsf{map}\ (\oplus(_, [a], _))\ (\mathsf{scanr}\ (\oplus)\ (\mathsf{map}\ f\ (\mathsf{map}\ [\cdot]\ x))))) \\
= & \quad \{ \text{ definition of } \mathsf{shift}_{\ll}\text{, and rule II } \} \\
& \mathsf{tail}\ x \mathbin{+\!\!+} [a]
\end{aligned}
$$

The induction case is shown as follows.

$$
\begin{aligned}
& \mathsf{tail}\ (x \mathbin{+\!\!+} (y \mathbin{+\!\!+} z)) \\
= & \quad \{ \text{ associativity of } \mathbin{+\!\!+}\text{, and induction hypothesis } \} \\
& \mathsf{tail}\ (x \mathbin{+\!\!+} y) \mathbin{+\!\!+} z \\
= & \quad \{ \text{ induction hypothesis, and associativity of } \mathbin{+\!\!+} \} \\
& \mathsf{tail}\ x \mathbin{+\!\!+} (y \mathbin{+\!\!+} z)
\end{aligned}
$$

Rule V

$$[\mathit{head}\ x] \mathbin{+\!\!+} \mathsf{tail}\ x = x$$

Proof: It is shown by the following induction on x.
The base case is shown as follows.

$$
\begin{aligned}
& [\mathit{head}\ [a]] \mathbin{+\!\!+} \mathsf{tail}\ [a] \\
= & \quad \{ \text{ definition of } \mathit{head} \text{ and } \mathsf{tail} \} \\
& [a] \mathbin{+\!\!+} [\,] \\
= & \quad \{ \text{ the empty list } \} \\
& [a]
\end{aligned}
$$

The induction case is shown as follows.

$$
\begin{aligned}
& [\mathit{head}\ (x \mathbin{+\!\!+} y)] \mathbin{+\!\!+} \mathsf{tail}\ (x \mathbin{+\!\!+} y) \\
= & \quad \{ \text{ definition of } \mathit{head}\text{, and rule IV } \} \\
& [\mathit{head}\ x] \mathbin{+\!\!+} \mathsf{tail}\ x \mathbin{+\!\!+} y \\
= & \quad \{ \text{ induction hypothesis } \} \\
& x \mathbin{+\!\!+} y
\end{aligned}
$$

Rule VI

$$\mathsf{shift}_{\ll}\ e\ x = \mathsf{tail}\ x \mathbin{+\!\!+} [e]$$

Proof: It is shown by the following induction on x.

$$
\begin{aligned}
& \mathsf{shift}_{\ll}\ e\ x \\
= \quad & \{\ \text{rule I}\ \} \\
& (\mathit{flatten} \circ \mathsf{shift}_{\ll}\ [e]\ \circ \mathsf{map}\ [\cdot])\ x \\
= \quad & \{\ \text{rule II}\ \} \\
& ((+\!\!+[e]) \circ \mathit{flatten} \circ \mathsf{shift}_{\ll}\ [\,]\ \circ \mathsf{map}\ [\cdot])\ x \\
= \quad & \{\ \text{definition of } \mathsf{tail}\ \} \\
& ((+\!\!+[e]) \circ \mathsf{tail})\ x \\
= \quad & \{\ \text{function composition}\ \} \\
& \mathsf{tail}\ x +\!\!+ [e]
\end{aligned}
$$

Appendix D

Dilation of Rects

We will show the following dilation of $rects'$.

$$
\begin{array}{lll}
\multicolumn{3}{l}{rects' = pack \circ rects} \\
\quad \textbf{where}\ pack & = & flatten \circ unwind_{\ominus} \circ \mathsf{map}\ \ unwind_{\phi} \\
\quad\quad unwind_{\oplus}\ |a| & = & |a| \\
\quad\quad unwind_{\oplus}\ (|a| \phi\ x) & = & |a| \oplus \mathsf{reduce}\ \ (_, \oplus)\ (\mathsf{map}\ \ | \cdot |\ x) \\
\quad\quad unwind_{\oplus}\ ((|a| \phi\ x) \ominus (NIL \phi\ y)) & = & unwind_{\oplus}\ (|a| \phi\ x) \oplus unwind_{\oplus}\ y
\end{array}
$$

We will first show the following dilation of TLs by induction. The result will be used in the proof of the dilation of $rects'$.

$$\mathsf{TLs} = \mathsf{reduce}\ \ (_, \ominus) \circ toplefts$$

The base case is shown below.

$$
\begin{array}{ll}
 & \mathsf{TLs}\ |a| \\
= & \{\ \text{definition of } \mathsf{TLs}\ \} \\
 & ||a|| \\
= & \{\ \text{definition of } \mathsf{reduce}\ \ \} \\
 & \mathsf{reduce}\ \ (_, \ominus)\ |||a||| \\
= & \{\ \text{definition of } toplefts\ \ \} \\
 & \mathsf{reduce}\ \ (_, \ominus)\ (toplefts\ |a|)
\end{array}
$$

The induction case of $\ominus$ is as follows.

