Parallel Programming with Tree Skeletons

（並列木スケルトンによる並列プログラミングの
理論と実現に関する研究）

松崎　公紀

Acknowledgements

Many people helped me to achieve this work. I would like to give my sincere gratitude to them, in particular to the following.

First of all, I would like to thank Associate Professor Dr. Zhenjiang Hu, who had been my supervisor at my course of the graduate school of the University of Tokyo, for his kind and helpful encouragement and guidance over six years. I started the research in this thesis at his interesting and solid work, and I am very glad to make this work closely with him. I also want to thank his invitation to my submission of the thesis.

Special thanks go Professor Dr. Masato Takeichi for his insightful comments that lead my research to the right direction, and Professor Dr. Hideya Iwasaki for his encouragements to carrying out this research. They as well as Zhenjiang Hu kindly took their time for discussion with me, and without them this work may not get to the accomplishment. The committee members gave several helpful comments to this thesis.

I have done this work with many members of the Information Processing Laboratory and the SkeTo project. Especially, I should thank Dr. Kazuhiko Kakehi, Mr. Kento Emoto, Mr. Akimasa Morihata, Mr. Kazutaka Matsuda, and Mr. Yoshiki Akashi, for the collaborative research with them. I indeed enjoyed the discussion with them.

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. I also acknowledge the financial support by Japan Society for the Promotion of Science and Ministry of Education, Culture, Sports, Science and Technology.

Last but not least, my gratitude and love are to my father and mother for their support and deep love.

Abstract

Parallel computing is an essential technique to deal with large scaled problems. In recent years, while hardware for parallel computing is getting widely available, developing software for parallel computing remains as a hard task for many programmers. The main difficulties are caused by the communication, synchronization, and data distribution required in parallel programs.

This thesis studies the theory and practice of parallel programming for trees based on parallel primitives called tree skeletons. Trees are important data structures for representing structured data. However, their irregular and ill-balanced structure makes it hard to develop efficient parallel programs on them, because naive divide-and-conquer parallel computation may lead to poor performance for ill-balanced trees. To remedy this situation, this thesis develops a new framework for parallel programming for trees on the basis of the programming model called skeletal parallel programming. Skeletal parallel programming, first proposed by Cole, encourages programmers to develop parallel programs by composing ready-made components called parallel skeletons (or algorithmic skeletons). A theory has been proposed for design of parallel skeletons for lists based on constructive algorithms, and several libraries of parallel skeletons have been developed to bring the theory into practice. This thesis extends these ideas from lists to trees.

The following are three important contributions in the thesis.

The first contribution is the design of parallel tree skeletons for both binary trees and general trees of arbitrary shape. Our parallel tree skeletons have a sequential interface but with a parallel implementation; the sequential interface is designed based on the theory of constructive algorithmics, while the parallel implementation is either based on tree contraction algorithms or newly developed ones.

The second contribution is a set of theories for skeletal parallel programming on trees. These theories provide us with a systematic method for deriving skeletal parallel programs from sequential programs. We illustrate effectiveness of the method by solving two classes of nontrivial problems, maximum marking problems and XPath queries.

The third contribution is an implementation of a parallel skeleton library for trees. We developed a new implementation algorithm for tree skeletons, in which a tree is divided with high locality and good load balance and tree skeletons are executed efficiently in

parallel even on distributed-memory parallel computers. These skeletons are implemented in C++ with the MPI library, and provided as a skeleton library called SkeTo.

Comittee

Associate Professor	Zhenjiang Hu
Professor	Masami Hagiya
Professor	Akihiko Takano
Professor	Masato Takeichi
Associate Professor	Takayasu Matsuo
Associate Professor	Kenjiro Taura

Contents

Chapter 1

Introduction

1.1 Background

The amount of data we need to deal with is getting larger and larger, and parallel computing is an essential technique for obtaining sufficient computational power to manipulate huge amount of data. Parallel computing has become widely available due to faster and cheaper computers and networks, such as clusters of tens or even hundreds of PCs connected by Gigabit Ethernet networks. Furthermore, multi-core processors, which are now also available for PCs, offer more opportunities for parallel computing.

Though parallel computers are widely available, parallel programming is still a difficult task. One reason is that parallel programs are more complicated than sequential ones. Programmers developing efficient parallel programs must take into account communication among processors, synchronization among processes, and allocation of data and resources. Another reason is the large variety of parallel computer architectures. Initially, shared memory computers modeled as Parallel Random Access Machines (PRAMs) were used; now, distributed-memory architectures such as PC clusters are also used. In the future, more complicated architectures achieved by integrating chip-level multiprocessing and grid computing will be the target platform for parallel programs.

Skeletal parallelism, first proposed by Cole [33] and described well by Rabhi and Gorlatch [110], is a novel paradigm for overcoming these difficulties. In skeletal parallelism, users build parallel programs by combining ready-made components called *parallel skeletons* (or *algorithmic skeletons*). These parallel skeletons are abstract computational patterns that can be implemented efficiently in parallel on many parallel computers. Skeletal parallelism provided by these parallel skeletons conceals their complicated parallel implementations form users. Skeletal parallelism has several advantages; the two most important ones are that users can build parallel programs as if they were building sequential ones without considering implementation details and that the programs are not only efficient but also architecture independent.

There have been many studies on skeletal parallelism for lists and arrays [19, 32, 34, 52, 65, 117, 118], some of which have used constructive algorithmics [15, 18, 67] to formalize

parallel skeletons. Constructive algorithmics is a theory originally proposed for systematic development of sequential algorithms. Formalization based on constructive algorithmics endows the parallel skeletons with many attractive features, allowing optimization by fusion transformation [65], for example.

This thesis addresses parallel programming for trees, which are important datatypes that are often used to represent structured data such as XML. As XML has become the de facto standard for storing and exchanging data, a growing amount of data is stored in the form of trees. As a result, there is a great demand for methods and systems for manipulating huge trees efficiently. Parallel computing is a promising approach to meeting this demand.

Developing efficient parallel tree programs is, however, a much harder task for programmers than developing list programs, because of the irregular and ill-balanced structure of trees. In sequential programming, programs for manipulating trees are written using recursive functions. In parallel programming, the computations that can be performed in parallel must be identified. A naive way to transform a recursive function on trees into a parallel function is to program the independent recursive calls to be computed in parallel. This corresponds to the parallel divide-and-conquer approach, but naive divide-and-conquer programs are inefficient when the input trees are ill-balanced.

Tree contraction algorithms, first proposed by Miller and Reif [98], are well-known parallel algorithms for manipulating trees. The main idea of these algorithms is to perform local contractions (remove some nodes by merging them with an adjacent node) in parallel not only at the bottom of the tree but also at the middle point. The most important advantage of tree contraction algorithms is that the parallel computation time is guaranteed even if the input tree is completely ill-balanced. There have been many studies on the implementation of tree contraction algorithms for several architecture models [2, 8, 46, 94, 98, 125], and on the derivation of tree contraction algorithms for problems related to trees and graphs [35, 57, 86, 100, 101].

Though there have been many studies on tree contraction algorithms, programmers still find it problematic to develop programs by tree contraction algorithms. There are three problems in particular.

- The implementation of a tree contraction algorithm greatly depends on the target architecture model, and there has been insufficient abstraction based on solid theory. This complicates the derivation of parallel programs from sequential programs.

- There have been few studies on manipulating general trees by using tree contraction algorithms. The original tree contraction algorithms by Miller and Reif [98] did not limit the tree shape to binary, but many implementations of these algorithms require the assumption of a binary-tree shape for efficiency. Therefore, systematic methods of manipulating general trees based on the manipulation of binary trees are required.

- While there are a few libraries and tools that support development of parallel tree programs, they mainly work on shared-memory parallel computers [38, 111]. Since distributed-memory parallel computers, such as PC clusters, are now widely used, an efficient library of parallel tree manipulations for use on distributed-memory parallel computers is necessary.

This thesis examines principles and practices for constructing parallel programs to manipulate trees in the context of skeletal parallelism. In the following section, a "short tour" is presented to illustrate the contributions of this thesis.

1.2 A Short Tour

The contributions of this thesis are illustrated with a well-known problem on trees called the *party planning problem* [36]. This problem is an instance of the maximum marking problems [17, 115] for which the derivation of parallel programs is described in Chapter 8.

> The president of a company wants to have a company party. To make the party fun for all attendees, the president does not want both an employee and his or her direct supervisor to attend. The company has a hierarchical structure; that is, the supervisory relations form a tree rooted at the president. The personnel office has ranked each employee with a conviviality rating, which is a real number. Given the structure of the company and the ratings of the employees, the problem is to select the guests so that the sum of their conviviality ratings is maximized.

The structure of the company is assumed to be a binary tree, for simplicity. In the following, let us derive a parallel program that returns a binary tree of boolean values where the selection of a node is represented by a mark of True.

The first step in the tour is to develop a sequential program. A known sequential program for solving the party planning problem is given in Figure 1.1. It uses two recursive functions: `max_sums` and `mark_node`. Recursive function `max_sums` takes a tree and computes two values in a bottom-up manner: the first value is the maximum sum when the root node is marked; the second value is the maximum sum when the root is not marked. Recursive function `mark_node` computes the resulting mark for each node in a top-down manner using `max_sums`.

The next step is to develop an efficient parallel program from this sequential program. One approach is to use a naive divide-and-conquer manner, in which we compute two recursive functions `max_sums` and `mark_node` in parallel. This is acceptable because these calls are independent. This approach works well for balanced binary trees but not always for ill-balanced trees. Another approach is to utilize the tree contraction algorithms. However, these tree contraction algorithms have been mainly developed for shared-memory

```
pair max_sums( node<int> n ) {
  if ( n.is_leaf( ) ) {
    return pair( n.v, 0 );
  } else {
    pair lv = max_sums( n.l );
    pair rv = max_sums( n.r );
    return pair( n.v + lv.snd + rv.snd,
                 max( lv.fst, lv.snd ) + max( rv.fst, rv.snd ) );
  }
}

node<bool> mark_node( node<int> n, bool is_parent_marked ) {
  node<bool> ret;
  pair sums = max_sums( n );
  ret.v = ( !is_parent_marked ) && ( sums.fst > sums.snd );
  if ( !n.is_leaf( ) ) {
    ret.l = mark_node( n.l, ret.v );
    ret.r = mark_node( n.r, ret.v );
  }
  return ret;
}

int main( int argc, char** argv ) {
  ...
  node<bool> result = mark_node( tree, false );
  ...
```

Figure 1.1. Sequential program for solving party planning problem.

```
pair max_sums_leaf( int v ) {
  return pair( v, 0 );
}

pair max_sums_node( int v, pair lv, pair rv ) {
  return pair( v + lv.snd + rv.snd,
               max( lv.fst, lv.snd ) + max( rv.fst, rv.snd ) );
}

bool mark_node_g( pair v, bool is_parent_marked ) {
  return ( !is_parent_marked ) && ( v.fst > v.snd );
}

int main( int argc, char** argv ) {
  ...
  node<pair> tree2 = uAcc( max_sums_leaf, max_sums_node, tree1 );
  node<bool> tree3 = dAcc( mark_node_g, mark_node_g, false,
                           tree2 );
  ...
}
```

Figure 1.2. Skeletal sequential program for solving party planning problem.

architecture models, so it is hard for sequential programmers to develop parallel programs based on them.

In this thesis, an approach is proposed based on *skeletal parallel programming* for developing parallel programs manipulating trees. One goal is to enable users to develop parallel programs systematically by composing *parallel tree skeletons* without considering the details of the parallel algorithms. Another goal is to have the skeletal parallel programs run reasonably quickly with efficient implementation of the parallel tree skeletons.

The first task is to design parallel tree skeletons based on the solid theory of constructive algorithmics. As illustrated in Figure 1.1, we often develop programs for manipulating trees as recursive functions on the structure of trees. These recursive functions, which are defined over the data structure, can be formalized as homomorphisms in constructive algorithmics theory. In the case of binary trees, these functions are called *tree homomorphisms*. Skillicorn [119] formalized five primitive functions, called *tree skeletons*, for parallel programming on trees. These tree skeletons are special cases of tree homomorphisms and capture a wide area of computations on trees. On the basis of Skillicorn's study, we formalize parallel tree skeletons that have an interface of recursive functions for binary trees (Chapter 2) and general trees (Chapter 4).

With the recursive interface of tree skeletons, the sequential program shown in Figure 1.1 can be converted into the skeletal program shown in Figure 1.2. The function `max_sums` in the sequential program computes in a bottom-up manner, so it is implemented using the upwards accumulate skeleton (`uAcc`) with two parametric functions (`max_sums_leaf` called for leaves and `max_sums_node` called for internal nodes). The function `mark_node` computes in a top-down manner, so it is implemented using the downwards accumulate skeleton (`dAcc`) with one parametric function (`mark_node_g` called for left and right children). As shown by the example in Figure 1.2, parallel skeletons can be used for many types of programs if suitable parametric functions are supplied. The decomposition of complex recursive functions into combinations of tree skeletons can be systematically done by applying *diffusion theorems*, which are described in Chapter 5. It is worth noting that the skeletal program shown in Figure 1.2 is still a sequential program since we only provide parametric functions that specify sequential computation.

The second task is to formalize the conditions for parallel execution of skeletons through the use of systematic derivation. We can develop efficient parallel programs by using tree contraction algorithms, but there are some conditions for using them. The conditions for parallel computations on binary trees are formalized based on their *balanced ternary-tree representations* in Chapter 3. Since the conditions formalized in Chapter 3 may still be unfamiliar to users, three algebraic properties of the parametric functions are described and a systematic method of deriving auxiliary functions for parallel implementation given these properties is proposed in Chapter 5.

In our example, the function `max_sums` uses two operators, + and `max`, that form a commutative semiring, and the function `mark_node_g` returns either `true` or `false`

for each node in the tree. The former attribute, called the "tupled-ring property", is formalized in Section 5.2.4, and the latter, called the "finiteness property", is formalized in Section 5.2.2. These properties are used to derive parallel programs systematically. In fact, some steps in the derivation can be performed automatically by code generators (Section 7.3) that take sequential functions with some annotations and generate parallel programs with tree skeletons. The generated parallel program for the party planning problem is shown in Figure 1.3. The `SKETO_DEF_UNOP`s and `SKETO_DEF_TEROP`s define parametric functions for the tree skeletons.

The third task is to provide an efficient implementation of tree skeletons. The developed parallel skeleton library, *SkeTo*[1] (Chapter 7), does this. In this skeleton library, tree skeletons are implemented in C++ on MPI (message passing interface) libraries for distributed-memory parallel computers. The skeletal parallel program shown in Figure 1.3 can be executed on the SkeTo library. We developed a new implementation algorithm of parallel tree skeletons for distributed-memory computers, which has two important properties of efficient parallel programs, high locality and good load balance (Chapter 6). The tree skeletons were implemented carefully to achieve high performance on distributed trees based on m-bridges in basic graph theory. The cost model of the tree skeletons helps to achieve good load balance.

The experimental results for the derived parallel program are plotted in Figure 1.4. The input trees were balanced one, randomly generated one, and completely ill-balanced one, each with 16,777,215 ($= 2^{24} - 1$) nodes. The program was executed on a cluster of identical PCs with two Pentium 4 2.8-GHz CPUs and 2-GB memory (one CPU is used for each PC). The PCs were connected by Gigabit Ethernet. Even for the ill-balanced tree, there was a good speedup. The experimental results are presented in Section 7.4.

1.3 Contributions and Organization of the Thesis

The body of the thesis consists of two parts. The first part (Chapters 2–5) addresses the principle of skeletal parallel programming for trees. These chapters cover the design of tree skeletons and the systematic derivation of skeletal parallel programs from given sequential programs. The second part (Chapters 6–9) addresses the practice of parallel programming for trees. These chapters describe the implementation of the SkeTo parallel skeleton library for distributed-memory environments and illustrate the effectiveness of parallel tree skeletons using two classes of nontrivial examples.

Principles of Parallel Programming for Trees

In Chapter 2, the notational conventions are presented and important previous studies on parallel programming for binary trees are reviewed. First, a general definition of

[1] SkeTo is the abbreviation for "SKEletons in TOkyo"; it also means a supporter or a helper in Japanese.

```
struct max_sums_ret {
  int v0, v1;
};

struct max_sums_inter {
  int v;
  int a00, a01, a10, a11;
};

SKETO_DEF_UNOP( max_sums_leaf, int,
                max_sums_ret,
  max_sums_ret ret;
  ret.v0 = x; ret.v1 = 0;
  return ret; )

SKETO_DEF_TEROP( max_sums_node, int, max_sums_ret, max_sums_ret,
                 max_sums_ret,
  max_sums_ret ret;
  ret.v0 = x + y.v1 + z.v1;
  ret.v1 = max( y.v0, y.v1 ) + max( z.v0, z.v0 );
  return ret; )

SKETO_DEF_UNOP( max_sums_phi, int,
                max_sums_inter,
  max_sums_inter ret;
  ret.v = x;
  ret.a00 = 0;        ret.a01 = INT_MIN;
  ret.a10 = INT_MIN;  ret.a11 = 0;
  return ret; )

SKETO_DEF_TEROP( max_sums_psiN, max_sums_inter, max_sums_ret,
                 max_sums_ret, max_sums_ret,
  max_sums_ret ret0 = max_sums_node( x.v, y, z );
  max_sums_ret ret;
  ret.v0 = max( x.a00 + ret0.v0, x.a01 + ret0.v1 );
  ret.v1 = max( x.a10 + ret0.v0, x.a11 + ret0.v1 );
  return ret; )

/* definition of other function objects */

int SketoMain( int argc, char** argv ) {
  ...
  dist_tree<max_sums_ret> *tree2
    = uAcc( max_sums_leaf, max_sums_node, max_sums_phi,
            max_sums_psiN, max_sums_psiL, max_sums_psiR,
            tree1 );
  dist_tree<bool> *tree3
    = dAcc( mark_node_gl, mark_node_gr, mark_node_phil,
            mark_node_phir, mark_node_psiu, mark_node_psid,
            false, tree2 );
  ...
```

Figure 1.3. Skeletal parallel program for solving party planning problem.

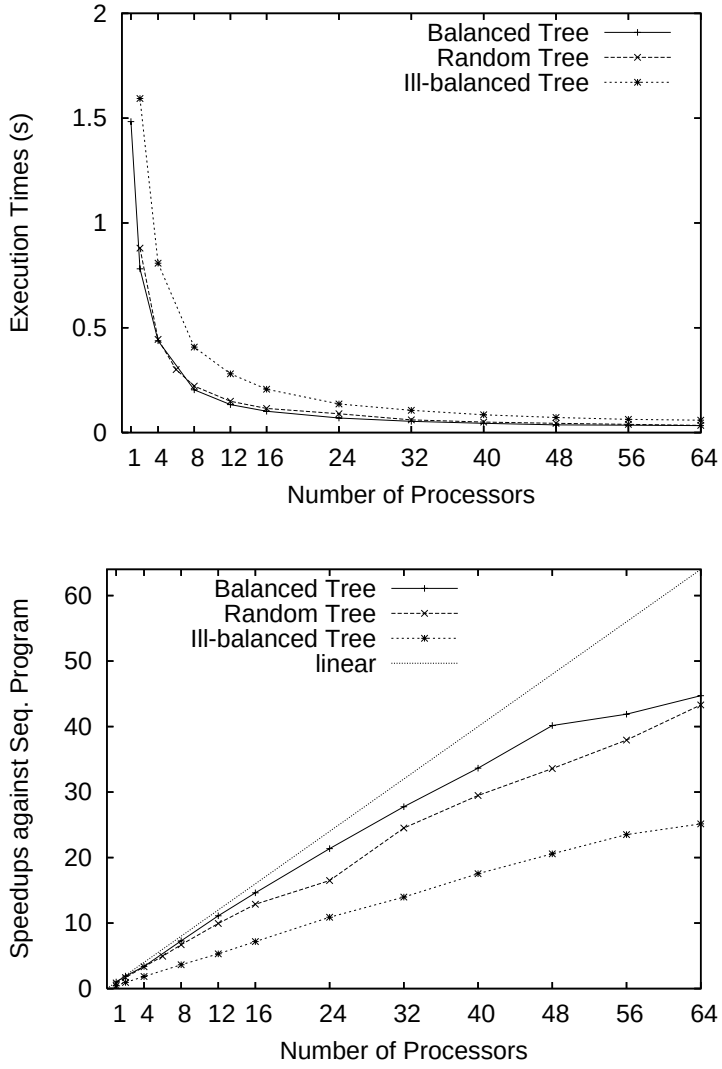


Figure 1.4. Execution time (top) and speedups (bottom) of the parallel program for the party planning problem.

tree computations called "tree homomorphisms" is introduced followed by two important studies on tree contraction algorithms: Miller and Reif's original tree contraction algorithm [98] and a novel implementation of the tree contraction algorithms on an exclusive-read exclusive-write (EREW) PRAM [2]. Then, the binary-tree skeletons, first formalized by Skillicorn [119], are introduced with some extensions.

In Chapter 3, the parallelism in the computation on binary trees is formalized. We propose the *balanced ternary-tree representation* based on the flexible division of trees, and formalize *tree associativity* on an analogy to the associativity in the parallel computation on lists. We develop a parallel implementation of basic tree skeletons on this ternary-tree representation, and investigate an algorithm for generating balanced ternary trees from given binary trees.

In Chapter 4, parallel skeletons for general trees with an arbitrary shape, called rose trees [96], are examined. There have been many studies on the parallel manipulation of binary trees, but few studies on general trees. We define *seven basic rose-tree skeletons* based on the idea of constructive algorithmics. These rose-tree skeletons are a straightforward extensions of parallel skeletons for binary trees, and can be implemented in parallel by using parallel binary-tree skeletons.

In Chapter 5, the derivation of skeletal parallel programs is described. Sequential programs for manipulating trees are often given as recursive functions, and there are often gaps between sequential recursive programs and skeletal parallel programs. We develop *diffusion theorems* for decomposing a complicated recursive function into simpler recursive functions that can be computed using tree skeletons. We, then show *three algebraic properties for the systematic derivation* of auxiliary functions for parallel tree skeletons.

Practices of Parallel Programming on Trees

In Chapter 6, implementation algorithms of parallel binary-tree skeletons for distributed-memory parallel computers are described. This implementation has two major features: good performance for local computation due to the serialized representation of trees, and concise cost model of the skeletons, which supports load balancing using information on the cost of parameter functions.

In Chapter 7, the implementation issues of the SkeTo library are shown. Three important issues are: implementation of the skeleton library in C++ and MPI based on the implementation algorithm of the binary-tree skeletons, an optimization mechanism using fusion transformation implemented using meta-programming techniques, and code generators based on the theories developed in Chapter 5. Several experiment results are shown here.

In Chapter 8, the derivation of parallel programs is shown for the maximum marking problems. The maximum marking problems include many important dynamic programming problems, an instance of which is the party planning problem used in the short tour.

Using the derivation methods presented, we can easily derive parallel programs for a class of maximum marking problems from efficient sequential programs given by Sasano et al.'s derivation methods [115, 116].

In Chapter 9, a more involved example is shown: the parallelization of XPath queries. XPath is a widely used language for addressing parts of an XML tree, but few studies have addressed the parallel processing of XPath queries. Parallel tree skeletons are shown to be applicable to the implementation of nontrivial XPath queries.

Chapter 10, related work is reviewed. It includes work on tree contraction algorithms and their applications, other approaches to implementing parallel computing for trees, skeletal parallel programming for trees and other data structures, program derivation techniques for parallelization, and skeletal environments.

In Chapter 11, the thesis concludes with remarks on future directions.

Chapter 2

Basis of Parallel Tree Computing on Binary Trees

2.1 Notations

In this thesis, we borrow the notation of Haskell [16, 107]. In the following, we briefly introduce important notations and data structures used in the thesis. Roughly speaking, the functions in this thesis can be read as mathematical function definitions except for the function applications denoted by spaces.

Functions and Operators

Function application is denoted by a space and the argument may be written without brackets. Thus $f\ a$ means $f(a)$. Functions are curried, and the function application associates to the left. Thus $f\ a\ b$ means $(f\ a)\ b$. The function application binds stronger than any other operator, so $f\ a \oplus b$ means $(f\ a) \oplus b$, but does not $f\ (a \oplus b)$. Function composition is denoted by an infix operator $\circ$. By definition, we have $(f \circ g)\ a = f\ (g\ a)$. Function composition is associative and its unit is the identity function denoted by id.

In some cases arguments do not affect the result of the functions. In such cases the arguments may be called don't-care and denoted as $\_$.

Infix binary operators will be denoted by $\oplus$, $\otimes$, etc, and their units are written as $\iota_\oplus$, $\iota_\otimes$, respectively. These operators can be sectioned and be treated as functions, i.e. $a \oplus b = (a\oplus)\ b = (\oplus b)\ a = (\oplus)\ a\ b$ holds.

In addition to the arithmetic operators, we use the following two operators. Operator $\uparrow$ returns the larger of the two arguments, and operator $\downarrow$ returns the smaller of the two. Using **if**-expression, we can define these two operators as follows.

$$a \uparrow b = \textbf{if } a > b \textbf{ then } a \textbf{ else } b$$
$$a \downarrow b = \textbf{if } a < b \textbf{ then } a \textbf{ else } b$$

Tuples

Tuples are constructed with the fixed number of elements that may have different types. We donate tuples with parenthesis such as $(1, 2)$ and $(1, 0.1, \texttt{"a"})$. A tuple with exact two elements is called a pair, and a tuple with exact three elements is called a triple.

Function *fst* takes out the first element of the input tuple. Similarly, functions *snd* and *thd* take out the second and third elements, respectively.

Some functions that take an input of the same type can be gathered by operator $\triangle$. For example, $(f \triangle g)\ x = (f\ x, g\ x)$.

Lists and List Comprehension

Lists are finite sequences of elements of the same type. Lists can be constructed in two ways; cons lists and join lists. A cons list is constructed either by an empty list (Nil) or by adding an element x to a list xs (Cons x xs). The datatype of a cons list whose elements are of type α is defined as follows.

$$\begin{aligned}&\textbf{data}\ \mathsf{List}\ \alpha = \mathsf{Nil}\\ &\qquad\qquad \mid\ \mathsf{Cons}\ \alpha\ (\mathsf{List}\ \alpha)\end{aligned}$$

In the thesis we denote the type and type constructor in the sans serif font for readability. We use abbreviations: $[\alpha]$ for List α, $[\,]$ for Nil, and $(a : as)$ for (Cons a as).

A join list is constructed by an empty list (Nil), a singleton list with element a (Singleton a), or by concatenating two smaller lists (Concat xs ys). The datatype of a join list whose elements are of type α is defined as follows.

$$\begin{aligned}&\textbf{data}\ \mathsf{List}\ \alpha = \mathsf{Nil}\\ &\qquad\qquad \mid\ \mathsf{Singleton}\ \alpha\\ &\qquad\qquad \mid\ \mathsf{Concat}\ (\mathsf{List}\ \alpha)\ (\mathsf{List}\ \alpha)\end{aligned}$$

We use abbreviations: $[a]$ for Singleton a, and $(xs +\!\!+ ys)$ for (Concat xs ys). We may use either of these definitions unless particularly noted.

List comprehension is a syntax sugar of generation of lists. Let ts be a list, expression $[1..\#ts]$ generates a list of increasing integers from one to the number of elements in ts, and list comprehension $[f\ t_i \mid i \in [1..\#ts]]$ generates a list by applying function f to each element in ts. In this thesis, we denote t_i for the i-th element of list ts, and we use similar notations such as r_i and rs.

We introduce a notation for consumption of lists. Let $\oplus$ be an associative operator with its unit $\iota_\oplus$, then $\sum_\oplus$ denotes the reduction of a list with the operator $\oplus$. Informally, the operation $\sum_\oplus$ is defined as follows.

$$\begin{aligned}\textstyle\sum_\oplus[\,] &= \iota_\oplus\\ \textstyle\sum_\oplus[a_1, a_2, \ldots, a_n] &= a_1 \oplus a_2 \oplus \cdots \oplus a_n\end{aligned}$$

We introduce two functions called *scans* or *prefix-sums*. The scan operation on lists, *scan*, takes an associative operator and a list, and accumulates values with the operator

from left to right. Function $scan'$ is a reversed scan operation. Informally, these two functions are defined as follows.

$$scan\ (\oplus)\ [a_1, a_2, \ldots, a_n] = [\iota_\oplus, a_1, \ldots, a_1 \oplus \cdots \oplus a_{n-1}]$$
$$scan'\ (\oplus)\ [a_1, a_2, \ldots, a_n] = [a_2 \oplus \cdots \oplus a_n, a_3 \oplus \cdots \oplus a_n, \ldots, a_n, \iota_\oplus]$$

Binary Trees

Binary trees are trees whose internal nodes have exactly two children. The datatype of binary trees whose internal nodes have values of type α and whose leaves have values of type β is defined as follows.

$$\textbf{data}\ \mathsf{BTree}\ \alpha\ \beta = \mathsf{BLeaf}\ \alpha \mid \mathsf{BNode}\ (\mathsf{BTree}\ \alpha\ \beta)\ \beta\ (\mathsf{BTree}\ \alpha\ \beta)$$

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.

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.

$$root_\mathrm{b}\ (\mathsf{BLeaf}\ a) = a$$
$$root_\mathrm{b}\ (\mathsf{BNode}\ \_\ b\ \_) = b$$

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

Rose Trees

Rose trees (a term coined by Meertens [96]) are trees whose internal nodes have an arbitrary number of children. In this thesis we assume all the nodes in a rose tree have values of the same type. The datatype of rose trees whose nodes have the values of type α is defined as follows using lists.

$$\textbf{data}\ \mathsf{RTree}\ \alpha = \mathsf{RNode}\ \alpha\ [\mathsf{RTree}\ \alpha]$$

The first argument of RNode is the value of the node, and the second argument is the list of subtrees. A leaf of a rose tree is represented by the RNode with an empty list.

Similar to binary trees, we introduce two functions for manipulating rose trees. Function $root_\mathrm{r}$ returns the value of the root node, and function $setroot_\mathrm{r}$ replaces the value of the root node with the specified value.

$$root_\mathrm{r}\ (\mathsf{RNode}\ a\ ts) = a$$

$$setroot_\mathrm{r}\ (\mathsf{RNode}\ \_\ ts)\ a' = \mathsf{RNode}\ a'\ ts$$

2.2 Tree Homomorphisms

Once a (recursive) datatype is specified, its manipulations are defined along with the specification of the datatype. Homomorphisms, which are natural manipulations defined along with a datatype, play important roles in the theories of constructive algorithmics.

For binary trees, we can define the following general form of recursive functions called *binary-tree homomorphism* (or *tree homomorphism* for short) [118, 119].

Definition 2.1 (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.

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

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.

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

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

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

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

Many tree computations return trees instead of basic values as seen in the short tour in the introduction. For such computations we define the following two computational patterns called *tree accumulations*. Note that these two tree accumulations are in fact tree homomorphisms as stated later.

Definition 2.2 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.

$$\begin{aligned} h^u\ (\mathsf{BLeaf}\ a) &= \mathsf{BLeaf}\ (k_l\ a) \\ h^u\ (\mathsf{BNode}\ l\ b\ r) &= \mathbf{let}\ l' = h^u\ l \\ &\qquad\qquad\ \ r' = h^u\ r \\ &\quad\ \mathbf{in}\ \ \mathsf{BNode}\ l'\ (k_n\ b\ (root_{\mathrm{b}}\ l')\ (root_{\mathrm{b}}\ r'))\ r' \end{aligned}$$ □

Definition 2.3 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.

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

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.

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

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.

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

2.3 Tree Contraction Algorithms

Tree contraction algorithms are very important parallel algorithms for efficient tree manipulations. The idea of tree contraction algorithms was first introduced by Miller and Reif [98], and then many implementations of tree contraction algorithms have been developed on various kinds of parallel-computing models [2, 8, 35, 41, 78, 95, 113].

In this section, we introduce two important implementations of tree contraction algorithms: the original tree contraction algorithm by Miller and Reif [98], and a cost-optimal algorithm on an EREW PRAM developed by Abrahamson et al. [2].

2.3.1 Miller and Reif's Algorithm

Tree contraction algorithms were originally developed for tree reductions. In the following, we review the basic idea of Miller and Reif's tree contraction algorithm by showing how we reduce a tree into one node by parallel removals of nodes. Note that Miller and Reif's algorithm accepts trees of any shape not limited to binary trees.

A naive parallel algorithm for tree reductions is to remove all leaves in parallel in a bottom-up manner. This parallel algorithm, of course, works well for balanced trees, but it runs as slow as a sequential one for completely ill-balanced trees because it takes computational cost depending on the height of the input tree.

The inefficiency of the naive parallel algorithm is caused by a long path from the root to a leaf, which the naive algorithm reduce sequentially. To reduce such a long path in parallel, we can adopt the well-known technique called *pointer jumping*. By using the pointer jumping, we can remove a half number of nodes on the path at one step.

We define two local operations named *rake* and *compress* as in Figure 2.1.

- Rake operation removes a leaf.
- Compress operation applied to an internal node with only one child removes the node and connects its parent and its child.

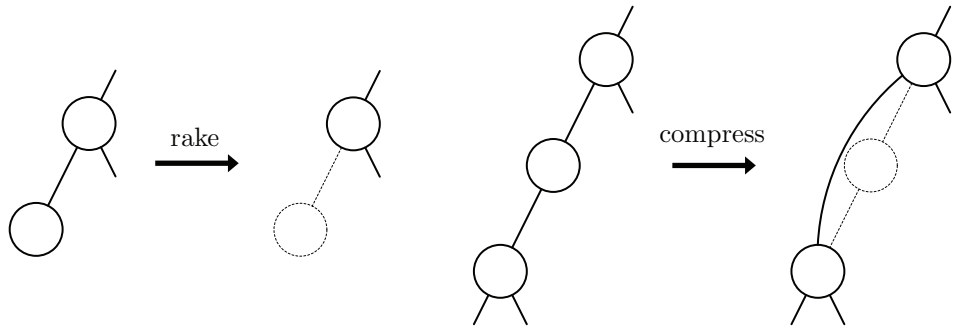


Figure 2.1. Two basic operators in Miller and Reif's tree contraction algorithm.

Miller and Reif's algorithm applies these operations in parallel. Note that the algorithm given below runs on concurrent-read concurrent-write (CRCW) PRAMs.

Algorithm 2.1 (Miller and Reif's Tree Contraction Algorithm)
Input: A tree of any shape (not necessarily to be a binary tree).
Output: The root node.

1. Repeat the following steps (a) and (b) until only one node remains.

   (a) Apply the rake operation in parallel to all the leaves.