$$
\begin{array}{ll}
 & \mathsf{TLs}\ (x \ominus y) \\
= & \{\ \text{definition of } \mathsf{TLs} \text{ and } \mathsf{scan}\ \} \\
 & \mathsf{TLs}\ x \ominus \mathsf{map}_r\ \ (\mathsf{zipwith}\ \ (\ominus)\ (bottom\ (\mathsf{TLs}\ x)))\ (\mathsf{TLs}\ y) \\
= & \left\{\ \text{induction hypothesis, and } bottom \circ \mathsf{TLs} = right' \circ toplefts\ \ \right\} \\
 & \mathsf{reduce}\ \ (_, \ominus)\ (toplefts\ x) \\
 & \ \ominus \mathsf{map}_r\ \ (\mathsf{zipwith}\ \ (\ominus)\ (right'\ (\mathsf{reduce}\ \ (_, \ominus)\ (\ toplefts\ x))))\ (\mathsf{reduce}\ \ (_, \ominus)\ (toplefts\ y)) \\
= & \{\ \text{definition of } \mathsf{reduce}\text{, and a row-wise map is changed to a simple map along}\ \} \\
 & \mathsf{reduce}\ \ (_, \ominus)\ (toplefts\ x) \\
 & \ \ominus \mathsf{map}\ \ (\mathsf{zipwith}\ \ (\ominus)\ (right'\ (\mathsf{reduce}\ \ (_, \ominus)\ (\ toplefts\ x))))\ (\mathsf{reduce}\ \ (_, \ominus)\ (toplefts\ y)) \\
= & \{\ \text{definition of } toplefts\ \} \\
 & \mathsf{reduce}\ \ (_, \ominus)\ (toplefts\ (x \ominus y))
\end{array}
$$

The induction case of ϕ is similarly shown.

Now, we proceed to the dilation of $rects'$. We will show it by induction.

The singleton case is shown below.

$$
\begin{array}{ll}
 & \mathit{rects}'\ |a| \\
= & \{\ \text{definition of } \mathit{rects}'\ \} \\
 & |||a||| \\
= & \{\ \text{definition of } \mathit{pack} \text{ and } \mathit{unwind}\ \} \\
 & \mathit{pack}\ |||a||| \\
= & \{\ \text{definition of } \mathit{rects}\ \} \\
 & \mathit{pack}\ (\mathit{rects}\ |a|)
\end{array}
$$

Next, we will show the induction case of $x \varobar y$. Here, we assume that $\mathit{width}\ x = 1$ for simplicity of the proof. Note that we can impose this assumption safely because the properties of constructors $\ominus$ and $\varobar$ allow restructuring of the input to satisfy the assumption.

The left hand side is calculated as follows.

$$
\begin{array}{ll}
 & \mathit{flatten}\ (\mathsf{map}\ \ \mathsf{TLs}\ (\mathsf{BRs}\ (x \varobar y))) \\
= & \{\ \text{definition of } \mathsf{BRs}\ \} \\
 & \mathit{flatten}\ (\mathsf{map}\ \ \mathsf{TLs}\ (\mathsf{map}_c\ \ (\lambda t.\, \mathsf{zipwith}\ \ (\varobar)\ t\ (\mathit{left}\ (\mathsf{BRs}\ y)))\ (\mathsf{BRs}\ x) \varobar \mathsf{BRs}\ y)) \\
= & \{\ \text{definition of } \mathsf{map} \text{ and } \mathit{flatten}\ \} \\
 & \mathit{flatten}\ (\mathsf{map}\ \ \mathsf{TLs}\ (\mathsf{map}_c\ \ (\lambda t.\, \mathsf{zipwith}\ \ (\varobar)\ t\ (\mathit{left}\ (\mathsf{BRs}\ y)))\ (\mathsf{BRs}\ x))) \\
 & \quad \varobar \mathit{flatten}\ (\mathsf{map}\ \ \mathsf{TLs}\ (\mathsf{BRs}\ y))
\end{array}
$$

The right hand side is as follows. We will omit redundant where-clauses for readability.

$$
\begin{array}{ll}
 & \mathit{pack}\ (\mathit{rects}\ (x \varobar y)) \\
= & \{\ \text{definition of } \mathit{pack} \text{ and } \mathit{rects}\ \} \\
 & (\mathit{flatten} \circ \mathit{unwind}_{\ominus} \circ \mathsf{map}\ \ \mathit{unwind}_{\varobar})\ (\mathsf{zipwith}_4\ \ f_s\ (\mathit{rects}\ x)\ (\mathit{rects}\ y)\ (\mathit{rights}\ x)\ (\mathit{lefts}\ y)) \\
 & \quad \textbf{where}\ \ f_s\ s_1\ s_2\ r_1\ l_2 = (s_1 \varobar \mathit{gemm}\ (_, \varobar)\ r_1\ l_2) \ominus (\mathit{NIL} \varobar s_2) \\
= & \{\ \mathit{rects}\ x \text{ and } \mathit{rights}\ x \text{ are singletons}\ \} \\
 & \mathit{flatten}\ (\mathit{unwind}_{\ominus}\ (\mathsf{zipwith}_4\ \ f'_s\ (\mathit{rects}\ x)\ (\mathit{rects}\ y)\ (\mathit{rights}\ x)\ (\mathit{lefts}\ y)) \\
 & \quad \textbf{where}\ \ f'_s\ s_1\ s_2\ r_1\ l_2 = (s_1 \varobar \mathsf{map}\ \ (\mathsf{the}\ \ r_1 \varobar)\ l_2) \varobar \mathit{unwind}_{\varobar}\ s_2) \\
= & \{\ \text{split of } \mathsf{zipwith}_4\ \} \\
 & \mathit{flatten}\ (\mathit{unwind}_{\ominus}\ (\mathsf{zipwith}\ \ (\varobar)\ \ (\mathsf{zipwith}_3\ \ f''_s\ (\mathit{rects}\ x)\ (\mathit{rights}\ x)\ (\mathit{lefts}\ y))) \\
 & \quad (\mathsf{map}\ \ \mathit{unwind}_{\varobar}\ \ (\mathit{rects}\ y))) \\
 & \quad \textbf{where}\ \ f''_s\ s_1\ r_1\ l_2 = s_1 \varobar \mathsf{map}\ \ (\mathsf{the}\ \ r_1 \varobar)\ l_2) \\
= & \{\ \text{swap of } \mathit{unwind}_{\varobar} \text{ and } \mathsf{zipwith}\ \} \\
 & \mathit{flatten}\ (\mathsf{zipwith}\ \ (\varobar)\ \ (\mathit{unwind}_{\ominus}\ (\mathsf{zipwith}_3\ \ f''_s\ (\mathit{rects}\ x)\ (\mathit{rights}\ x)\ (\mathit{lefts}\ y))) \\
 & \quad (\mathit{unwind}_{\ominus}\ (\mathsf{map}\ \ \mathit{unwind}_{\varobar}\ \ (\mathit{rects}\ y))) \\
= & \{\ \mathit{flatten}\ (\mathsf{zipwith}\ \ (\varobar)\ X\ Y) = \mathit{flatten}\ X \varobar \mathit{flatten}\ Y \text{ when } \mathit{width}\ X = 1\ \} \\
 & \mathit{flatten}\ (\mathit{unwind}_{\ominus}\ (\mathsf{zipwith}_3\ \ f''_s\ (\mathit{rects}\ x)\ (\mathit{rights}\ x)\ (\mathit{lefts}\ y))) \\
 & \quad \varobar \mathit{flatten}\ (\mathit{unwind}_{\ominus}\ (\mathsf{map}\ \ \mathit{unwind}_{\varobar}\ \ (\mathit{rects}\ y)))
\end{array}
$$

Since we can use the induction hypothesis

$$
\mathit{flatten}\ (\mathsf{map}\ \ \mathsf{TLs}\ (\mathsf{BRs}\ y)) = \mathit{flatten}\ (\mathit{unwind}_{\ominus}\ (\mathsf{map}\ \ \mathit{unwind}_{\varobar}\ \ (\mathit{rects}\ y))),
$$

the rest of the proof is to show the following equation.

$$
\begin{array}{l}
\mathit{flatten}\ (\mathsf{map}\ \ \mathsf{TLs}\ (\mathsf{map}_c\ \ (\lambda t.\, \mathsf{zipwith}\ \ (\varobar)\ t\ (\mathit{left}\ (\mathsf{BRs}\ y)))\ (\mathsf{BRs}\ x))) \\
\quad = \mathit{flatten}\ (\mathit{unwind}_{\ominus}\ (\mathsf{zipwith}_3\ \ f''_s\ (\mathit{rects}\ x)\ (\mathit{rights}\ x)\ (\mathit{lefts}\ y))) \\
\qquad \textbf{where}\ \ f''_s\ s_1\ r_1\ l_2 = s_1 \varobar \mathsf{map}\ \ (\mathsf{the}\ \ r_1 \varobar)\ l_2)
\end{array}
$$

We will show this equation by induction on x.

The base case ($x = |a|$) is shown as follows. Note that *height* $y = 1$ for the consistency.

$$
\begin{array}{ll}
 & \mathit{flatten}\ (\mathsf{map}\ \mathsf{TLs}\ (\mathsf{map}_c\ (\lambda t. \mathsf{zipwith}\ (\phi)\ t\ (\mathit{left}\ (\mathsf{BRs}\ y)))\ (\mathsf{BRs}\ |a|))) \\
= & \{ \text{ definition of } \mathsf{BRs} \ \} \\
 & \mathit{flatten}\ (\mathsf{map}\ \mathsf{TLs}\ (\mathsf{map}_c\ (\lambda t. \mathsf{zipwith}\ (\phi)\ t\ (\mathit{left}\ (\mathsf{BRs}\ y)))\ ||a||)) \\
= & \{\ \mathit{left}\ (\mathsf{BRs}\ y) = y \text{ because } \mathit{height}\ y = 1 \ \} \\
 & \mathit{flatten}\ (\mathsf{map}\ \mathsf{TLs}\ (\mathsf{map}_c\ (\lambda t. \mathsf{zipwith}\ (\phi)\ t\ y)\ ||a||)) \\
= & \{ \text{ definition of } \mathsf{map}_c \ \} \\
 & \mathit{flatten}\ (\mathsf{map}\ \mathsf{TLs}\ ||a| \phi\ y|) \\
= & \{ \text{ definition of } \mathsf{map} \text{ and } \mathsf{TLs} \ \} \\
 & \mathit{flatten}\ (|||a|| \phi\ \mathsf{map}_r\ (\mathsf{zipwith}\ (\phi)\ |a|)\ (\mathsf{TLs}\ y)|) \\
= & \{ \text{ change of } \mathsf{map}_r \text{ to } \mathsf{map} \text{ from } \mathit{height}\ (\mathsf{TLs}\ y) = 1, \text{ and definition of } \mathit{flatten} \ \} \\
 & ||a|| \phi\ \mathsf{map}\ (|a|\phi)\ (\mathsf{TLs}\ y) \\
= & \{ \text{ the dilation of } \mathsf{TLs} \ \} \\
 & ||a|| \phi\ \mathsf{map}\ (|a|\phi)\ (\mathsf{reduce}\ (_, \ominus)\ (\mathit{toplefts}\ y)) \\
= & \{\ \mathit{toplefts}\ y = \mathit{lefts}\ y \text{ when } \mathit{height}\ y = 1 \text{ from definition of } \mathit{toplefts} \text{ and } \mathit{lefts} \ \} \\
 & ||a|| \phi\ \mathsf{map}\ (|a|\phi)\ (\mathsf{reduce}\ (_, \ominus)\ (\mathit{toplefts}\ y)) \\
= & \{\ \mathsf{reduce}\ (_, \ominus)\ |a| = a = \mathsf{the}\ a, \text{ and } \mathit{lefts}\ y \text{ is a singleton } \} \\
 & ||a|| \phi\ \mathsf{map}\ (|a|\phi)\ (\mathsf{reduce}\ (_, \ominus)\ (\mathit{lefts}\ y)) \\
= & \{ \text{ definition of } \mathit{flatten} \text{ and } \mathit{unwind} \ \} \\
 & \mathit{flatten}\ (\mathit{unwind}_{\overline{\circ}}\ |||a|| \phi\ \mathsf{map}\ (|a|\phi)\ (\mathsf{reduce}\ (_, \ominus)\ (\mathit{lefts}\ y))|) \\
= & \{ \text{ definition of } \mathsf{zipwith}_3 \text{ and } f''_s \ \} \\
 & \mathit{flatten}\ (\mathit{unwind}_{\overline{\circ}}\ (\mathsf{zipwith}_3\ f''_s\ |||a|||\ |||a|||\ (\mathit{lefts}\ y))) \\
 & \quad \textbf{where}\ f''_s\ s_1\ r_1\ l_2 = s_1 \phi\ \mathsf{map}\ (\mathsf{the}\ r_1 \phi)\ l_2) \\
= & \{ \text{ definition of } \mathit{rects} \text{ and } \mathit{rights} \ \} \\
 & \mathit{flatten}\ (\mathit{unwind}_{\overline{\circ}}\ (\mathsf{zipwith}_3\ f''_s\ (\mathit{rects}\ |a|)\ (\mathit{rights}\ |a|)\ (\mathit{lefts}\ y)))
\end{array}
$$