   (b) Apply the compress operation in parallel to internal nodes that have only one child. To prevent the tree from becoming disconnected, we do not apply the compress operation to the root node nor two adjacent internal nodes simultaneously. □

Theorem 2.1 *Let N be the number of nodes in the input tree. Miller and Reif's tree contraction algorithm computes reductions in* $\mathrm{O}(\log N)$ *parallel steps on CRCW PRAMs with N processors.* □

2.3.2 Abrahamson et al's Algorithm

Based on Miller and Reif's idea, several efficient tree contraction algorithms have been developed. In this section, we review an efficient and practical algorithm on exclusive-read exclusive-write (EREW) PRAMs developed by Abrahamson et al. [2].

In Abrahamson et al.'s algorithm, we assume that the input tree is a binary tree. Under this assumption, application of the rake operation to a leaf make its parent node have only one child, and thus application of the compress operation to the parent node may follow. With this in mind, we define two tree contracting operations called *contractL* and *contractR*[1]. The contractL operation is applied to an internal node whose left child

[1]These two operations are also called *shunt*, and Abrahamson et al.'s tree contraction algorithm is also called shunt contraction.

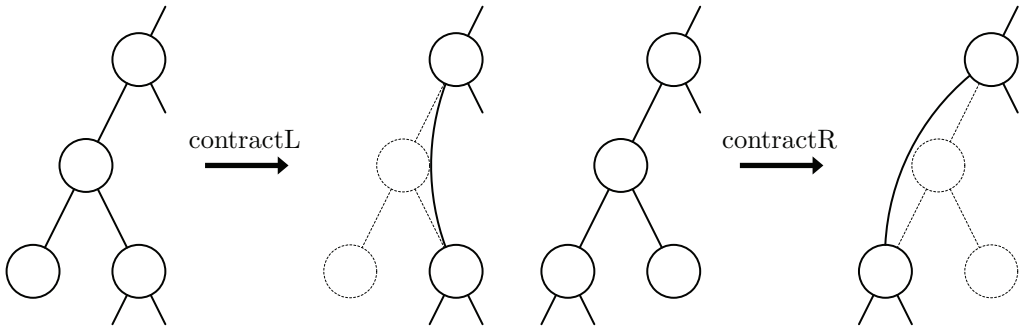


Figure 2.2. Two basic operations in Abrahamson et al.'s tree contraction algorithm.

is a leaf, and removes the node and its left leaf as shown in Figure 2.2. The contractR operation is symmetric to the contractL operation.

To implement tree reductions on EREW PRAMs, we need to apply the two contract operations to disjoint parts in parallel. Abrahamson et al.'s algorithm solves this problem by a novel scheduling based on the prefix numbering of leaves from left to right. This numbering can be computed efficiently in parallel on the Euler-tour of trees which can be constructed by the list-ranking technique [112].

Based on the numbering of leaves, Abrahamson et al.'s tree contraction algorithm applies the two contracting operations in parallel. The algorithm reduces the number of nodes to the half for each step. The detail of the algorithm is as follows.

Algorithm 2.2 (Shunt Contraction Algorithm [2])
Input: A binary tree.
Output: The root node.

1. Number the leaves from left to right starting from 0. This numbering is a prefix-sum computation on the Euler-tour of the tree.
2. Iterate $\lceil \log n \rceil - 1$ times the following steps (a)–(c) where n is the number of nodes in the input tree.
   (a) Apply the contractL operation to each internal node whose left child is an odd-numbered leaf except the root node.
   (b) Apply the contractR operation to each internal node whose right child is an odd-numbered leaf except the root node and nodes involved in the previous step.
   (c) Renumber leaves by dividing their number by 2.
3. Remove two children of the root node. □

Theorem 2.2 ([2]) *Let N be the number of nodes in the input binary tree, and P be the number of processors. The shunt contraction algorithm reduce the tree into a node in $O(N/P + \log P)$ parallel steps on EREW PRAMs.* □

We have shown how we can reduce a tree into its root node in parallel, without mentioning the computation along the contractions. In fact, we require some conditions on the parameter functions to compute tree manipulations based on tree contraction algorithms. In the following, we review the conditions given by Abrahamson et al. [2]. Abrahamson et al. discussed the condition for applying the tree contraction algorithm to the algebraic tree computations. There are two types of algebraic tree computations: the bottom-up algebraic tree computation (B-ATC) and the top-down algebraic tree computation (T-ATC).

Let $\mathcal{S}$ be a set of values, $\mathcal{F}$ be a set of binary functions over $\mathcal{S}$, $\mathcal{G}$ be a set of unary function over $\mathcal{S}$. A tree of B-ATC problem $(\mathcal{S}, \mathcal{F}, \mathcal{G})$ is a binary tree where every leaf has a value in $\mathcal{S}$ and every internal node has a binary function in $\mathcal{F}$, and we want to evaluate the expression associated to the tree. For this B-ATC problem, Abrahamson et al. showed the following condition for the parallel implementation based on tree contraction algorithms [2].

Lemma 2.1 ([2]) *For a given tree of B-ATC problem $(\mathcal{S}, \mathcal{F}, \mathcal{G})$, tree reduction and upwards accumulation on the tree are computed in the same cost as tree contraction algorithms if the following conditions are satisfied.*

- *The set $\mathcal{F}$ is an indexed set whose elements can be evaluated in* $\mathrm{O}(1)$ *sequential time.*
- *The set $\mathcal{G}$ is an indexed set whose elements can be evaluated in* $\mathrm{O}(1)$ *sequential time and includes the identity function.*
- *For all $g_i, g_j \in \mathcal{G}$, $f_m \in \mathcal{F}$, and $a \in \mathcal{S}$, the functions g_s and g_t given by*

$$g_s(x) = g_i(f_m(g_j(x), a)) \quad \text{and} \quad g_t(x) = g_i(f_m(a, g_j(x)))$$

both belong to $\mathcal{G}$ and their indices s and t can be computed in $\mathrm{O}(1)$ *sequential time from i, j, m, and a.* □

Let $\mathcal{S}$ be a set of values, $\mathcal{G}$ be a set of unary function on $\mathcal{S}$, a tree of T-ATC problem $(\mathcal{S}, \mathcal{G})$ is a binary tree where the root has a value in $\mathcal{S}$ and every edge has a function in $\mathcal{G}$. Given a T-ATC tree we want to compute a value for each path from the root to a node by evaluating the functions on the path from the root. For the T-ATC problem, Abrahamson et al. showed the following condition for the parallel implementation.

Lemma 2.2 ([2]) *For a given tree of T-ATC problem $(\mathcal{S}, \mathcal{G})$, downwards accumulation on the tree can be computed in the same cost as the tree contraction algorithm, if the following conditions are satisfied.*

- *The set $\mathcal{G}$ is an indexed set whose element can be evaluated in* $\mathrm{O}(1)$ *sequential time and includes the identity function.*

- *The set $\mathcal{G}$ is closed under composition and for each $g_i, g_j \in \mathcal{G}$ the index of $g_i \circ g_j$ can be computed in* $\mathrm{O}(1)$ *sequential time from i and j.* □

The third item in Lemma 2.1 and the second item in Lemma 2.2 formalize certain closure properties in the set of functions. These closure properties are the most important conditions for the parallel implementation.

As we have reviewed, Abrahamson et al. only showed the conditions for the parallel evaluation of algebraic tree computations. In general, however, we have more complicated tree computation in which the computation on an internal node is defined with its value. We show another condition in the following section for parallel tree skeletons based on the closure property of parameter functions for those tree manipulations.

2.4 Binary-Tree Skeletons

Parallel binary-tree skeletons are basic computational patterns manipulating binary trees in parallel. In this section, we introduce a set of binary-tree skeletons with some discussions on their formalization and parallelization, and then provide three additional skeletons for specialized computations.

2.4.1 Basic Binary-Tree Skeletons

A set of binary-tree skeletons, first proposed by Skillicorn [119], includes the following five higher-order functions. These skeletons are basic primitives in the parallel computation for trees.

- Two node-wise computations: map_b and $\mathsf{zipwith}_\mathrm{b}$
- Two bottom-up computations: $\mathsf{reduce}_\mathrm{b}$ and uAcc_b (upwards accumulate)
- One top-down computation: dAcc_b (downwards accumulate)

We give the formal denotational definition of them as sequential recursive functions in Figure 2.3. We denote the parallel binary-tree skeletons in the sans-serif font with a subscript b. In the following, we show the intuitive meaning of the basic binary-tree skeletons together with additional conditions for parallel implementation.

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 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 $\mathsf{uAcc_b}$ 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 $\mathsf{reduce_b}$ and $\mathsf{uAcc_b}$ 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.

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

We denote the function k satisfying the condition above as $k = \langle \phi, \psi_n, \psi_l, \psi_r \rangle_{\mathrm{u}}$.

The parallel skeleton $\mathsf{dAcc_b}$ 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 $\mathsf{dAcc_b}$ 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.

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

We denote the pair of functions g_l and g_r satisfying the condition above as $(g_l, g_r) = \langle \phi_l, \phi_r, \psi_u, \psi_d \rangle_{\mathrm{d}}$.

The conditions for the parallel implementation of the $\mathsf{reduce_b}$, $\mathsf{uAcc_b}$, and $\mathsf{dAcc_b}$ skeletons given as auxiliary functions are newly introduced by the author. We discuss the conditions more in details in Chapter 3.

These parallel tree skeletons can be implement efficiently in parallel for many parallel architectures. 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 $\mathsf{reduce_b}$, $\mathsf{uAcc_b}$ and $\mathsf{dAcc_b}$ are satisfied. The $\mathsf{map_b}$ and $\mathsf{zipwith_b}$ skeletons can be implemented just by applying functions independently to each node, and they run in $\mathrm{O}(N/P)$ parallel time. The $\mathsf{reduce_b}$ skeleton can be implemented by tree contraction algorithms and it runs in $\mathrm{O}(N/P + \log P)$ parallel time. The $\mathsf{uAcc_b}$ and $\mathsf{dAcc_b}$ skeletons are generalized computational patterns of the algebraic tree computations [2], and Gibbons et al. [50] developed parallel implementations of them based on tree contraction algorithms. With their implementations, the $\mathsf{uAcc_b}$ and $\mathsf{dAcc_b}$ skeletons can be computed in $\mathrm{O}(N/P + \log P)$ parallel time. In other words, the $\mathsf{map_b}$ and $\mathsf{zipwith_b}$ skeletons achieve linear speedup under $P \leq \mathrm{O}(N)$, and the $\mathsf{reduce_b}$, $\mathsf{uAcc_b}$, and $\mathsf{dAcc_b}$

$$
\begin{aligned}
&\mathsf{map_b} :: (\alpha \to \gamma) \to (\beta \to \delta) \to \mathsf{BTree}\ \alpha\ \beta \to \mathsf{BTree}\ \gamma\ \delta\\
&\mathsf{map_b}\ k_l\ k_n\ (\mathsf{BLeaf}\ n) \quad = \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)\\
&\\
&\mathsf{zipwith_b} :: (\alpha \to \alpha' \to \gamma) \to (\beta \to \beta' \to \delta) \to \mathsf{BTree}\ \alpha\ \beta \to \mathsf{BTree}\ \alpha'\ \beta'\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \to \mathsf{BTree}\ \gamma\ \delta\\
&\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')\\
&\\
&\mathsf{reduce_b} :: (\beta \to \alpha \to \alpha \to \alpha) \to \mathsf{BTree}\ \alpha\ \beta \to \alpha\\
&\mathsf{reduce_b}\ k\ (\mathsf{BLeaf}\ n) \quad = n\\
&\mathsf{reduce_b}\ k\ (\mathsf{BNode}\ l\ n\ r) = k\ (\mathsf{reduce_b}\ k\ l)\ n\ (\mathsf{reduce_b}\ k\ r)\\
&\\
&\mathsf{uAcc_b} :: (\beta \to \alpha \to \alpha \to \alpha) \to \mathsf{BTree}\ \alpha\ \beta \to \mathsf{BTree}\ \alpha\ \alpha\\
&\mathsf{uAcc_b}\ k\ (\mathsf{BLeaf}\ n) \quad = \mathsf{BLeaf}\ n\\
&\mathsf{uAcc_b}\ k\ (\mathsf{BNode}\ l\ n\ r) = \mathbf{let}\ l' = \mathsf{uAcc_b}\ k\ l\\
&\qquad\qquad\qquad\qquad\qquad\quad r' = \mathsf{uAcc_b}\ k\ r\\
&\qquad\qquad\qquad\qquad\quad\ \mathbf{in}\ \ \mathsf{BNode}\ l'\ (k\ (root_b\ l')\ n\ (root_b\ r'))\ r'\\
&\\
&\mathsf{dAcc_b} :: ((\gamma \to \beta \to \gamma), (\gamma \to \beta \to \gamma)) \to \gamma \to \mathsf{BTree}\ \alpha\ \beta \to \mathsf{BTree}\ \gamma\ \gamma\\
&\mathsf{dAcc_b}\ (g_l, g_r)\ c\ (\mathsf{BLeaf}\ n) \quad = \mathsf{BLeaf}\ c\\
&\mathsf{dAcc_b}\ (g_l, g_r)\ c\ (\mathsf{BNode}\ l\ n\ r) = \mathsf{BNode}\ (\mathsf{dAcc_b}\ (g_l, g_r)\ (g_l\ c\ b)\ l)\ c\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\mathsf{dAcc_b}\ (g_l, g_r)\ (g_r\ c\ b)\ r)
\end{aligned}
$$

Figure 2.3. Definition of basic binary-tree skeletons.

$$
\begin{aligned}
&\mathsf{getchl_b} :: \alpha \to \mathsf{BTree}\ \beta\ \beta \to \mathsf{BTree}\ \alpha\ \beta\\
&\mathsf{getchl_b}\ c\ (\mathsf{BLeaf}\ n) \quad = \mathsf{BLeaf}\ c\\
&\mathsf{getchl_b}\ c\ (\mathsf{BNode}\ n\ l\ r) = \mathsf{BNode}\ (root_b\ l)\ (\mathsf{getchl_b}\ c\ l)\ (\mathsf{getchl_b}\ c\ r)\\
&\\
&\mathsf{getchr_b} :: \alpha \to \mathsf{BTree}\ \beta\ \beta \to \mathsf{BTree}\ \alpha\ \beta\\
&\mathsf{getchr_b}\ c\ (\mathsf{BLeaf}\ n) \quad = \mathsf{BLeaf}\ c\\
&\mathsf{getchr_b}\ c\ (\mathsf{BNode}\ n\ l\ r) = \mathsf{BNode}\ (root_b\ r)\ (\mathsf{getchr_b}\ c\ l)\ (\mathsf{getchr_b}\ c\ r)\\
&\\
&\mathsf{getparent_b} :: \beta \to \mathsf{BTree}\ \alpha\ \beta \to \mathsf{BTree}\ \beta\ \beta\\
&\mathsf{getparent_b}\ c\ (\mathsf{BLeaf}\ n) \quad = \mathsf{BLeaf}\ c\\
&\mathsf{getparent_b}\ c\ (\mathsf{BNode}\ n\ l\ r) = \mathsf{BNode}\ c\ (\mathsf{getparent_b}\ n\ l)\ (\mathsf{getparent_b}\ n\ r)
\end{aligned}
$$

Figure 2.4. Definition of three additional communication skeletons.

skeletons achieve linear speedup under $P \leq \mathrm{O}(N/\log N)$. For distributed-memory parallel computers, we will develop an efficient implementation of the parallel tree skeletons, and with our implementation the tree skeletons achieve linear speedup under $P \leq \mathrm{O}(\sqrt{N})$. See Chapter 6 for more details.

From a theoretical view point, parallel tree skeletons can be defined as instances of tree homomorphism. The map_b, $\mathsf{reduce}_\mathrm{b}$, and uAcc_b skeletons can be defined in the form of tree homomorphism as follows.

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

$$\mathsf{reduce}_\mathrm{b}\ k = (\![id, k]\!)_\mathrm{b}$$

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

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

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

The $\mathsf{zipwith}_\mathrm{b}$ skeleton cannot be defined as a tree homomorphism, but we can formalize it with tree anamorphism [97].

There is another formalization of the uAcc_b and dAcc_b skeletons based on the tree homomorphism, which was studied by Gibbons [48] and Skillicorn [118, 119]. In the formalization, the uAcc_b is computed by gathering the subtree for each node followed by the mapping of $\mathsf{reduce}_\mathrm{b}$ to each node. The dAcc_b is formalized in a reversed way where another homomorphism, called path homomorphism, is introduced to formalize paths from the root to nodes. In Chapter 9, we review the formalization of these two tree accumulations and use them to derive parallel programs for XPath queries.

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 [118, 119] do. For algebraic tree computations our conditions cover the same class as those studied by Abrahamson et al. [2], even though they are specified in a different way. We show that our conditions are given in a natural way based on flexible division of binary trees in the following chapter.

2.4.2 Specialized Binary-Tree Skeletons

The five basic parallel skeletons are general computational patterns covering a wide class of tree manipulations. In the following, we introduce three specialized skeletons for local communication. The formal definition of three communication skeletons is given in Figure 2.4. These skeletons enhance the readability and efficiency of skeleton programs.

The parallel skeleton $\mathsf{getchl}_{\mathrm{b}}$ shows an upwards communication on a binary tree where for each node the value of an internal node is given by its left-child's value, and the value of each leaf is given as the argument of the $\mathsf{getchl}_{\mathrm{b}}$ skeleton. The parallel skeleton $\mathsf{getchr}_{\mathrm{b}}$ is symmetric to the $\mathsf{getchl}_{\mathrm{b}}$ skeleton: this skeleton takes right-child's value for each internal node, and assigns the argument to each leaf. In fact, these two skeletons can be expressed with the $\mathsf{map}_{\mathrm{b}}$ and $\mathsf{uAcc}_{\mathrm{b}}$ skeletons; the $\mathsf{getchl}_{\mathrm{b}}$ skeleton is given as follows.

$$
\begin{aligned}
\mathsf{getchl}_{\mathrm{b}}\ c = {} & \mathsf{map}_{\mathrm{b}}\ \mathit{fst}\ \mathit{fst} \circ \mathsf{uAcc}_{\mathrm{b}}\ \langle \phi, \psi_n, \psi_l, \psi_r \rangle_{\mathrm{u}} \circ \mathsf{map}_{\mathrm{b}}\ (\lambda a.(c, a))\ \mathit{id} \\
& \textbf{where}\ \phi\ n = (\mathsf{True}, n, \_) \\
& \qquad \psi_n\ (\_, lc)\ (\mathsf{True}, n, \_)\ (\_, \_) = (lc, n) \\
& \qquad \psi_n\ (\_, \_)\ (\mathsf{False}, n, c)\ (\_, \_) = (c, n) \\
& \qquad \psi_l\ (\_, n', \_)\ (\mathsf{True}, n, \_)\ (\_, \_) = (\mathsf{False}, n, n') \\
& \qquad \psi_l\ (\_, \_, \_)\ (\mathsf{False}, n, c)\ (\_, \_) = (\mathsf{False}, n, c) \\
& \qquad \psi_r\ (\_, lc)\ (\mathsf{True}, n, \_)\ (\_, \_, \_) = (\mathsf{False}, n, lc) \\
& \qquad \psi_r\ (\_, \_)\ (\mathsf{False}, n, c)\ (\_, \_, \_) = (\mathsf{False}, n, c)
\end{aligned}
$$

The parallel skeleton $\mathsf{getparent}_{\mathrm{b}}$ shows a downwards communication on a binary tree. For each node except the root node the value of the node is given by its parent's value, and the value of the root node is given by the argument of the $\mathsf{getparent}_{\mathrm{b}}$ skeleton. The $\mathsf{getparent}_{\mathrm{b}}$ skeleton can be expressed with the $\mathsf{dAcc}_{\mathrm{b}}$ skeleton as follows.

$$
\begin{aligned}
\mathsf{getparent}_{\mathrm{b}}\ c = {} & \mathsf{dAcc}_{\mathrm{b}}\ \langle \mathit{id}, \mathit{id}, \psi_u, \psi_d \rangle_{\mathrm{d}}\ c \\
& \textbf{where}\ \psi_u\ \_\ m = m \\
& \qquad \psi_d\ \_\ n = n
\end{aligned}
$$

One important advantage of the communication skeletons is that they can be implemented more efficiently than composing basic binary-tree skeletons. We can implement the communication skeletons in $\mathrm{O}(N/P)$ parallel time for binary trees of N nodes on EREW PRAMs with P processor, while the $\mathsf{uAcc}_{\mathrm{b}}$ and $\mathsf{dAcc}_{\mathrm{b}}$ skeletons require $\mathrm{O}(N/P + \log P)$ parallel time on the same parallel computation model. This improvement comes from less dependency in the communication, i.e., in the specialized skeletons the dependency is local.

Chapter 3

Tree Associativity and Ternary-Tree Representation

Associativity is one of the most important properties in parallel programming. In parallel programs for lists, the associativity of list concatenation enables us to divide a list into arbitrary smaller sublists, so that the divide-and-conquer approach can be naturally applied. Many studies have addressed themselves on formalization of parallel algorithms based on the associativity of list concatenation [34, 60, 108, 117], and on derivation of parallel programs by finding associative operators from sequentially defined programs for lists [32, 43].

In spite of the success of the associativity on lists, few studies have discussed the associativity in the context of parallel programming for trees. Though tree contraction algorithms, well-known parallel algorithms for efficient tree manipulations, have been studied by many researchers, the previous studies did not clarify the basic theory of the parallelism in tree contraction algorithms.

In this chapter we study a basic property of the parallel programming for binary trees. The main idea in formalizing the parallelism is that the parallelism is provided by flexible divisions of data structures. As we can see easily, a divide-and-conquer program, which divides a binary tree at the root node into two subtrees, may be inefficient if the tree is ill-balanced. We explain this inefficiency by the inflexible division of binary trees at the root node.

We start by observing a flexible division of binary trees in which a tree is divided at an arbitrary node (not only the root node), and then introduce a new *ternary-tree representation* of binary trees. Based on the ternary-tree representation, we formalize a novel property named *tree associativity* as relationship among ternary-tree representations. Tree associativity provides a balanced ternary-tree structure for any ill-balanced binary tree. It is an analogy to the associativity for parallel list computation, and provides sufficient flexibility for arranging local manipulations in parallel.

Preliminary work of this chapter is presented in [81, 82].

We can implement each tree skeleton as a simple recursive computation on this ternary-tree representation. Therefore, we can compute tree manipulations efficiently in parallel on balanced ternary-tree representation of a binary tree. We furthermore develop a deterministic algorithm for generating a balanced ternary tree from a given binary tree.

This chapter is organized as follows. We first review how associativity works for parallel manipulations of lists in Section 3.1. In Section 3.2, we then examine flexible division of binary trees that is formalized as ternary-tree representation, and formalize a new property named tree associativity on the ternary-tree representation. In Section 3.3, we show that tree homomorphisms can be implemented on this ternary-tree representation, where the parallelization condition for tree skeletons is given based on tree associativity. We develop a sequential algorithm for transforming a binary tree into a balanced ternary-tree representation in Section 3.4. Section 3.5 discusses relations with the previous studies, and Section 3.6 summarize this chapter.

3.1 Associativity in Parallel Programming for Lists

Associativity is one of the most important algebraic properties in parallel programming. In particular for lists, associativity of the list concatenation, $+\!\!+$, enables us to divide a list into smaller sublists, which are manipulated with associative operators in parallel. In this section, we review how the associativity of the list concatenation works in the context of parallel programming for lists.

One approach to implementing parallel programs is the divide-and-conquer, in which a list is divided recursively into two smaller lists. For simplicity, let the number of elements of the input list be a power of two, and under this condition we can divide a list into two halves recursively. We can formalize the division of lists as a binary-tree structure generated by function *list2bt* defined as follows. The resulting binary tree is a leaf-labeled tree.

$$\begin{aligned} &\mathit{list2bt}\ [a] &&= \mathsf{BLeaf}\ a \\ &\mathit{list2bt}\ (l +\!\!+ r) &&= \mathsf{BNode}\ (\mathit{list2bt}\ l)\ \_\ (\mathit{list2bt}\ r) \end{aligned}$$

Function *bt2list* that restores the list structure from the binary-tree representation is defined as follows.

$$\begin{aligned} &\mathit{bt2list}\ (\mathsf{BLeaf}\ a) &&= [a] \\ &\mathit{bt2list}\ (\mathsf{BNode}\ l\ \_\ r) &&= \mathit{bt2list}\ l +\!\!+ \mathit{bt2list}\ r \end{aligned}$$

Here equation $\mathit{bt2list} \circ \mathit{list2bt} = \mathit{id}$ holds. Figure 3.1 shows an example of the division of a list.

Computation of divide-and-conquer parallel programs for lists can be performed along the binary-tree representation of lists. For example, let k be a given function and $\oplus$ be an associative operator, a list homomorphism

$$\begin{aligned} &\mathit{hom}\ k\ (\oplus)\ [a] &&= k\ a \\ &\mathit{hom}\ k\ (\oplus)\ (l +\!\!+ r) &&= (\mathit{hom}\ k\ (\oplus)\ l) \oplus (\mathit{hom}\ k\ (\oplus)\ r) \end{aligned}$$

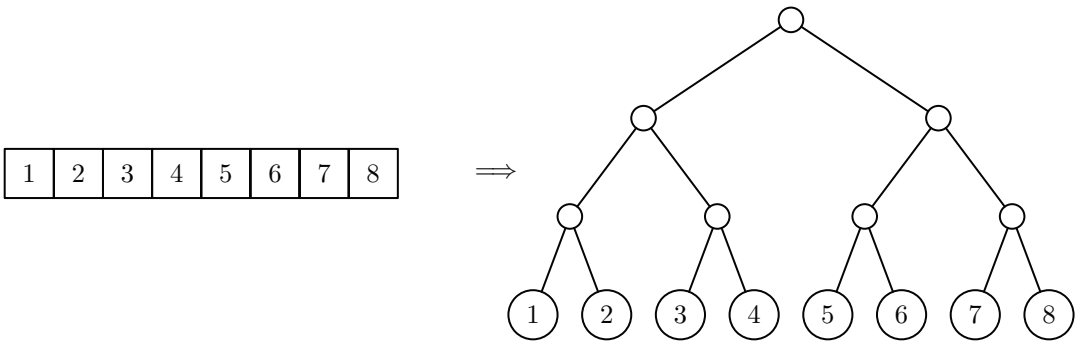


Figure 3.1. Binary-tree representation of dividing a list.

can be implemented as a tree homomorphism on the binary-tree representation as follows.

$$
\begin{aligned}
&\mathit{hom}\ k\ (\oplus) = (\![k_l, k_n]\!)_{\mathrm{b}} \circ \mathit{list2bt} \\
&\qquad \textbf{where}\ k_l\ a \qquad = k\ a \\
&\qquad\qquad k_n\ l\ \_\ r = l \oplus r
\end{aligned}
$$

Since the computations for two subtrees of a node are independent of each other, this naive divide-and-conquer program on the binary-tree representation computes the list homomorphism efficiently in parallel.

In the following, we see a parallel implementation of more involved computation for lists called scans or prefix sums. A well-known parallel implementation of the scan was developed by Kogge and Stone [74], and it consists of two steps. With the binary-tree representation of lists, we can formalize the implementation of the scan as two sweeps on the representation: the first is a bottom-up sweep, and the second is a top-down sweep.

$$
\begin{aligned}
&\mathit{scan}\ (\oplus) = \mathit{bt2list} \circ \mathit{scan}^d\ \iota_\oplus \circ \mathit{fst} \circ \mathit{scan}^u \circ \mathit{list2bt} \\
&\qquad \textbf{where}\ \mathit{scan}^u\ (\mathsf{BLeaf}\ a) \qquad = (\mathsf{BLeaf}\ \_, a) \\
&\qquad\qquad \mathit{scan}^u\ (\mathsf{BNode}\ l\ \_\ r) = \textbf{let}\ (l', lv) = \mathit{scan}^u\ l \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad (r', rv) = \mathit{scan}^u\ r \\
&\qquad\qquad\qquad\qquad\qquad\quad \textbf{in}\ (\mathsf{BNode}\ lv\ l'\ r', lv \oplus rv) \\
&\qquad\qquad \mathit{scan}^d\ c\ (\mathsf{BLeaf}\ \_) \qquad = c \\
&\qquad\qquad \mathit{scan}^d\ c\ (\mathsf{BNode}\ lv\ l\ r) = \mathsf{BNode}\ (\mathit{scan}^d\ c\ l)\ \_\ (\mathit{scan}^d\ (c \oplus lv)\ r)
\end{aligned}
$$

The upwards function scan^u returns two values: an intermediate binary tree and a value passed to the parent. The intuitive definition of the functions scan^u and scan^d are given in Figure 3.2. Note that since both scan^u and scan^d are applied to the two subtrees independently, we can implement a parallel program of the scan in a naive divide-and-conquer style for the binary-tree representation. The parallel algorithm computes the scan operation in logarithmic time to the number of elements of the list because the height of binary-tree structure is logarithmic to the number of elements.

3.2 Tree Associativity on Ternary-Tree Representation

One naive way to divide a binary tree is to divide it at the root node into two subtrees, and based on this division we can compute tree homomorphisms in parallel in a divide-and-conquer way. Ill-balanced tree structures, however, may spoil the parallelism, and in

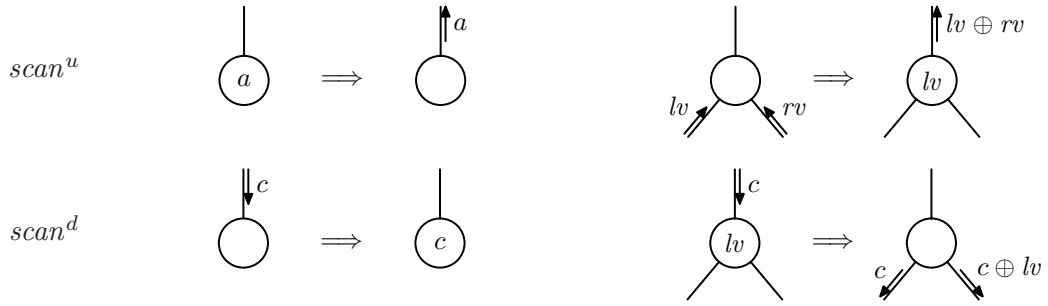


Figure 3.2. Intuitive explanation of functions $scan^u$ and $scan^d$.

the worst case the naive divide-and-conquer programs may run as slow as sequential ones. The problem is due to insufficient freedom in dividing a tree into subtrees.

In this section we discuss another more flexible division of binary trees and formalize the parallelism in parallel tree manipulations. We first propose to represent the division of binary trees as a ternary-tree structure, and then formalize a novel property called *tree associativity* on this ternary-tree representation.

3.2.1 Division of Binary Trees and Ternary-Tree Representation

Consider dividing a binary tree at any node instead of just the root. Let x be a node in a binary tree, we can divide the tree at node x into the following three parts: the left subtree of x, the right subtree of x, and the other nodes including x, as shown in Figure 3.3. We first define two keywords *terminal node* and *segment* to discuss the division of binary trees.

Definition 3.1 (Terminal Node) We call the node at which a binary tree is divided (e.g., x in Figure 3.3) as *terminal node*. □

Definition 3.2 (Segment) We call a set of consecutive nodes that appear in dividing a binary tree as a *segment*. Different from a subtree, a segment may not have all the descendants in the original binary tree. □

Note that all the segments in this thesis form binary trees.

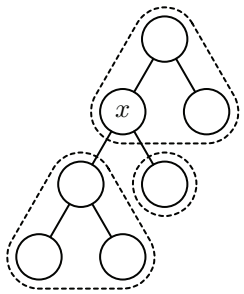


Figure 3.3. Dividing a binary tree into three segments at a terminal node x.

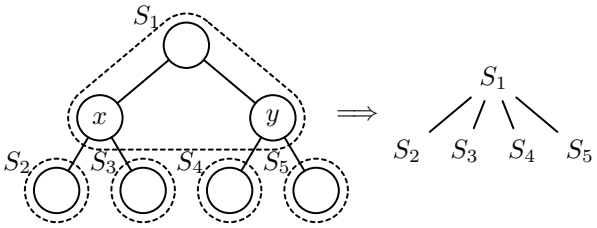


Figure 3.4. When segment S_1 has two terminal nodes x and y, it has four child segments.

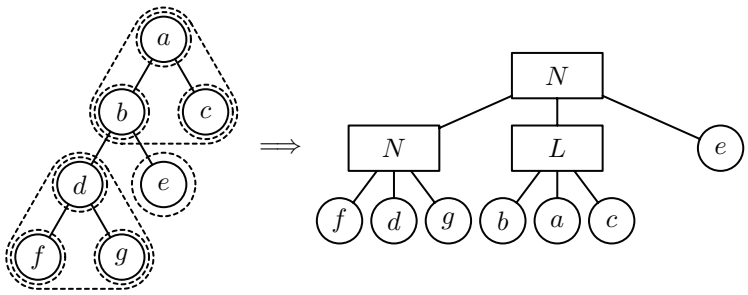


Figure 3.5. Example of ternary-tree representation of a binary tree.

In principle we may divide a binary tree at any internal node, but in practice we should impose some conditions due to the non-linear structure of the tree. As seen in Figure 3.4, when a segment has k terminal nodes it has $2k$ child segments. Such a segment with more than two children makes the handling of the global structure of segments complicated. For the consistent handling of the global structure of segments, the global structure should be kept to be binary through divisions, and therefore we restrict each segment to have at most one terminal node. Under this restriction each segment has zero or two child segments and the global structure of the segments forms a binary tree. Note that we can obtain at least one division satisfying this restriction because dividing a tree at the root node always satisfies the restriction.

We divide a binary tree recursively until each segment consists of one node only. Since division of a binary tree yields three segments, we represent the recursive division of a binary tree as a ternary tree. For each division of a segment, we insert a ternary internal node and put the left-child segment to its left child, the parent segment to its center child, and the right-child segment to its right child, respectively. Figure 3.5 illustrates a ternary-tree representation of a binary tree. A leaf in the ternary-tree representation corresponds to a node in the original binary tree, and a subtree in the ternary-tree representation corresponds to a segment that appears during the recursive division.