Then, we will show the induction case ($x = |a| \ominus z$). Note that we should assume that $y = v \ominus w$ and *width* $v = 1$ for the consistency. Also, z and w should have the same width.

The right hand side is calculated as follows.

$$
\begin{array}{ll}
 & \mathit{flatten}\ (\mathsf{map}\ \mathsf{TLs}\ (\mathsf{map}_c\ (\lambda t. \mathsf{zipwith}\ (\phi)\ t\ (\mathit{left}\ (\mathsf{BRs}\ (v \ominus w))))\ (\mathsf{BRs}\ (|a| \ominus z)))) \\
= & \{ \text{ definition of } \mathsf{BRs} \ \} \\
 & \mathit{flatten}\ (\mathsf{map}\ \mathsf{TLs}\ (\mathsf{map}_c\ (\lambda t. \mathsf{zipwith}\ (\phi)\ t\ (\mathit{left} \\
 & \ (\mathsf{map}_r\ (\lambda t. \mathsf{zipwith}\ (\ominus)\ t\ (\mathit{top}\ (\mathsf{BRs}\ w)))\ (\mathsf{BRs}\ v) \ominus (\mathsf{BRs}\ w)))) \\
 & \ \ (\mathsf{map}_r\ (\lambda t. \mathsf{zipwith}\ (\ominus)\ t\ (\mathit{top}\ (\mathsf{BRs}\ z)))\ (\mathsf{BRs}\ |a|) \ominus (\mathsf{BRs}\ z)))) \\
= & \{ \text{ simplification } \} \\
 & \mathit{flatten}\ (\mathsf{map}\ \mathsf{TLs}\ (\mathsf{map}_c\ (\lambda t. \mathsf{zipwith}\ (\phi)\ t\ (\mathit{left} \\
 & \ (\mathsf{map}_r\ (\lambda t. \mathsf{zipwith}\ (\ominus)\ t\ (\mathit{top}\ (\mathsf{BRs}\ w)))\ (\mathsf{BRs}\ v) \ominus (\mathsf{BRs}\ w))))\ ((||a| \ominus z|) \ominus (\mathsf{BRs}\ z))))) \\
= & \{ \text{ definition of } \mathit{left} \text{ and } \mathsf{map}_r \ \} \\
 & \mathit{flatten}\ (\mathsf{map}\ \mathsf{TLs}\ (\mathsf{map}_c\ (\lambda t. \mathsf{zipwith}\ (\phi)\ t\ (\mathit{left} \\
 & \ (\mathsf{map}_r\ (\lambda t. \mathsf{zipwith}\ (\ominus)\ t\ (\mathit{top}\ (\mathsf{BRs}\ w)))\ (\mathsf{BRs}\ v)) \ominus \mathit{left}\ (\mathsf{BRs}\ w)))\ ((||a| \ominus z|) \ominus (\mathsf{BRs}\ z)))) \\
= & \{ \text{ split of } \mathsf{map}_c \ \} \\
 & \mathit{flatten}\ (\mathsf{map}\ \mathsf{TLs}\ (\mathsf{map}_c\ (\lambda t. \mathsf{zipwith}\ (\phi)\ t\ (\mathit{left} \\
 & \ (\mathsf{map}_r\ (\lambda t. \mathsf{zipwith}\ (\ominus)\ t\ (\mathit{top}\ (\mathsf{BRs}\ w)))\ (\mathsf{BRs}\ v))))\ (||a| \ominus z|) \\
 & \ \ \ominus (\mathsf{map}_c\ (\lambda t. \mathsf{zipwith}\ (\phi)\ t\ (\mathit{left}\ (\mathsf{BRs}\ w)))\ (\mathsf{BRs}\ z)))) \\
= & \{ \text{ definition of } \mathsf{map} \text{ and } \mathit{flatten} \ \} \\
 & \mathit{flatten}\ (\mathsf{map}\ \mathsf{TLs}\ (\mathsf{map}_c\ (\lambda t. \mathsf{zipwith}\ (\phi)\ t\ (\mathit{left} \\
 & \ (\mathsf{map}_r\ (\lambda t. \mathsf{zipwith}\ (\ominus)\ t\ (\mathit{top}\ (\mathsf{BRs}\ w)))\ (\mathsf{BRs}\ v))))\ (||a| \ominus z|))) \\
 & \ \ \ominus \mathit{flatten}\ (\mathsf{map}\ \mathsf{TLs}\ (\mathsf{map}_c\ (\lambda t. \mathsf{zipwith}\ (\phi)\ t\ (\mathit{left}\ (\mathsf{BRs}\ w)))\ (\mathsf{BRs}\ z)))
\end{array}
$$

The left hand size is as follows.

$$
\begin{array}{ll}
& \mathit{flatten}\ (\mathit{unwind}_{\sigma}\ (\mathsf{zipwith}_3\ f''_s\ (\mathit{rects}\ (|a| \ominus z))\ (\mathit{rights}\ (|a| \ominus z))\ (\mathit{lefts}\ (v \ominus w)))) \\
& \quad \textbf{where}\ f''_s\ s_1\ r_1\ l_2 = s_1 \phi\ \mathsf{map}\ (\mathsf{the}\ r_1 \phi)\ l_2) \\
= & \{ \text{ definition of } \mathit{rects}, \mathit{rights}, \text{ and } \mathit{lefts} \ \} \\
& \mathit{flatten}\ (\mathit{unwind}_{\sigma}\ (\mathsf{zipwith}_3\ f''_s\ ((|||a||| \phi\ \mathsf{map}\ (\mathsf{zipwith}\ (\ominus)\ ||a||)\ (\mathit{tops}\ z)) \ominus (\mathit{NIL} \phi\ \mathit{rects}\ z)) \\
& \ ((|||a||| \phi\ \mathsf{map}\ (\mathsf{zipwith}\ (\ominus)\ ||a||)\ (\mathit{toprights}\ z)) \ominus (\mathit{NIL} \phi\ \mathit{rights}\ z)) \\
& \ ((\mathit{lefts}\ v\ \phi\ \mathit{gemm}\ (_, \mathsf{zipwith}\ (\ominus))\ (\mathit{bottomlefts}\ v)\ (\mathit{toplefts}\ w)) \ominus (\mathit{NIL} \phi\ \mathit{lefts}\ w)))) \\
= & \{ \text{ definition of } \mathit{flatten}, \mathit{unwind}, \text{ and } \mathsf{zipwith}_3, \text{ and } \mathsf{zipwith}\ (\ominus)\ ||a|| = \mathsf{map}\ (|a| \ominus)\ \} \\
& |f''_s\ ||a||\ ||a||\ (\mathsf{the}\ (\mathit{lefts}\ v))| \\
& \ \ominus \mathit{flatten}\ (\mathsf{reduce}\ (_, \ominus)\ (\mathsf{map}\ |\cdot|\ (\mathsf{zipwith}_3\ f''_s\ (\mathsf{map}\ (\mathsf{map}\ (|a| \ominus))\ (\mathit{tops}\ z)) \\
& \ (\mathsf{map}\ (\mathsf{map}\ (|a| \ominus))\ (\mathit{toprights}\ z))\ (\mathit{gemm}\ (_, \mathsf{zipwith}\ (\ominus))\ (\mathit{bottomlefts}\ v)\ (\mathit{toplefts}\ w))))) \\
& \ \ominus \mathit{flatten}\ (\mathit{unwind}_{\sigma}\ (\mathsf{zipwith}_3\ f''_s\ (\mathit{rects}\ z)\ (\mathit{rights}\ z)\ (\mathit{lefts}\ w)))
\end{array}
$$

The induction hypothesis guarantees the following equation of the last half parts of the results of the above calculations.

$$
\begin{array}{l}
\mathit{flatten}\ (\mathsf{map}\ \mathsf{TLs}\ (\mathsf{map}_c\ (\lambda t.\, \mathsf{zipwith}\ (\phi)\ t\ (\mathit{left}\ (\mathsf{BRs}\ w)))\ (\mathsf{BRs}\ z))) \\
= \mathit{flatten}\ (\mathit{unwind}_{\sigma}\ (\mathsf{zipwith}_3\ f''_s\ (\mathit{rects}\ z)\ (\mathit{rights}\ z)\ (\mathit{lefts}\ w)))
\end{array}
$$

The rest of the proof is to show the following equation.

$$
\begin{array}{l}
\mathit{flatten}\ (\mathsf{map}\ \mathsf{TLs}\ (\mathsf{map}_c\ (\lambda t.\, \mathsf{zipwith}\ (\phi)\ t\ (\mathit{left} \\
\ (\mathsf{map}_r\ (\lambda t.\, \mathsf{zipwith}\ (\ominus)\ t\ (\mathit{top}\ (\mathsf{BRs}\ w)))\ (\mathsf{BRs}\ v))))\ (||a| \ominus z|))) \\
= |f''_s\ ||a||\ ||a||\ (\mathsf{the}\ (\mathit{lefts}\ v))| \\
\ \ominus \mathit{flatten}\ (\mathsf{reduce}\ (_, \ominus)\ (\mathsf{map}\ |\cdot|\ (\mathsf{zipwith}_3\ f''_s\ (\mathsf{map}\ (\mathsf{map}\ (|a| \ominus))\ (\mathit{tops}\ z)) \\
\ (\mathsf{map}\ (\mathsf{map}\ (|a| \ominus))\ (\mathit{toprights}\ z))\ (\mathit{gemm}\ (_, \mathsf{zipwith}\ (\ominus))\ (\mathit{bottomlefts}\ v)\ (\mathit{toplefts}\ w)))))
\end{array}
$$

The left hand side is calculated follows.