The ternary-tree representation must restore the original binary-tree structure. One naive way to achieve this is to store the information of the terminal node for each division, for example, by embedding a pointer to the terminal node in each internal node. This formalization, however, makes it hard to discuss the characteristics of the ternary-tree representation due to the pointers. In this thesis, we examine another specification without

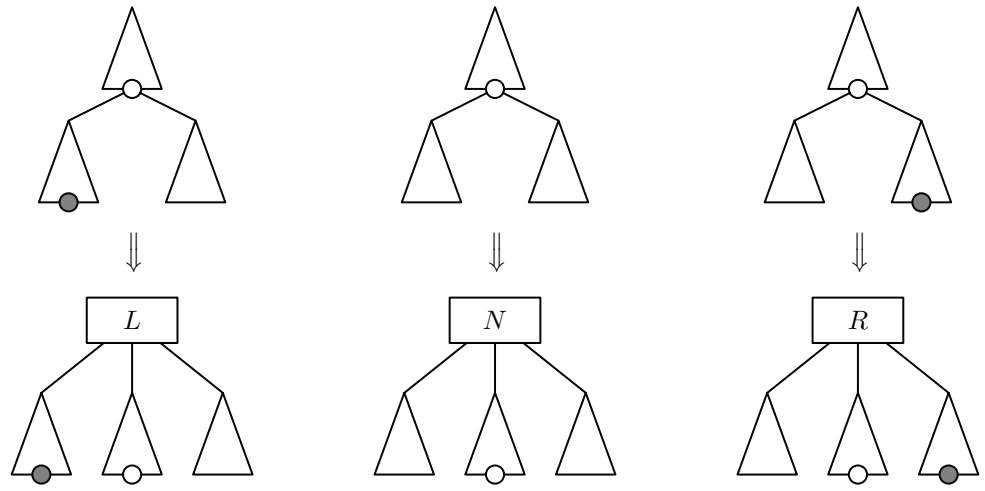


Figure 3.6. Illustrating the labels of the ternary-tree representation. A white circle denotes the terminal node of the current division, and a dark circle denotes the terminal node of the previous division.

pointers where we embed a label into the internal node. We embed one of the following three labels for each internal node. Intuitive meaning of these three labels is given in Figure 3.6.

- TNodeN (in figures, simply N): The subtree whose root node is labeled TNodeN represents a segment with *no* terminal node of the previous division.
- TNodeL (in figures, simply L): The subtree whose root node is labeled TNodeL represents a segment with a terminal node x of the previous division, and x is included in the *left* child segment after dividing the segment.
- TNodeR (in figures, simply R): The subtree whose root node is labeled TNodeR represents a segment with a terminal node x of the previous division, and x is included in the *right* child segment after dividing the segment.

A terminal node corresponding to a previous division must not be included in the parent segment after dividing the segment, since the parent segment always has a terminal node at which the segment is divided. Because of the restriction that a segment has at most one terminal node, the three labels cover all the cases of the ternary-tree representation. We can find the terminal node on the ternary-tree representation by using these labels. For a given internal node, traversing the ternary tree from its center child to the leaves by selecting recursively left/right child at node TNodeL /TNodeR. For example, in Figure 3.5 the global binary tree is divided at node b, which is given on the ternary-tree representation by traversing from the center child of the root node to the left.

The flexible division of binary trees yields a lot of ternary-tree representations for a given binary tree. For example, for the binary tree with seven nodes in Figures 3.3 and 3.5, there are five possible ternary-tree representations as shown in Figure 3.7.

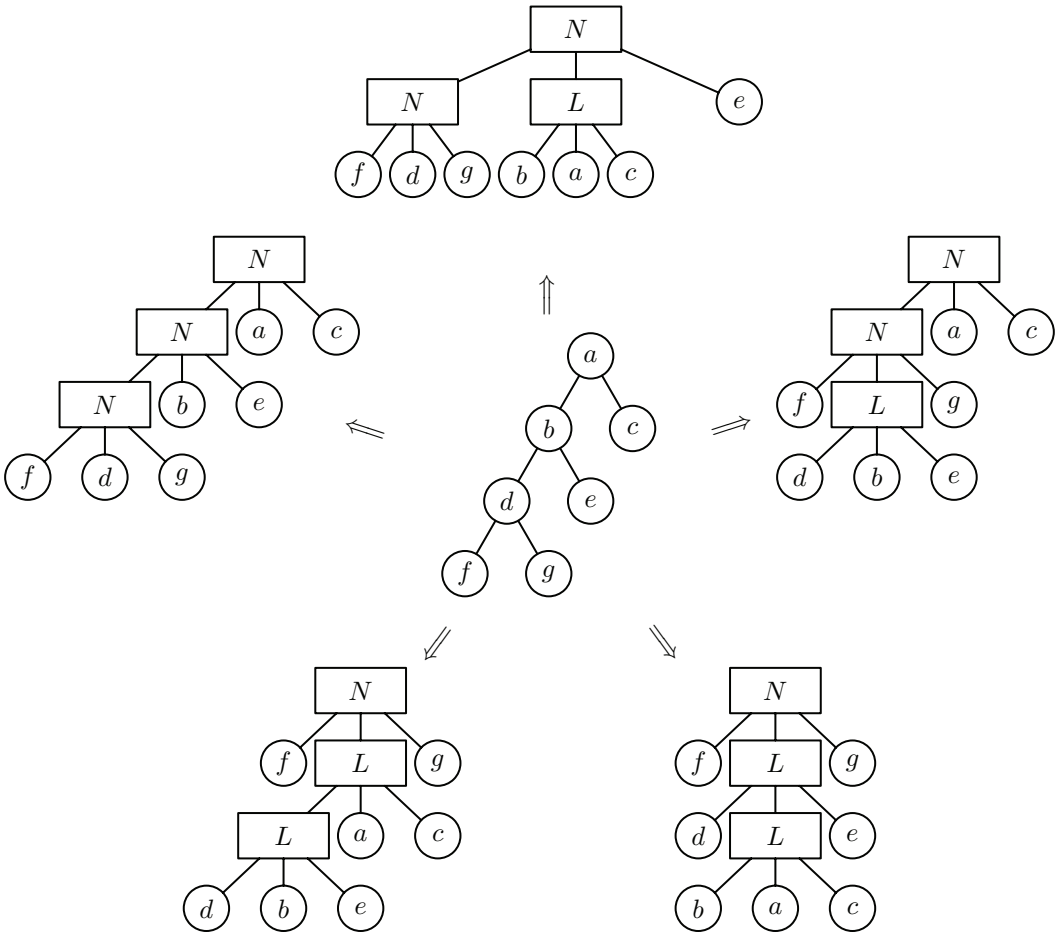


Figure 3.7. All possible ternary-tree representations for a binary tree.

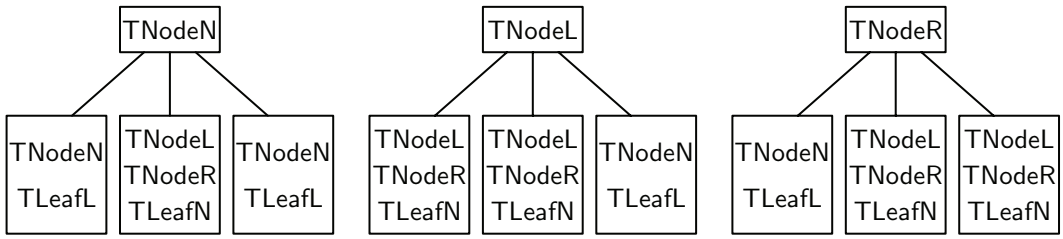


Figure 3.8. The possible structures in ternary-tree representation. One of the constructors is selected for each child.

We define the type of ternary-tree representation for a binary tree of type BTree α β as follows.

$$\begin{aligned}\textbf{data}\ \mathsf{TTree}\ \alpha\ \beta = {} & \mathsf{TLeafL}\ \alpha \\ \mid {} & \mathsf{TLeafN}\ \beta \\ \mid {} & \mathsf{TNodeN}\ (\mathsf{TTree}\ \alpha\ \beta)\ (\mathsf{TTree}\ \alpha\ \beta)\ (\mathsf{TTree}\ \alpha\ \beta) \\ \mid {} & \mathsf{TNodeL}\ (\mathsf{TTree}\ \alpha\ \beta)\ (\mathsf{TTree}\ \alpha\ \beta)\ (\mathsf{TTree}\ \alpha\ \beta) \\ \mid {} & \mathsf{TNodeR}\ (\mathsf{TTree}\ \alpha\ \beta)\ (\mathsf{TTree}\ \alpha\ \beta)\ (\mathsf{TTree}\ \alpha\ \beta)\end{aligned}$$

The first two constructors denote leaves of the ternary-tree representation: TLeafL denotes a leaf that corresponds to a leaf in the original binary tree; TLeafN denotes a leaf that corresponds to an internal node in the original binary tree. The other three constructors correspond to three labels of the internal nodes of the ternary-tree representation.

Not all the ternary trees of the type above represent binary trees. Since the original binary tree has no terminal node before division, the root of a ternary tree should be TNodeN or TLeafL. Since a new terminal node is included in the parent segment for each division, and thus the center child of each internal node should be either TNodeL, TNodeR or TLeafN. For an internal node labeled TNodeN, its left child segment and its right child segment do not have any terminal node, and thus both the left child and the right child should be either TNodeN or TLeafL. For an internal node labeled TNodeL, its left child segment has a terminal node and thus the left child should be either TNodeL, TNodeR or TLeafN, while the right child should be labeled TNodeN or TLeafL. A node labeled TNodeR is symmetric to the node labeled TNodeL. In summary, the possible structures of the ternary trees are given as shown in Figure 3.8.

For a given correct ternary-tree representation, we can restore the original binary tree using the following function *tt2bt*.

$$\begin{aligned}&tt2bt :: \mathsf{TTree}\ \alpha\ \beta \to \mathsf{BTree}\ \alpha\ \beta \\ &tt2bt\ t = tt2bt'\ t\ \_\ \_ \\ \\ &tt2bt'\ (\mathsf{TLeafL}\ a)\ \_\ \_ \quad = \mathsf{BLeaf}\ a \\ &tt2bt'\ (\mathsf{TLeafN}\ b)\ x\ y \quad = \mathsf{BNode}\ b\ x\ y \\ &tt2bt'\ (\mathsf{TNodeN}\ l\ n\ r)\ \_\ \_ = tt2bt'\ n\ (tt2bt'\ l\ \_\ \_)\ (tt2bt'\ r\ \_\ \_) \\ &tt2bt'\ (\mathsf{TNodeL}\ l\ n\ r)\ x\ y\ = tt2bt'\ n\ (tt2bt'\ l\ x\ y)\ (tt2bt'\ r\ \_\ \_) \\ &tt2bt'\ (\mathsf{TNodeR}\ l\ n\ r)\ x\ y\ = tt2bt'\ n\ (tt2bt'\ l\ \_\ \_)\ (tt2bt'\ r\ x\ y)\end{aligned}$$

In the definition above, the second and the third arguments of function $tt2bt'$ are for the left and the right subtrees of the terminal node. The arguments are don't-care values for the nodes TLeafL and TNodeN since the corresponding segments have no terminal node by definition. Since the root node of the ternary-tree representation is either TLeafL or TNodeN, the initial values are also don't-care values.

We will discuss how to obtain balanced ternary-tree representations from given binary trees in Section 3.4.

3.2.2 Tree Associativity

In parallel programming, we change the order of local computations based on the associativity of operators to perform them in parallel. As seen in Section 3.1, associativity of the list concatenation, $+\!\!+$, plays an important role in parallel programming on lists by providing flexible division of lists. We have introduced the ternary-tree representation for the flexible division of binary trees, and now we formalize the tree-version associativity based on the ternary-tree representation.

On the ternary-tree representation, changing the order of local computation corresponds to swapping an internal node with one of its children. As the running example, consider a ternary tree whose root node is TNodeN and its left child is also TNodeN (Figure 3.9, left). Let a, b, c, d, and e denote subtrees, then we can denote such a tree as follows.

$$\mathsf{TNodeN}\ (\mathsf{TNodeN}\ a\ b\ c)\ d\ e$$

Note that two segments corresponding to b and d have a terminal node. This ternary tree represents a binary tree in which the root segment d has two child segments b on the left and e on the right, and the segment b has two child segments a on the left and c on the right (Figure 3.9, center). The ternary tree above can be obtained by the division at the terminal node in d followed by the division at the terminal node in b. In fact, we can swap the order of divisions for this binary tree, that is, we divide the tree at the terminal node in b and then divide the parent segment at the terminal node in d. This division yields the following ternary tree (Figure 3.9, right).

$$\mathsf{TNodeN}\ a\ (\mathsf{TNodeL}\ b\ d\ e)\ c$$

Since the two ternary trees represent the same binary tree, the following equation should hold. We denote $a \equiv_{tt2bt} b$ if two ternary trees a and b represent the same binary tree.

$$\mathsf{TNodeN}\ (\mathsf{TNodeN}\ a\ b\ c)\ d\ e \equiv_{tt2bt} \mathsf{TNodeN}\ a\ (\mathsf{TNodeL}\ b\ d\ e)\ c$$

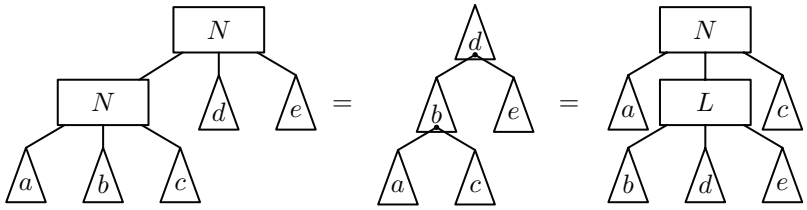


Figure 3.9. Illustration of the equation between two ternary tree representations: TNodeN (TNodeN a b c) d e $\equiv_{tt2bt}$ TNodeN a (TNodeL b d e) c.

By examining the possible structures shown in Figure 3.8 in the same way, we obtain the following five equations.

$$\begin{aligned}
&\mathsf{TNodeN}\ a\ b\ (\mathsf{TNodeN}\ c\ d\ e) \equiv_{tt2bt} \mathsf{TNodeN}\ c\ (\mathsf{TNodeR}\ a\ b\ d)\ e\\
&\mathsf{TNodeL}\ (\mathsf{TNodeL}\ a\ b\ c)\ d\ e \equiv_{tt2bt} \mathsf{TNodeL}\ a\ (\mathsf{TNodeL}\ b\ d\ e)\ c\\
&\mathsf{TNodeR}\ a\ b\ (\mathsf{TNodeL}\ c\ d\ e) \equiv_{tt2bt} \mathsf{TNodeL}\ c\ (\mathsf{TNodeR}\ a\ b\ d)\ e\\
&\mathsf{TNodeL}\ (\mathsf{TNodeR}\ a\ b\ c)\ d\ e \equiv_{tt2bt} \mathsf{TNodeR}\ a\ (\mathsf{TNodeL}\ b\ d\ e)\ c\\
&\mathsf{TNodeR}\ a\ b\ (\mathsf{TNodeR}\ c\ d\ e) \equiv_{tt2bt} \mathsf{TNodeR}\ c\ (\mathsf{TNodeR}\ a\ b\ d)\ e
\end{aligned}$$

We have in total six equations, which are illustrated in Figure 3.10. Note that, we do not have equations for two forms TNodeL a b (TNodeN c d e) and TNodeR (TNodeN a b c) d e due to the restriction that a segment must not have more than one terminal node.

The six equations represent local transformations of ternary trees. We confirm that the local transformations have enough expressiveness by showing that for given two ternary trees representing the same binary tree we can transform one to another using the six equations above. Before discussing the local transformations in more details, we define a special form of the ternary-tree representation.

Definition 3.3 (Plain Ternary-Tree Representation) A ternary tree is said to be plain if it consists only of the constructors TLeafL , TLeafN, and TNodeN. □

In other words, a plain ternary tree is a ternary tree without constructors TNodeL and TNodeR.

Firstly, we show the one-to-one correspondence between binary trees and plain ternary trees.

Lemma 3.1 *For a given binary tree, there is exactly one plain ternary tree that represents the binary tree.*

Proof. As stated in Section 3.2.1, the center child of an internal node must be labeled as either TNodeL , TNodeR, or TLeafN, since the corresponding segment must have a terminal node. Therefore, the center child of a plain ternary tree is TLeafN, which represents the division at the root node. Since the root node is unique in a tree, there is at most one plain ternary tree representing a binary tree.