$$
\begin{array}{ll}
& \mathit{flatten}\ (\mathsf{map}\ \mathsf{TLs}\ (\mathsf{map}_c\ (\lambda t.\, \mathsf{zipwith}\ (\phi)\ t\ (\mathit{left} \\
& \ (\mathsf{map}_r\ (\lambda t.\, \mathsf{zipwith}\ (\ominus)\ t\ (\mathit{top}\ (\mathsf{BRs}\ w)))\ (\mathsf{BRs}\ v))))\ (||a| \ominus z|))) \\
= & \{ \text{ distribution of } \mathit{left}\ \} \\
& \mathit{flatten}\ (\mathsf{map}\ \mathsf{TLs}\ (\mathsf{map}_c\ (\lambda t.\, \mathsf{zipwith}\ (\phi)\ t\ (\\
& \ \mathsf{map}_r\ (\lambda t.\, \mathsf{zipwith}\ (\ominus)\ t\ (\mathit{left}\ (\mathit{top}\ (\mathsf{BRs}\ w)))\ (\mathit{left}\ (\mathsf{BRs}\ v)))))\ (||a| \ominus z|))) \\
= & \{ \ \mathit{left}\ (\mathit{top}\ (\mathsf{BRs}\ w)) = |w|, \text{ and } \mathit{left}\ (\mathsf{BRs}\ v) = |v|\ \} \\
& \mathit{flatten}\ (\mathsf{map}\ \mathsf{TLs}\ (\mathsf{map}_c\ (\lambda t.\, \mathsf{zipwith}\ (\phi)\ t\ (\mathsf{map}_r\ (\lambda t.\, \mathsf{zipwith}\ (\ominus)\ t\ |w|)\ |v|))\ (||a| \ominus z|))) \\
= & \{ \text{ simplification } \} \\
& \mathit{flatten}\ (\mathsf{map}\ \mathsf{TLs}\ (|(|a| \ominus z)\ \phi\ (v \ominus w)|)) \\
= & \mathsf{TLs}\ ((|a| \ominus z)\ \phi\ (v \ominus w))
\end{array}
$$

The right hand side is as follows.

$$
\begin{array}{ll}
 & |f''_s \ ||a|| \ ||a|| \ (\mathsf{the} \ (\mathit{lefts} \ v))| \\
 & \ominus \mathit{flatten} \ (\mathsf{reduce} \ (_,\ominus) \ (\mathsf{map} \ |\cdot| \ (\mathsf{zipwith}_3 \ f''_s \ (\mathsf{map} \ (\mathsf{map} \ (|a|\ominus)) \ (\mathit{tops} \ z)) \\
 & \quad (\mathsf{map} \ (\mathsf{map} \ (|a|\ominus)) \ (\mathit{toprights} \ z)) \ (\mathit{gemm} \ (_,\mathsf{zipwith} \ (\ominus)) \ (\mathit{bottomlefts} \ v) \ (\mathit{toplefts} \ w))))) \\
= & \{ \text{ definition of } f''_s\text{, and simplification } \} \\
 & ||a|| \phi \ \mathsf{map} \ (|a|\phi) \ (\mathsf{the} \ (\mathit{lefts} \ v)) \\
 & \ominus \mathit{flatten} \ (\mathsf{reduce} \ (_,\ominus) \ (\mathsf{map} \ |\cdot| \ (\mathsf{zipwith}_3 \ f''_s \ (\mathsf{map} \ (\mathsf{map} \ (|a|\ominus)) \ (\mathit{tops} \ z)) \\
 & \quad (\mathsf{map} \ (\mathsf{map} \ (|a|\ominus)) \ (\mathit{toprights} \ z)) \ (\mathit{gemm} \ (_,\mathsf{zipwith} \ (\ominus)) \ (\mathit{bottomlefts} \ v) \ (\mathit{toplefts} \ w))))) \\
= & \{ \ \mathit{flatten} \circ \mathsf{reduce} \ (_,\ominus) \circ \mathsf{map} \ |\cdot| = \mathsf{reduce} \ (_,\ominus) \ \} \\
 & ||a|| \phi \ \mathsf{map} \ (|a|\phi) \ (\mathsf{the} \ (\mathit{lefts} \ v)) \\
 & \ominus \mathsf{reduce} \ (_,\ominus) \ (\mathsf{zipwith}_3 \ f''_s \ (\mathsf{map} \ (\mathsf{map} \ (|a|\ominus)) \ (\mathit{tops} \ z)) \\
 & \quad (\mathsf{map} \ (\mathsf{map} \ (|a|\ominus)) \ (\mathit{toprights} \ z)) \ (\mathit{gemm} \ (_,\mathsf{zipwith} \ (\ominus)) \ (\mathit{bottomlefts} \ v) \ (\mathit{toplefts} \ w))) \\
= & \{ \ \mathit{bottomlefts} \ v \text{ is a singleton, and the definition of } \mathit{gemm} \ \} \\
 & ||a|| \phi \ \mathsf{map} \ (|a|\phi) \ (\mathsf{the} \ (\mathit{lefts} \ v)) \\
 & \ominus \mathsf{reduce} \ (_,\ominus) \ (\mathsf{zipwith}_3 \ f''_s \ (\mathsf{map} \ (\mathsf{map} \ (|a|\ominus)) \ (\mathit{tops} \ z)) \\
 & \quad (\mathsf{map} \ (\mathsf{map} \ (|a|\ominus)) \ (\mathit{toprights} \ z)) \ (\mathsf{map} \ (\mathsf{zipwith} \ (\ominus)) \ (\mathsf{the} \ (\mathit{bottomlefts} \ v)) \ (\mathit{toplefts} \ w))) \\
= & \{ \text{ definition of } \mathsf{reduce} \ (_,\ominus) \ \} \\
 & \mathsf{reduce} \ (_,\ominus) \ (|||a|| \phi \ \mathsf{map} \ (|a|\phi) \ (\mathsf{the} \ (\mathit{lefts} \ v))| \\
 & \quad \phi \ (\mathsf{zipwith}_3 \ f''_s \ (\mathsf{map} \ (\mathsf{map} \ (|a|\ominus)) \ (\mathit{tops} \ z)) \\
 & \quad (\mathsf{map} \ (\mathsf{map} \ (|a|\ominus)) \ (\mathit{toprights} \ z)) \ (\mathsf{map} \ (\mathsf{zipwith} \ (\ominus)) \ (\mathsf{the} \ (\mathit{bottomlefts} \ v)) \ (\mathit{toplefts} \ w)))) \\
= & \{ \text{ equivalence under } \mathit{height} \ v = 1 \text{ and } \mathit{width} \ z = 1 \ \} \\
 & \mathsf{reduce} \ (_,\ominus) \ (|||a|| \phi \ \mathsf{map} \ (|a|\phi) \ (\mathsf{the} \ (\mathit{toplefts} \ v))| \\
 & \quad \phi \ (\mathsf{zipwith}_3 \ f''_s \ (\mathsf{map} \ (\mathsf{map} \ (|a|\ominus)) \ (\mathit{toplefts} \ z)) \\
 & \quad (\mathsf{map} \ (\mathsf{map} \ (|a|\ominus)) \ (\mathit{toplefts} \ z)) \ (\mathsf{map} \ (\mathsf{zipwith} \ (\ominus)) \ (\mathsf{the} \ (\mathit{toplefts} \ v)) \ (\mathit{toplefts} \ w)))) \\
= & \{ \text{ change of } f''_s \text{ to } f'_{tl} \ \} \\
 & \mathsf{reduce} \ (_,\ominus) \ (|||a|| \phi \ \mathsf{map} \ (|a|\phi) \ (\mathsf{the} \ (\mathit{toplefts} \ v))| \\
 & \quad \phi \ (\mathsf{zipwith} \ f'_{tl} \ (\mathsf{map} \ (\mathsf{map} \ (|a|\ominus)) \ (\mathit{toplefts} \ z)) \\
 & \quad (\mathsf{map} \ (\mathsf{zipwith} \ (\ominus)) \ (\mathsf{the} \ (\mathit{toplefts} \ v)) \ (\mathit{toplefts} \ w)))) \\
 & \quad\quad \textbf{where} \ f'_{tl} \ tl_1 \ tl_2 = tl_1 \ \phi \ \mathsf{map} \ (\mathsf{the} \ tl_1\phi) \ tl_2) \\
= & \{ \ \mathit{right}' \ |a| = \mathsf{the} \ |a|\text{, and } \mathsf{map} \ (|a|\ominus) = \mathsf{zipwith} \ (\ominus) \ ||a|| \ \} \\
 & \mathsf{reduce} \ (_,\ominus) \ (f_{tl} \ ||a|| \ (\mathsf{the} \ (\mathit{toplefts} \ v)) \\
 & \quad \phi \ (\mathsf{zipwith} \ f_{tl} \ (\mathsf{map} \ (\mathsf{zipwith} \ (\ominus) \ ||a||) \ (\mathit{toplefts} \ z)) \\
 & \quad (\mathsf{map} \ (\mathsf{zipwith} \ (\ominus)) \ (\mathsf{the} \ (\mathit{toplefts} \ v)) \ (\mathit{toplefts} \ w)))) \\
 & \quad\quad \textbf{where} \ f_{tl} \ tl_1 \ tl_2 = tl_1 \ \phi \ \mathsf{map} \ (\mathit{right}' \ tl_1\phi) \ tl_2) \\
= & \{ \text{ definition of } \mathsf{zipwith} \ \} \\
 & \mathsf{reduce} \ (_,\ominus) \ (\mathsf{zipwith} \ f_{tl} \ (|||a||| \phi \ \mathsf{map} \ (\mathsf{zipwith} \ (\ominus) \ ||a||) \ (\mathit{toplefts} \ z)) \\
 & \quad (\mathit{toplefts} \ v \ \phi \ \mathsf{map} \ (\mathsf{zipwith} \ (\ominus)) \ (\mathsf{the} \ (\mathit{toplefts} \ v)) \ (\mathit{toplefts} \ w))) \\
= & \{ \text{ definition of } \mathit{toplefts} \ \} \\
 & \mathsf{reduce} \ (_,\ominus) \ (\mathit{toplefts} \ ((|a| \ominus z) \ \phi \ (v \ominus w)))
\end{array}
$$

Since we have shown the dilation $\mathsf{TLs} = \mathsf{reduce} \ (_,\ominus) \circ \mathit{toplefts}$, we have $\mathsf{TLs} \ ((|a| \ominus z) \ \phi \ (v \ominus w)) = \mathsf{reduce} \ (_,\ominus) \ (\mathit{toplefts} \ ((|a| \ominus z) \ \phi \ (v \ominus w)))$. Therefore, we have shown the induction case $(x \ \phi \ y)$ of the dilation of rects'.

The other induction case is shown similarly.

Publication Lists

International Journals

1. Kento Emoto, Zhenjiang Hu, Kazuhiko Kakehi, Masato Takeichi:
   A Compositional Framework for Developing Parallel Programs on Two-Dimensional Arrays.
   International Journal of Parallel Programming, Vol. 35, No. 6, pp. 615–658, Springer Netherlands, 2007.