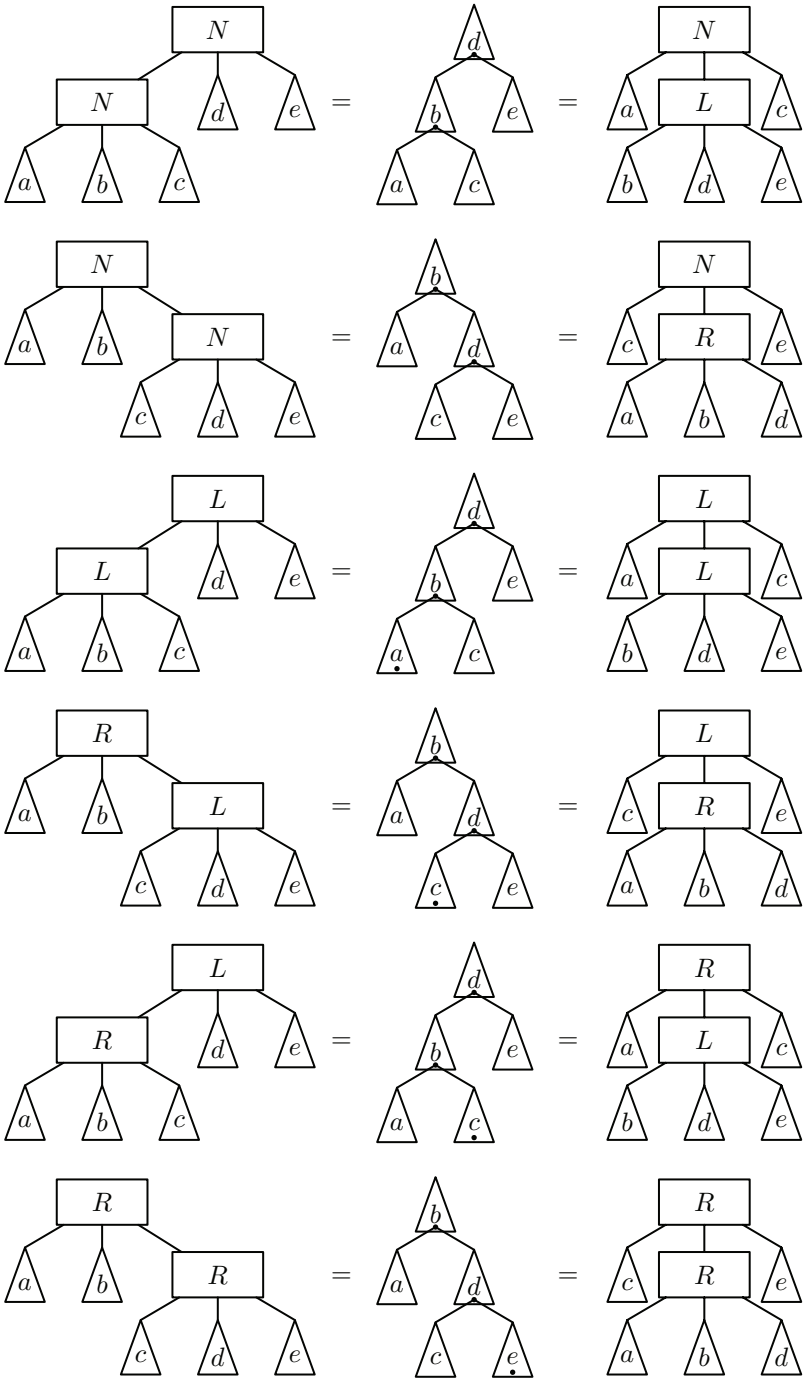


Figure 3.10. Illustration of the six equations of local transformations. A dot in a subtree represents the terminal node.

Next, we show that for any binary tree there is a corresponding plain ternary tree. We can define function *bt2plain* that derives the plain ternary-tree representation from a binary tree. The function *bt2plain* recursively divides a binary tree at the root node.

$$\begin{aligned} &\mathit{bt2plain}\ (\mathsf{BLeaf}\ a) &&= \mathsf{TLeafL}\ a \\ &\mathit{bt2plain}\ (\mathsf{BNode}\ l\ b\ r) &&= \mathsf{TNodeN}\ (\mathit{bt2plain}\ l)\ (\mathsf{TLeafN}\ b)\ (\mathit{bt2plain}\ r) \end{aligned}$$

This function returns a plain ternary tree since there are only three constructors TLeafL, TNodeN, and TLeafN in the function body. □

Secondly, we prove that we can transform a valid ternary tree into the plain ternary tree that represents the same binary tree by applying the local transformations.

Lemma 3.2 *A valid ternary tree can be transformed into a plain ternary tree by the following two equations.*

$$\mathsf{TNodeN}\ (\mathsf{TNodeN}\ a\ b\ c)\ d\ e \equiv_{tt2bt} \mathsf{TNodeN}\ a\ (\mathsf{TNodeL}\ b\ d\ e)\ c$$

$$\mathsf{TNodeN}\ a\ b\ (\mathsf{TNodeN}\ c\ d\ e) \equiv_{tt2bt} \mathsf{TNodeN}\ c\ (\mathsf{TNodeR}\ a\ b\ d)\ e$$

Proof. A single top-down algorithm with the following operations achieves the transformation to the plain ternary trees.

(a) If the root node is a leaf, do nothing.

(b) If the center child of the root node is a leaf, apply the operations to the left and the right children of the root node.

(c) If the center child of the root node is labeled as TNodeL, apply the first equation from right to left, and then apply the operations to the new root node again.

(d) If the center child of the root node is labeled as TNodeR, apply the second equation from right to left, and then apply the operations to the new root node again.

Termination of the algorithm can be proved by decrease of the numbers of TNodeL and TNodeR, and the size of the ternary tree. By the operations (c) and (d), the numbers of TNodeL and TNodeR decrease by one, respectively. The operation (b) does not reduce the numbers of TNodeL and TNodeR, but it reduces the size of the ternary tree. The correctness of the algorithm follows from the correctness of the two equations. □

Given two ternary trees representing the same binary tree, we can transform one to the other as the following lemma states.

Lemma 3.3 *Given two ternary trees representing the same binary tree, one tree can be transformed into the other by the two equations in Lemma 3.2.*

Proof. By Lemma 3.1 the corresponding binary tree has a unique plain ternary tree. By Lemma 3.2, two ternary trees can be transformed into the plain ternary tree by the two equations. Note that the local transformations of the two equations are reversible. It follows from these fact that the lemma holds. □

The Lemmas 3.2 and 3.3 also point out that the first two equations suffice for transforming ternary trees. In fact, among the six equations on the ternary-tree representation, the latter four equations can be derived from the former two equations. Generalizing TNodeN, TNodeL, and TNodeR to three functions g_n, g_l, g_r, we obtain the following lemma.

Lemma 3.4 *Let g_n be a function satisfying the following proposition,*

$$\forall x, z : g_n\ x\ y\ z = g_n\ x\ y'\ z \Longrightarrow y = y'$$

and g_l and g_r be functions satisfying the following two equations for any values a, b, c, d, and e.

$$g_n\ (g_n\ a\ b\ c)\ d\ e = g_n\ a\ (g_l\ b\ d\ e)\ c$$
$$g_n\ a\ b\ (g_n\ c\ d\ e) = g_n\ c\ (g_r\ a\ b\ d)\ e$$

Then, the following four equations hold.

$$g_l\ (g_l\ a\ b\ c)\ d\ e = g_l\ a\ (g_l\ b\ d\ e)\ c$$
$$g_r\ a\ b\ (g_l\ c\ d\ e) = g_l\ c\ (g_r\ a\ b\ d)\ e$$
$$g_l\ (g_r\ a\ b\ c)\ d\ e = g_r\ a\ (g_l\ b\ d\ e)\ c$$
$$g_r\ a\ b\ (g_r\ c\ d\ e) = g_r\ c\ (g_r\ a\ b\ d)\ e$$

Proof. We only show the proof for the first equation of interest. We prove the equation by transforming expression $g_n\ x\ (g_l\ (g_l\ a\ b\ c)\ d\ e)\ y$, in which x and y are arbitrary values, and the second argument of the function g_n is the left-hand side of the equation.

$$\begin{aligned}
& g_n\ x\ (g_l\ (g_l\ a\ b\ c)\ d\ e)\ y \\
= \quad & \{\text{first equation in assumption from right to left}\} \\
& g_n\ (g_n\ x\ (g_l\ a\ b\ c)\ y)\ d\ e \\
= \quad & \{\text{first equation in assumption from right to left applied to the second call of } g_n\} \\
& g_n\ (g_n\ (g_n\ x\ a\ y)\ b\ c)\ d\ e \\
= \quad & \{\text{first equation in assumption from left to right applied to the first call of } g_n\} \\
& g_n\ (g_n\ x\ a\ y)\ (g_l\ b\ d\ e)\ c \\
= \quad & \{\text{first equation in assumption from left to right}\} \\
& g_n\ x\ (g_l\ a\ (g_l\ b\ d\ e)\ c)\ y
\end{aligned}$$

The equation above holds for any values x and y, and thus the equation for the second arguments

$$g_l\ (g_l\ a\ b\ c)\ d\ e = g_l\ a\ (g_l\ b\ d\ e)\ c$$

also holds. This is the first equation of interest.

The other three equations can be proved in the same manner. □

This lemma states that the former two equations in six equations are essential in the transformation among ternary-tree representations. We therefore define the tree-version associativity with the two equations as follows.

Definition 3.4 (Tree Associativity) Three functions g_n, g_l, g_r are called *tree associative*, if the following two equations hold for any a, b, c, d, and e.

$$g_n\ (g_n\ a\ b\ c)\ d\ e = g_n\ a\ (g_l\ b\ d\ e)\ c$$
$$g_n\ a\ b\ (g_n\ c\ d\ e) = g_n\ c\ (g_r\ a\ b\ d)\ e$$ □

Lemma 3.5 *The three constructors of the ternary-tree representation,* TNodeN, TNodeL *and* TNodeR, *are tree associative modulo function tt2bt.*

Proof. The constructors satisfy the following two equations by definition.

$$\mathsf{TNodeN}\ (\mathsf{TNodeN}\ a\ b\ c)\ d\ e \equiv_{tt2bt} \mathsf{TNodeN}\ a\ (\mathsf{TNodeL}\ b\ d\ e)\ c$$
$$\mathsf{TNodeN}\ a\ b\ (\mathsf{TNodeN}\ c\ d\ e) \equiv_{tt2bt} \mathsf{TNodeN}\ c\ (\mathsf{TNodeR}\ a\ b\ d)\ e$$

It follows from the equations above that the lemma holds.

3.3 Implementation of Tree Homomorphisms on Ternary-Tree Representation

In this section we develop an implementation of tree homomorphisms on the ternary-tree representation. First, we specify conditions for the implementation of tree reductions on the ternary-tree representation, where the tree associativity plays an important role. We then develop efficient implementations of tree accumulations. The implementations are very similar to that of *scan* on the binary-tree representation of lists.

3.3.1 Conditions for Implementing Tree Homomorphisms

First, we define a natural computational pattern on the ternary-tree representation named *ternary-tree homomorphism*.

Definition 3.5 (Ternary-Tree Homomorphism) Let k'_l and k'_n be given functions, and g'_n, g'_l, g'_r be tree associative functions. Function h' is called *ternary-tree homomorphism*, if it is defined on ternary trees as follows.

$$\begin{aligned}
h'\ (\mathsf{TLeafL}\ a) &= k'_l\ a \\
h'\ (\mathsf{TLeafN}\ b) &= k'_n\ b \\
h'\ (\mathsf{TNodeN}\ l\ n\ r) &= g'_n\ (h'\ l)\ (h'\ n)\ (h'\ r) \\
h'\ (\mathsf{TNodeL}\ l\ n\ r) &= g'_l\ (h'\ l)\ (h'\ n)\ (h'\ r) \\
h'\ (\mathsf{TNodeR}\ l\ n\ r) &= g'_r\ (h'\ l)\ (h'\ n)\ (h'\ r)
\end{aligned}$$

We may denote a ternary-tree homomorphism as $h' = (\![k'_l, k'_n, g'_n, g'_l, g'_r]\!)_{\mathrm{t}}$. □

As we have seen in the previous section, the ternary-tree representation provides great flexibility in terms of the order of local computations, and the flexibility supports parallel computation on the ternary-tree representation. However, the flexibility of the ternary-tree representation imposes some conditions on the implementation of tree homomorphisms.

In the following, we specify the conditions for implementing tree homomorphism $(\![k_l, k_n]\!)_{\mathrm{b}}$ by ternary-tree homomorphism $(\![k'_l, k'_n, g'_n, g'_l, g'_r]\!)_{\mathrm{t}}$.

First, the ternary-tree homomorphism simulates the tree homomorphism on the plain ternary trees. We can formalize this condition by induction on the structure of binary trees. For the base case, i.e., $\mathsf{BLeaf}\ a$, the results of the tree homomorphism and the ternary-tree homomorphism are given as follows.

$$(\![k_l, k_n]\!)_{\mathrm{b}}\ (\mathsf{BLeaf}\ a) = k_l\ a$$

$$\begin{aligned}(\![k'_l, k'_n, g'_n, g'_l, g'_r]\!)_{\mathrm{t}}\ (bt2plain\ (\mathsf{BLeaf}\ a)) &= (\![k'_l, k'_n, g'_n, g'_l, g'_r]\!)_{\mathrm{t}}\ (\mathsf{TLeafL}\ a)\\ &= k'_l\ a\end{aligned}$$

From these calculations, we have an equation $k_l\ a = k'_l\ a$. For inductive step, i.e., $\mathsf{BNode}\ l\ b\ r$, the results are given as follows.

$$(\![k_l, k_n]\!)_{\mathrm{b}}\ (\mathsf{BNode}\ l\ b\ r) = k_n\ ((\![k_l, k_n]\!)_{\mathrm{b}}\ l)\ b\ ((\![k_l, k_n]\!)_{\mathrm{b}}\ r)$$

$$\begin{aligned}&(\![k'_l, k'_n, g'_n, g'_l, g'_r]\!)_{\mathrm{t}}\ (bt2plain\ (\mathsf{BNode}\ l\ b\ r))\\ &\quad = (\![k'_l, k'_n, g'_n, g'_l, g'_r]\!)_{\mathrm{t}}\ (\mathsf{TNodeN}\ (bt2plain\ l)\ (\mathsf{TLeafN}\ b)\ (bt2plain\ r))\\ &\quad = g'_n\ ((\![k'_l, k'_n, g'_n, g'_l, g'_r]\!)_{\mathrm{t}}\ (bt2plain\ l))\ (k'_n\ b)\ ((\![k'_l, k'_n, g'_n, g'_l, g'_r]\!)_{\mathrm{t}}\ (bt2plain\ r))\end{aligned}$$

With the induction hypothesis

$$(\![k_l, k_n]\!)_{\mathrm{b}}\ x = (\![k'_l, k'_n, g'_n, g'_l, g'_r]\!)_{\mathrm{t}}\ (bt2plain\ x)$$

for $x = l$ and $x = r$, we obtain the following equation

$$k_n\ l'\ b\ r' = g'_n\ l'\ (k'_n\ b)\ r'$$

where l' and r' denote the results of homomorphisms. Note that l' and r' may have any value in the range of the tree homomorphism. The second condition is the tree associativity on three functions g'_n, g'_l and g'_r that is necessary by definition.

Note that these two conditions also form a sufficient condition for the implementation of the tree homomorphism by the ternary-tree homomorphism. By Lemma 3.2, computation on any ternary-tree representation is equivalent to that on the plain ternary tree representing the same binary-tree if the functions are tree associative. The induction above guarantees the correctness of the ternary-tree homomorphism on the plain ternary trees.

We summarize the discussion as the following lemma.

Lemma 3.6 *The necessary and sufficient condition for implementing tree homomorphism* $(\![k_l, k_n]\!)_{\mathrm{b}}$ *by ternary-tree homomorphism* $(\![k'_l, k'_n, g'_n, g'_l, g'_r]\!)_{\mathrm{t}}$ *is that the functions satisfy the following three conditions:*

- *for any* a, $k'_l\ a = k_l\ a$ *holds;*
- *for any* l *and* r *in the range of the tree homomorphism, and for any* b, $g'_n\ l\ (k'_n\ b)\ r = k_n\ l\ b\ r$ *holds;*

- g'_n, g'_l, *and* g'_r *are tree associative.*

Proof. It follows from the discussion above that the lemma holds. □

Based on this lemma, we can specify the condition of the implementation of the $\mathsf{reduce}_\mathrm{b}$ skeleton by ternary-tree homomorphisms. Note that the following condition is just what we have given in Section 2.4.

Corollary 3.1 *The* $\mathsf{reduce}_\mathrm{b}$ *skeleton called with parameter function* k, $\mathsf{reduce}_\mathrm{b}\ k$, *can be implemented by the ternary-tree homomorphism* $(\![id, \phi, \psi_n, \psi_l, \psi_r]\!)_\mathrm{t}$, *if there exist four functions* ϕ, ψ_n, ψ_l *and* ψ_r *satisfying the following equations.*

$$\begin{aligned}
&\psi_n\ l\ (\phi\ b)\ r = k\ l\ b\ r \\
&\psi_n\ (\psi_n\ a\ b\ c)\ d\ e = \psi_n\ a\ (\psi_l\ b\ d\ e)\ c \\
&\psi_n\ a\ b\ (\psi_n\ c\ d\ e) = \psi_n\ c\ (\psi_r\ a\ b\ d)\ e
\end{aligned}$$

Proof. We obtain this corollary from the fact that the $\mathsf{reduce}_\mathrm{b}$ skeleton is a homomorphism, $\mathsf{reduce}_\mathrm{b}\ k = (\![id, k]\!)_\mathrm{b}$, and Lemma 3.6. □

It may be surprising that any given tree homomorphism can be written as a ternary-tree homomorphism unless we care about the efficiency of the implementation. The idea is to introduce functions as the results of local computation. Recall that a subtree of a ternary tree represents a segment and a segment with a terminal node has two child segments. For such a segment with two child segments, which is labeled as either TLeafN, TNodeL, or TNodeR, we generate a binary function that takes two values from the child segments. For readability, we denote functions as f_x where subscript x may denote certain parameter of the function.