Domestic Journals

2. 野村 芳明, 江本 健斗, 松崎 公紀, 胡 振江, 武市 正人:
   木スケルトンによる XPath クエリの並列化とその評価.
   コンピュータソフトウェア, Vol. 24, No. 3, pp. 51–62, 2007.

Refereed Papers (International Conferences and Workshops)

3. Kento Emoto, Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi:
   Domain-Specific Optimization Strategy for Skeleton Programs.
   In *13th International Euro-Par Conference, Rennes ,France , August 28-31, 2007. Proceedings*, Lecture Notes in Computer Science 4641, pp. 705–714, Springer, 2007.

4. Kazuhiko Kakehi, Kiminori Matsuzaki, and Kento Emoto:
   Efficient Parallel Tree Reductions on Distributed Memory Environments.
   In *Computational Science — ICCS 2007*, Lecture Notes in Computer Science 4488, pp. 601–608, Springer, 2007.

5. Kento Emoto, Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi:
   Surrounding Theorem: Developing Parallel Programs for Matrix-Convolutions.
   In *12th International Euro-Par Conference, Dresden, Germany, August/September 2006, Proceedings*, Lecture Notes in Computer Science 4128, pp. 605–614, Springer, 2006.

6. Kiminori Matsuzaki, Kento Emoto, Hideya Iwasaki, and Zhenjiang Hu:
   A Library of Constructive Skeletons for Sequential Style of Parallel Programming.
   In *First International Conference on Scalable Information Systems (InfoScale 2006), May 29–June 1, 2006, Hong Kong*, pp. 12, 2006.

Domestic Conference and Workshop or Non-Refereed Papers

7. Kento Emoto, Zhenjiang Hu, Kazuhiko Kakehi, Kiminori Matsuzaki, and Masato Takeichi:
   A Generative Matrix Library in Fortress Relieves Programmers' Headache!
   日本ソフトウェア科学会第 **25** 回大会論文集, 筑波大学, 2009.

8. 江本 健斗, 松崎 公紀, 胡 振江, 武市 正人:
近傍要素を必要とするスケルトンプログラムの最適化.
第 **9** 回プログラミングおよびプログラミング言語ワークショップ **(PPL 2007)** 論文集, pp. 125–139, 2007.

9. 野村 芳明, 江本 健斗, 松崎 公紀, 胡 振江, 武市 正人:
木スケルトンによる XPath クエリの並列化とその評価.
日本ソフトウェア科学会第 **22** 回大会論文集, 東北大学, 2005.

10. 松崎 公紀, 明石 良樹, 江本 健斗, 岩崎 英哉, 胡 振江:
助っ人：構成的な並列スケルトンによる並列プログラミングライブラリ.
日本ソフトウェア科学会第 **22** 回大会論文集, 東北大学, 2005.

11. Kazuhiko Kakehi, Kiminori Matsuzaki, Akimasa Morihata, Kento Emoto, and Zhenjiang Hu:
Parallel Dynamic Programming using Data-Parallel Skeletons.
日本ソフトウェア科学会第 **22** 回大会論文集, 東北大学, 2005.

12. 江本 健斗, 胡 振江, 筧 一彦, 武市正人:
二次元配列上の構成的並列スケルトンの実現.
日本ソフトウェア科学会第 **21** 回大会論文集, 東京工業大学, 2004.

Technical Reports

13. Kento Emoto, Zhenjiang Hu, Kazuhiko Kakehi, Kiminori Matsuzaki, and Masato Takeichi:
Generator-based GG Fortress Library
Technical Report METR 2008-16, Department of Mathematical Engineering and Information Physics, the University of Tokyo, 2008.

14. Kento Emoto, Zhenjiang Hu, Kazuhiko Kakehi, Kiminori Matsuzaki, and Masato Takeichi:
Generator-based GG Fortress Library—Collection of GGs and Theories—
Technical Report METR 2008-17, Department of Mathematical Engineering and Information Physics, the University of Tokyo, 2008.

15. Kento Emoto, Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi:
Domain-Specific Optimization for Skeleton Programs Involving Neighbor Elements.
Technical Report METR 2007-05, Department of Mathematical Engineering and Information Physics, the University of Tokyo, 2007.

16. Kazuhiko Kakehi, Kiminori Matsuzaki, Kento Emoto, and Zhenjiang Hu:
An Practicable Framework for Tree Reductions under Distributed Memory Environments.
Technical Report METR 2006-64, Department of Mathematical Engineering and Information Physics, the University of Tokyo, 2006.

17. Kento Emoto, Zhenjiang Hu, Kazuhiko Kakehi, and Masato Takeichi:
A Compositional Framework for Developing Parallel Programs on Two Dimensional Arrays.
Technical Report METR2005-09, Department of Mathematical Engineering and Information Physics, the University of Tokyo, 2005.

Presentations

18. 江本 健斗, 胡 振江, 筧 一彦, 松崎 公紀, 武市 正人:
プログラム運算に基づく最適化機能つき Fortress ライブラリ.
第 11 回プログラミングおよびプログラミング言語ワークショップ (PPL 2009), カテゴリ 3, 2009.

19. 江本 健斗, 胡 振江, 筧 一彦, 松崎 公紀, 武市 正人:
Generator-of-generators に基づく Fortress ライブラリ.
第６９回プログラミング研究会 (PRO-2008-1), 2008.

20. 江本 健斗:
助っ人：構成的な並列スケルトンによる並列プログラミングライブラリ.
第８回プログラミングおよびプログラミング言語ワークショップ (PPL 2006), カテゴリ 3, 2006.

21. Kento Emoto:
Developing Parallel Programs for Matrix-Convolutions with Constructive Skeletons
Presentation at Informal Workshop on Skeletal Parallel Programming, the University of Tokyo, 2006.