Lemma 3.7 *A tree homomorphism* $(\![k_l, k_n]\!)_\mathrm{b}$ *can be implemented by a ternary-tree homomorphism* $(\![k'_l, k'_n, g'_n, g'_l, g'_r]\!)_\mathrm{t}$ *with the parameter functions defined as follows.*

$$\begin{aligned}
k'_l\ a &= k_l\ a \\
k'_n\ b &= \lambda x\ y.k_n\ x\ b\ y \\
g'_n\ l\ f_n\ r &= f_n\ l\ r \\
g'_l\ f_l\ f_n\ r &= \lambda x\ y.f_n\ (f_l\ x\ y)\ r \\
g'_r\ l\ f_n\ f_r &= \lambda x\ y.f_n\ l\ (f_r\ x\ y)
\end{aligned}$$

Proof. We can prove this lemma by checking the equations in Lemma 3.6. The first equation holds by the definition of k'_l. The second equation holds as the following calculation shows.

$$\begin{aligned}
g'_n\ l\ (k'_n\ b)\ r = &\quad \{\text{definition of } k'_n\} \\
&\ g'_n\ l\ (\lambda x\ y.k_n\ x\ b\ y)\ r \\
= &\quad \{\text{definition of } g'_n\} \\
&\ (\lambda x\ y.k_n\ x\ b\ y)\ l\ r \\
= &\quad \{\text{reduction}\} \\
&\ k_n\ l\ b\ r
\end{aligned}$$

Finally, tree associativity on functions g'_n, g'_l, and g'_r can be proved by simple calculation. For example, the following calculations show that equation $g'_n\ (g'_n\ a\ f_b\ c)\ f_d\ e = g'_n\ a\ (g'_l\ f_b\ f_d\ e)\ c$ holds.

$$
\begin{aligned}
\text{LHS} = \quad & \{\text{definition of } g'_n\} \\
& g'_n\ (f_b\ a\ c)\ f_d\ e \\
= \quad & \{\text{definition of } g'_n\} \\
& f_d\ (f_b\ a\ c)\ e
\end{aligned}
$$

$$
\begin{aligned}
\text{RHS} = \quad & \{\text{definition of } g'_l\} \\
& g'_n\ a\ (\lambda x\ y.f_d\ (f_b\ x\ y)\ e)\ c \\
= \quad & \{\text{definition of } g'_n\} \\
& (\lambda x\ y.f_d\ (f_b\ x\ y)\ e)\ a\ c \\
= \quad & \{\text{reduction}\} \\
& f_d\ (f_b\ a\ c)\ e
\end{aligned}
$$

We can prove the other equation, $g'_n\ a\ f_b\ (g'_n\ c\ f_d\ e) = g'_n\ c\ (g'_r\ a\ f_b\ f_d)\ e$, easily in the same manner, where both sides are reduced into $f_b\ a\ (f_d\ c\ e)$. □

In general new functions generated by g'_l and g'_r expand in terms of the size of the function body and the computational cost. For efficient implementation of the ternary-tree homomorphism, we attach some requirements on the size of generated functions. One sufficient but a bit strict requirement is given as the closure property, which limits the size of functions to a certain constant. We can find another relaxed requirement named as uniform closure property in the discussion by Miller and Teng [101].

In the following, we demonstrate how to find a set of suitable functions of the ternary-tree homomorphism. A systematic way to derive the set of functions is the *generalization-and-test* approach, which has been studied for the derivation of parallel programs for lists [31,43]. In this approach, we start at a functional form given by templatization of the function for internal nodes. We then test whether it is closed under generating functions or generalize the functional form until the form is closed.

We now show the derivation of an efficient ternary-tree homomorphism using the tree homomorphism $height_{\mathrm{b}}$ in Section 2.2 as an example. The function $height_{\mathrm{b}}$ is a tree homomorphism $(\!|height_l, height_n|\!)_{\mathrm{b}}$ where the function $height_n$ is defined as follows.

$$height_n\ l\ b\ r = 1 + (l \uparrow r)$$

For the first step, we abstract the constant value in the function $height_n$ to obtain the following form

$$f_x\ l\ r = x + (l \uparrow r)$$

where x denotes a value introduced by the templatization of the function $height_n$. Then, we simulate the generation of functions by g'_l and g'_r using instances of the form. By

substituting instances for the arguments of g'_l, we obtain a new function as follows.

$$\begin{aligned}
g'_l\ f_l\ f_n\ r = &\quad \{\text{substituting instances}\}\\
&\lambda x\ y.(\lambda x'\ y'.l + (x' \uparrow y'))\ ((\lambda x'\ y'.n + (x'' \uparrow y''))\ x\ y)\ r\\
= &\quad \{\text{reduction}\}\\
&\lambda x\ y.(\lambda x'\ y'.l + (x' \uparrow y'))\ (x \uparrow y)\ r\\
= &\quad \{\text{reduction}\}\\
&\lambda x\ y.l + ((x \uparrow y) \uparrow r)\\
= &\quad \{\text{commutativity of } \uparrow \text{ and distributivity of } + \text{ over } \uparrow\}\\
&\lambda x\ y.(l + r) \uparrow (l + (x \uparrow y))
\end{aligned}$$

Unfortunately, the function generated by g'_l is not in the original form. Therefore, we again abstract the function to the following form of functions. Note that the new functional form is a generalized one of the original form.

$$f_{(a,b)} = \lambda x\ y.a \uparrow (b + (x \uparrow y))$$

In this case, we can prove that the form is closed under generating functions by g'_l and g'_r, as the following instantiations and calculations show.

$$\begin{aligned}
&g'_l\ f_{(a_l,b_l)}\ f_{(a_n,b_n)}\ r\\
&\quad = \quad \{\text{substituting instances}\}\\
&\qquad \lambda x\ y.f_{(a_n,b_n)}\ (f_{(a_l,b_l)}\ x\ y)\ r\\
&\quad = \quad \{\text{unfolding two functions } f_{(a_n,b_n)} \text{ and } f_{(a_l,b_l)} \text{ with arithmetic rules}\}\\
&\qquad \lambda x\ y.a_n \uparrow (b_n + a_l) \uparrow (b_n + r) \uparrow (b_n + b_l + (x \uparrow y))\\
&\quad = \quad \{\text{folding to the functional form}\}\\
&\qquad \lambda x\ y.f_{(a_n \uparrow (b_n + a_l) \uparrow (b_n + r),\, b_n + b_l)}\ x\ y
\end{aligned}$$

$$\begin{aligned}
&g'_r\ l\ f_{(a_n,b_n)}\ f_{(a_r,b_r)}\\
&\quad = \quad \{\text{substituting instances}\}\\
&\qquad \lambda x\ y.f_{(a_n,b_n)}\ l\ (f_{(a_r,b_r)}\ x\ y)\\
&\quad = \quad \{\text{unfolding two functions } f_{(a_n,b_n)} \text{ and } f_{(a_r,b_r)} \text{ with arithmetic rules}\}\\
&\qquad \lambda x\ y.a_n \uparrow (b_n + l) \uparrow (b_n + a_r) \uparrow (b_n + b_r + (x \uparrow y))\\
&\quad = \quad \{\text{folding to the functional form}\}\\
&\qquad \lambda x\ y.f_{(a_n \uparrow (b_n + l) \uparrow (b_n + a_r),\, b_n + b_r)}\ x\ y
\end{aligned}$$

Based on these calculations, we can use the functions g'_l and g'_r for implementing ternary-tree homomorphism. Noting that the functional form is preserved through the computation of the ternary-tree homomorphism, we can simplify the definition a bit. By substituting pair (a, b) for function $f_{(a,b)}$, and with Lemma 3.7, we have the following ternary-tree homomorphism $(\!|k_l, k_n, g_n, g_l, g_r|\!)_{\mathrm{t}}$ for the tree homomorphism $height_{\mathrm{b}}$, where the five functions are defined as follows.

$$\begin{aligned}
k'_l\ a &= 1\\
k'_n\ b &= (-\infty,\ 1)\\
g'_n\ l\ (a_n, b_n)\ r &= a_n \uparrow (b_n + (l \uparrow r))\\
g'_l\ (a_l, b_l)\ (a_n, b_n)\ r &= (a_n \uparrow (b_n + a_l) \uparrow (b_n + r),\ b_n + b_l)\\
g'_r\ l\ (a_n, b_n)\ (a_r, b_r) &= (a_n \uparrow (b_n + l) \uparrow (b_n + a_r),\ b_n + b_r)
\end{aligned}$$

3.3.2 Implementation of Tree Accumulations

As seen in Section 3.1, the parallel implementation of *scan* consists of the bottom-up and the top-down sweeps on the binary-tree representation of lists. In this section, we develop implementations of the two tree accumulations in the same way on the ternary-tree representation of binary trees.

We first extend the data structure of the ternary-tree representation to attach a value to every internal node. We use this extended ternary tree for the intermediate data in the implementation of the tree accumulations. Let the attached value have type γ, we define the datatype of the extended ternary tree as follows.

$$
\begin{aligned}
\textbf{data}\ \mathsf{TTree'}\ \alpha\ \beta\ \gamma = {} & \mathsf{TLeafL'}\ \alpha \\
\mid {} & \mathsf{TLeafN'}\ \beta \\
\mid {} & \mathsf{TNodeN'}\ \gamma\ (\mathsf{TTree'}\ \alpha\ \beta\ \gamma)\ (\mathsf{TTree'}\ \alpha\ \beta\ \gamma)\ (\mathsf{TTree'}\ \alpha\ \beta\ \gamma) \\
\mid {} & \mathsf{TNodeL'}\ \gamma\ (\mathsf{TTree'}\ \alpha\ \beta\ \gamma)\ (\mathsf{TTree'}\ \alpha\ \beta\ \gamma)\ (\mathsf{TTree'}\ \alpha\ \beta\ \gamma) \\
\mid {} & \mathsf{TNodeR'}\ \gamma\ (\mathsf{TTree'}\ \alpha\ \beta\ \gamma)\ (\mathsf{TTree'}\ \alpha\ \beta\ \gamma)\ (\mathsf{TTree'}\ \alpha\ \beta\ \gamma)
\end{aligned}
$$

Implementation of Upwards Accumulation

Since the $\mathsf{uAcc_b}$ skeleton applies the $\mathsf{reduce_b}$ skeleton to each subtree, it requires the argument function to satisfy the same condition as the $\mathsf{reduce_b}$ skeleton for efficient parallel implementations. Let k be a function satisfying the condition for the $\mathsf{uAcc_b}$ skeleton:

$$k = \langle \phi, \psi_n, \psi_l, \psi_r \rangle_{\mathrm{u}}$$

with auxiliary functions ϕ, ψ_n, ψ_l, and ψ_r. Under this condition, we can implement the $\mathsf{uAcc_b}$ skeleton by a bottom-up and a top-down sweeps on the ternary-tree representation.

The bottom-up sweep computes tree reduction and puts the intermediate result on each internal node. Function $\mathsf{uAcc}^u_{\mathrm{b}}$ that performs the bottom-up sweep is defined as follows. In the definition, the return value is a pair of the intermediate tree and a value passed to the parent. Figure 3.11 shows an intuitive illustration of the function $\mathsf{uAcc}^u_{\mathrm{b}}$.

$$
\begin{aligned}
\mathsf{uAcc}^u_{\mathrm{b}}\ \langle \phi, \psi_n, \psi_l, \psi_r \rangle_{\mathrm{u}}\ (\mathsf{TLeafL}\ a) &= (\mathsf{TLeafL'}\ a, a) \\
\mathsf{uAcc}^u_{\mathrm{b}}\ \langle \phi, \psi_n, \psi_l, \psi_r \rangle_{\mathrm{u}}\ (\mathsf{TLeafN}\ b) &= (\mathsf{TLeafN'}\ (\phi\ b), \phi\ b) \\
\mathsf{uAcc}^u_{\mathrm{b}}\ \langle \phi, \psi_n, \psi_l, \psi_r \rangle_{\mathrm{u}}\ (\mathsf{TNodeN}\ l\ n\ r) &= \textbf{let}\ (l', lv) = \mathsf{uAcc}^u_{\mathrm{b}}\ \langle \phi, \psi_n, \psi_l, \psi_r \rangle_{\mathrm{u}}\ l \\
&\qquad\qquad (n', nv) = \mathsf{uAcc}^u_{\mathrm{b}}\ \langle \phi, \psi_n, \psi_l, \psi_r \rangle_{\mathrm{u}}\ n \\
&\qquad\qquad (r', rv) = \mathsf{uAcc}^u_{\mathrm{b}}\ \langle \phi, \psi_n, \psi_l, \psi_r \rangle_{\mathrm{u}}\ r \\
&\qquad \textbf{in}\ (\mathsf{TNodeN'}\ (lv, rv)\ l'\ n'\ r',\ \psi_n\ l\ n\ r) \\
\mathsf{uAcc}^u_{\mathrm{b}}\ \langle \phi, \psi_n, \psi_l, \psi_r \rangle_{\mathrm{u}}\ (\mathsf{TNodeL}\ l\ n\ r) &= \textbf{let}\ (l', lv) = \mathsf{uAcc}^u_{\mathrm{b}}\ \langle \phi, \psi_n, \psi_l, \psi_r \rangle_{\mathrm{u}}\ l \\
&\qquad\qquad (n', nv) = \mathsf{uAcc}^u_{\mathrm{b}}\ \langle \phi, \psi_n, \psi_l, \psi_r \rangle_{\mathrm{u}}\ n \\
&\qquad\qquad (r', rv) = \mathsf{uAcc}^u_{\mathrm{b}}\ \langle \phi, \psi_n, \psi_l, \psi_r \rangle_{\mathrm{u}}\ r \\
&\qquad \textbf{in}\ (\mathsf{TNodeL'}\ (lv, rv)\ l'\ n'\ r',\ \psi_l\ l\ n\ r) \\
\mathsf{uAcc}^u_{\mathrm{b}}\ \langle \phi, \psi_n, \psi_l, \psi_r \rangle_{\mathrm{u}}\ (\mathsf{TNodeR}\ l\ n\ r) &= \textbf{let}\ (l', lv) = \mathsf{uAcc}^u_{\mathrm{b}}\ \langle \phi, \psi_n, \psi_l, \psi_r \rangle_{\mathrm{u}}\ l \\
&\qquad\qquad (n', nv) = \mathsf{uAcc}^u_{\mathrm{b}}\ \langle \phi, \psi_n, \psi_l, \psi_r \rangle_{\mathrm{u}}\ n \\
&\qquad\qquad (r', rv) = \mathsf{uAcc}^u_{\mathrm{b}}\ \langle \phi, \psi_n, \psi_l, \psi_r \rangle_{\mathrm{u}}\ r \\
&\qquad \textbf{in}\ (\mathsf{TNodeR'}\ (lv, rv)\ l'\ n'\ r',\ \psi_r\ l\ n\ r)
\end{aligned}
$$

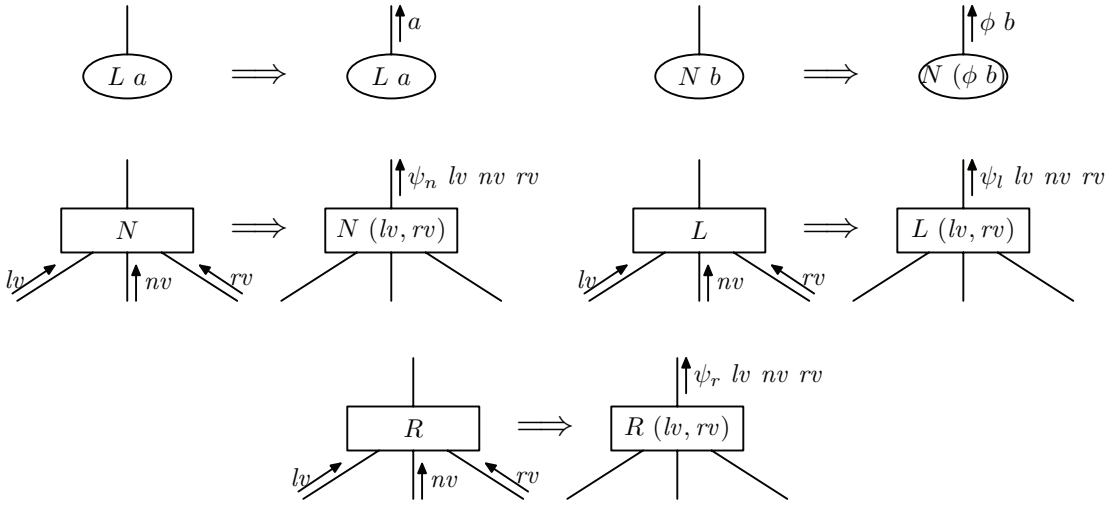


Figure 3.11. Illustration of the function $\mathsf{uAcc}^{u}_{\mathrm{b}}$.

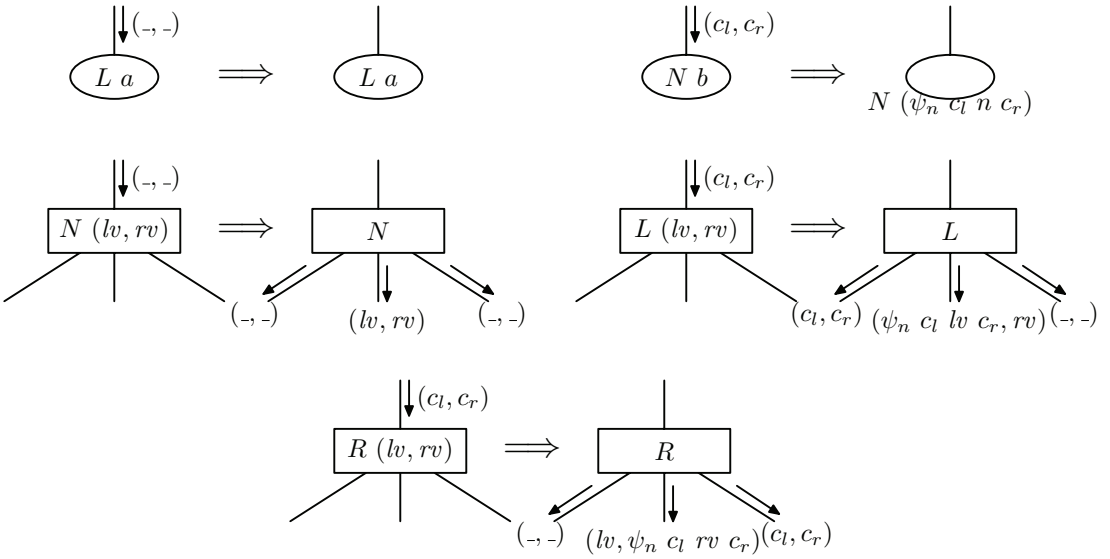


Figure 3.12. Illustration of the function $\mathsf{uAcc}^{d}_{\mathrm{b}}$.

After the bottom-up sweep, the top-down sweep computes the result for each node. Function $\mathsf{uAcc}^d_{\mathrm{b}}$ that performs the top-down sweep is defined as follows. The accumulative parameter is a pair of two values, which are the results of tree reductions of the two child segments. Since the global structure has no terminal node, the accumulative parameter is at the beginning a pair of don't-care values. Figure 3.12 shows an intuitive illustration of the function $\mathsf{uAcc}^d_{\mathrm{b}}$.

$$
\begin{aligned}
&\mathsf{uAcc}^d_{\mathrm{b}}\ \psi_n\ (\_,\_)\ (\mathsf{TLeafL}'\ a) = \mathsf{TLeafL}\ a\\
&\mathsf{uAcc}^d_{\mathrm{b}}\ \psi_n\ (c_l, c_r)\ (\mathsf{TLeafN}'\ b) = \mathsf{TLeafN}\ (\psi_n\ c_l\ b\ c_r)\\
&\mathsf{uAcc}^d_{\mathrm{b}}\ \psi_n\ (\_,\_)\ (\mathsf{TNodeN}'\ (lv, rv)\ l\ n\ r)\\
&\qquad = \mathsf{TNodeN}\ (\mathsf{uAcc}^d_{\mathrm{b}}\ \psi_n\ (\_,\_)\ l)\ (\mathsf{uAcc}^d_{\mathrm{b}}\ \psi_n\ (lv, rv)\ n)\\
&\qquad\qquad\quad (\mathsf{uAcc}^d_{\mathrm{b}}\ \psi_n\ (\_,\_)\ r)\\
&\mathsf{uAcc}^d_{\mathrm{b}}\ \psi_n\ (c_l, c_r)\ (\mathsf{TNodeL}'\ (lv, rv)\ l\ n\ r)\\
&\qquad = \mathsf{TNodeL}\ (\mathsf{uAcc}^d_{\mathrm{b}}\ \psi_n\ (c_l, c_r)\ l)\ (\mathsf{uAcc}^d_{\mathrm{b}}\ \psi_n\ (\psi_n\ c_l\ lv\ c_r, rv)\ n)\\
&\qquad\qquad\quad (\mathsf{uAcc}^d_{\mathrm{b}}\ \psi_n\ (\_,\_)\ r)\\
&\mathsf{uAcc}^d_{\mathrm{b}}\ \psi_n\ (c_l, c_r)\ (\mathsf{TNodeR}'\ (lv, rv)\ l\ n\ r)\\
&\qquad = \mathsf{TNodeR}\ (\mathsf{uAcc}^d_{\mathrm{b}}\ \psi_n\ (\_,\_)\ l)\ (\mathsf{uAcc}^d_{\mathrm{b}}\ \psi_n\ (lv, \psi_n\ c_l\ rv\ c_r)\ n)\\
&\qquad\qquad\quad (\mathsf{uAcc}^d_{\mathrm{b}}\ \psi_n\ (c_l, c_r)\ r)
\end{aligned}
$$

Using these two functions, we can implement the $\mathsf{uAcc}_{\mathrm{b}}$ skeleton on the ternary-tree representation.

$$\mathsf{uAcc}_{\mathrm{b}}\ \langle \phi, \psi_n, \psi_l, \psi_r\rangle_{\mathrm{u}} = \mathit{tt2bt} \circ \mathsf{uAcc}^d_{\mathrm{b}}\ \psi_n\ (\_,\_) \circ \mathit{fst} \circ \mathsf{uAcc}^u_{\mathrm{b}}\ \langle \phi, \psi_n, \psi_l, \psi_r\rangle_{\mathrm{u}} \circ \mathit{bt2tt}$$

Since the computation of the $\mathsf{uAcc}^u_{\mathrm{b}}$ and $\mathsf{uAcc}^d_{\mathrm{b}}$ functions is independent among the subtrees, the implementation can be easily parallelized by the divide-and-conquer approach.

Lemma 3.8 *The upwards accumulation* $\mathsf{uAcc}_{\mathrm{b}}\ \langle \phi, \psi_n, \psi_l, \psi_r\rangle_{\mathrm{u}}$ *can be implemented in parallel on the ternary-tree representation.*

Proof Sketch. An implementation of the upwards accumulation has been shown so far. We therefore only need to prove the correctness of the implementation. The proof consists of the following two parts: proving the correctness on the plain ternary trees, and verifying the tree associativity.

First, we prove the correctness of the implementation on the plain ternary trees. We prove the correctness by inductions on the structure of binary trees: $(\mathsf{BLeaf}\ a)$ for the base case, and $(\mathsf{BNode}\ l\ b\ r)$ for the inductive step. Here, we need to prove the following auxiliary proposition

$$\mathit{root}_{\mathrm{b}} \circ \mathsf{uAcc}_{\mathrm{b}}\ \langle \phi, \psi_n, \psi_l, \psi_r\rangle_{\mathrm{u}} = \mathit{snd} \circ \mathsf{uAcc}^u_{\mathrm{b}}\ \langle \phi, \psi_n, \psi_l, \psi_r\rangle_{\mathrm{u}} \circ \mathit{bt2tt}$$

in order to prove the main equation

$$\mathsf{uAcc}_{\mathrm{b}}\ \langle \phi, \psi_n, \psi_l, \psi_r\rangle_{\mathrm{u}} = \mathit{tt2bt} \circ \mathsf{uAcc}^d_{\mathrm{b}}\ \psi_n\ (\_,\_) \circ \mathit{fst} \circ \mathsf{uAcc}^u_{\mathrm{b}}\ \langle \phi, \psi_n, \psi_l, \psi_r\rangle_{\mathrm{u}} \circ \mathit{bt2tt}\ .$$

For tree associativity, we prove the following two equations that represent the tree associativity modulo function *tt2bt*. Here, the function $\mathsf{uAcc}_{\mathrm{t}}$ is the core part of the

implementation of the $\mathsf{uAcc_b}$ skeleton on ternary-tree representation, which is given as $\mathsf{uAcc_t} = \mathsf{uAcc_b^d}\ \psi_n\ (\_,\_) \circ \mathit{fst} \circ \mathsf{uAcc_b^u}\ \langle \phi, \psi_n, \psi_l, \psi_r \rangle_{\mathrm{u}}$

$$
\begin{aligned}
&\mathit{tt2bt}\ (\mathsf{uAcc_t}\ (\mathsf{TNodeN}\ (\mathsf{TNodeN}\ a\ b\ c)\ d\ e)) \\
&\qquad = \mathit{tt2bt}\ (\mathsf{uAcc_t}\ (\mathsf{TNodeN}\ a\ (\mathsf{TNodeL}\ b\ d\ e)\ c)) \\
&\mathit{tt2bt}\ (\mathsf{uAcc_t}\ (\mathsf{TNodeN}\ a\ b\ (\mathsf{TNodeN}\ c\ d\ e))) \\
&\qquad = \mathit{tt2bt}\ (\mathsf{uAcc_t}\ (\mathsf{TNodeN}\ c\ (\mathsf{TNodeR}\ a\ b\ d)\ e))
\end{aligned}
$$

We can prove these two equations by substituting the definition of the functions *tt2bt* and $\mathsf{uAcc_t}$ and unfolding the both sides of the equations. □

Implementation of Downwards Accumulation

The $\mathsf{dAcc_b}$ skeleton takes a pair of functions g_l and g_r. In Section 2.4, we gave the condition for the existence of parallel implementation as $(g_l, g_r) = \langle \phi_l, \phi_r, \psi_u, \psi_d \rangle_{\mathrm{d}}$ with four auxiliary functions. Similar to the case of implementing tree homomorphism, we can find such auxiliary functions for any given pair of functions if we do not care about the efficiency.

Lemma 3.9 *Let g_l and g_r be given functions. There exist a set of auxiliary functions ϕ_l, ϕ_r, ψ_u and ψ_d such that the following three equations hold.*

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

Proof. The following definition of the auxiliary functions satisfies the three equations above.

$$
\begin{aligned}
&\phi_l\ n = \lambda c.g_l\ c\ n \\
&\phi_r\ n = \lambda c.g_r\ c\ n \\
&\psi_d\ c\ f_n = f_n\ c \\
&\psi_u\ f_n\ f_m = f_m \circ f_n
\end{aligned}
$$

We can verify the equations by simple calculations. □

Therefore, we can derive a set of auxiliary functions for efficient parallel implementations from a form of unary functions closed under function composition.

In the following, assume $(g_l, g_r) = \langle \phi_l, \phi_r, \psi_u, \psi_d \rangle_{\mathrm{d}}$ hold. Under this assumption, we can implement the $\mathsf{dAcc_b}$ skeleton with a bottom-up sweep followed by a top-down sweep.

The bottom-up sweep computes the partial results from the root node to the two children of the terminal node for each segment. Function $\mathsf{dAcc_b^u}$ that computes the bottom-up sweep is defined as follows. In the computation of the function $\mathsf{dAcc_b^u}$, two local results

are put on the internal node and passed to the parent node. Figure 3.13 shows an intuitive illustration of the function $\mathsf{dAcc}^u_{\mathrm{b}}$.

$$
\begin{aligned}
&\mathsf{dAcc}^u_{\mathrm{b}}\ \langle \phi_l, \phi_r, \psi_u, \psi_d \rangle_{\mathrm{d}}\ (\mathsf{TLeafL}\ a) = (\mathsf{TLeafL}'\ \_, (\_, \_)) \\
&\mathsf{dAcc}^u_{\mathrm{b}}\ \langle \phi_l, \phi_r, \psi_u, \psi_d \rangle_{\mathrm{d}}\ (\mathsf{TLeafN}\ b) = (\mathsf{TLeafN}'\ \_, (\phi_l\ b, \phi_r\ b)) \\
&\mathsf{dAcc}^u_{\mathrm{b}}\ \langle \phi_l, \phi_r, \psi_u, \psi_d \rangle_{\mathrm{d}}\ (\mathsf{TNodeN}\ l\ n\ r) \\
&\quad = \mathbf{let}\ (l', (\_, \_)) \quad = \mathsf{dAcc}^u_{\mathrm{b}}\ \langle \phi_l, \phi_r, \psi_u, \psi_d \rangle_{\mathrm{d}}\ l \\
&\qquad\quad (n', (n_l, n_r)) = \mathsf{dAcc}^u_{\mathrm{b}}\ \langle \phi_l, \phi_r, \psi_u, \psi_d \rangle_{\mathrm{d}}\ n \\
&\qquad\quad (r', (\_, \_)) \quad = \mathsf{dAcc}^u_{\mathrm{b}}\ \langle \phi_l, \phi_r, \psi_u, \psi_d \rangle_{\mathrm{d}}\ r \\
&\qquad \mathbf{in}\ (\mathsf{TNodeN}'\ (n_l, n_r)\ l'\ n'\ r', (n_l, n_r)) \\
&\mathsf{dAcc}^u_{\mathrm{b}}\ \langle \phi_l, \phi_r, \psi_u, \psi_d \rangle_{\mathrm{d}}\ (\mathsf{TNodeL}\ l\ n\ r) \\
&\quad = \mathbf{let}\ (l', (l_l, l_r)) \quad = \mathsf{dAcc}^u_{\mathrm{b}}\ \langle \phi_l, \phi_r, \psi_u, \psi_d \rangle_{\mathrm{d}}\ l \\
&\qquad\quad (n', (n_l, n_r)) = \mathsf{dAcc}^u_{\mathrm{b}}\ \langle \phi_l, \phi_r, \psi_u, \psi_d \rangle_{\mathrm{d}}\ n \\
&\qquad\quad (r', (\_, \_)) \quad = \mathsf{dAcc}^u_{\mathrm{b}}\ \langle \phi_l, \phi_r, \psi_u, \psi_d \rangle_{\mathrm{d}}\ r \\
&\qquad \mathbf{in}\ (\mathsf{TNodeL}'\ (n_l, n_r)\ l'\ n'\ r', (\psi_u\ n_l\ l_l, \psi_u\ n_l\ l_r)) \\
&\mathsf{dAcc}^u_{\mathrm{b}}\ \langle \phi_l, \phi_r, \psi_u, \psi_d \rangle_{\mathrm{d}}\ (\mathsf{TNodeR}\ l\ n\ r) \\
&\quad = \mathbf{let}\ (l', (\_, \_)) \quad = \mathsf{dAcc}^u_{\mathrm{b}}\ \langle \phi_l, \phi_r, \psi_u, \psi_d \rangle_{\mathrm{d}}\ l \\
&\qquad\quad (n', (n_l, n_r)) = \mathsf{dAcc}^u_{\mathrm{b}}\ \langle \phi_l, \phi_r, \psi_u, \psi_d \rangle_{\mathrm{d}}\ n \\
&\qquad\quad (r', (r_l, r_r)) \quad = \mathsf{dAcc}^u_{\mathrm{b}}\ \langle \phi_l, \phi_r, \psi_u, \psi_d \rangle_{\mathrm{d}}\ r \\
&\qquad \mathbf{in}\ (\mathsf{TNodeR}'\ (n_l, n_r)\ l'\ n'\ r', (\psi_u\ n_r\ r_l, \psi_u\ n_r\ r_r))
\end{aligned}
$$

After the bottom-up sweep, the top-down sweep computes the result for each node. The following function $\mathsf{dAcc}^d_{\mathrm{b}}$ computes the top-down sweep. Two facts are worth noting: The value of the accumulative parameter is the same as the value passed to the corresponding segment, and the computations on TNodeN', TNodeL', and TNodeR' are the same except for the constructors. Figure 3.14 shows an intuitive illustration of the function $\mathsf{dAcc}^d_{\mathrm{b}}$.

$$
\begin{aligned}
&\mathsf{dAcc}^d_{\mathrm{b}}\ \psi_d\ c\ (\mathsf{TLeafL}'\ \_) = \mathsf{TLeafL}\ c \\
&\mathsf{dAcc}^d_{\mathrm{b}}\ \psi_d\ c\ (\mathsf{TLeafN}'\ \_) = \mathsf{TLeafN}\ c \\
&\mathsf{dAcc}^d_{\mathrm{b}}\ \psi_d\ c\ (\mathsf{TNodeN}'\ (n_l, n_r)\ l\ n\ r) \\
&\quad = \mathsf{TNodeN}\ (\mathsf{dAcc}^d_{\mathrm{b}}\ \psi_d\ (\psi_d\ c\ n_l)\ l)\ (\mathsf{dAcc}^d_{\mathrm{b}}\ \psi_d\ c\ n)\ (\mathsf{dAcc}^d_{\mathrm{b}}\ \psi_d\ (\psi_d\ c\ n_r)\ r) \\
&\mathsf{dAcc}^d_{\mathrm{b}}\ \psi_d\ c\ (\mathsf{TNodeL}'\ (n_l, n_r)\ l\ n\ r) \\
&\quad = \mathsf{TNodeL}\ (\mathsf{dAcc}^d_{\mathrm{b}}\ \psi_d\ (\psi_d\ c\ n_l)\ l)\ (\mathsf{dAcc}^d_{\mathrm{b}}\ \psi_d\ c\ n)\ (\mathsf{dAcc}^d_{\mathrm{b}}\ \psi_d\ (\psi_d\ c\ n_r)\ r) \\
&\mathsf{dAcc}^d_{\mathrm{b}}\ \psi_d\ c\ (\mathsf{TNodeR}'\ (n_l, n_r)\ l\ n\ r) \\
&\quad = \mathsf{TNodeR}\ (\mathsf{dAcc}^d_{\mathrm{b}}\ \psi_d\ (\psi_d\ c\ n_l)\ l)\ (\mathsf{dAcc}^d_{\mathrm{b}}\ \psi_d\ c\ n)\ (\mathsf{dAcc}^d_{\mathrm{b}}\ \psi_d\ (\psi_d\ c\ n_r)\ r)
\end{aligned}
$$

Using these two functions, we can implement the $\mathsf{dAcc}_{\mathrm{b}}$ skeleton on the ternary-tree representation.

$$\mathsf{dAcc}_{\mathrm{b}}\ \langle \phi_l, \phi_r, \psi_u, \psi_d \rangle_{\mathrm{d}}\ c = \mathit{tt2bt} \circ \mathsf{dAcc}^d_{\mathrm{b}}\ \psi_d\ c \circ \mathit{fst} \circ \mathsf{dAcc}^u_{\mathrm{b}}\ \langle \phi_l, \phi_r, \psi_u, \psi_d \rangle_{\mathrm{d}} \circ \mathit{bt2tt}$$

Since the computation of the $\mathsf{dAcc}^u_{\mathrm{b}}$ and $\mathsf{dAcc}^d_{\mathrm{b}}$ functions is independent among the subtrees, we can easily implement the $\mathsf{dAcc}_{\mathrm{b}}$ skeleton in parallel by the divide-and-conquer approach.

Lemma 3.10 *The downwards accumulation* $\mathsf{dAcc}_{\mathrm{b}}\ \langle \phi_l, \phi_r, \psi_u, \psi_d \rangle_{\mathrm{d}}$ *can be implemented in parallel on the ternary-tree representation.*

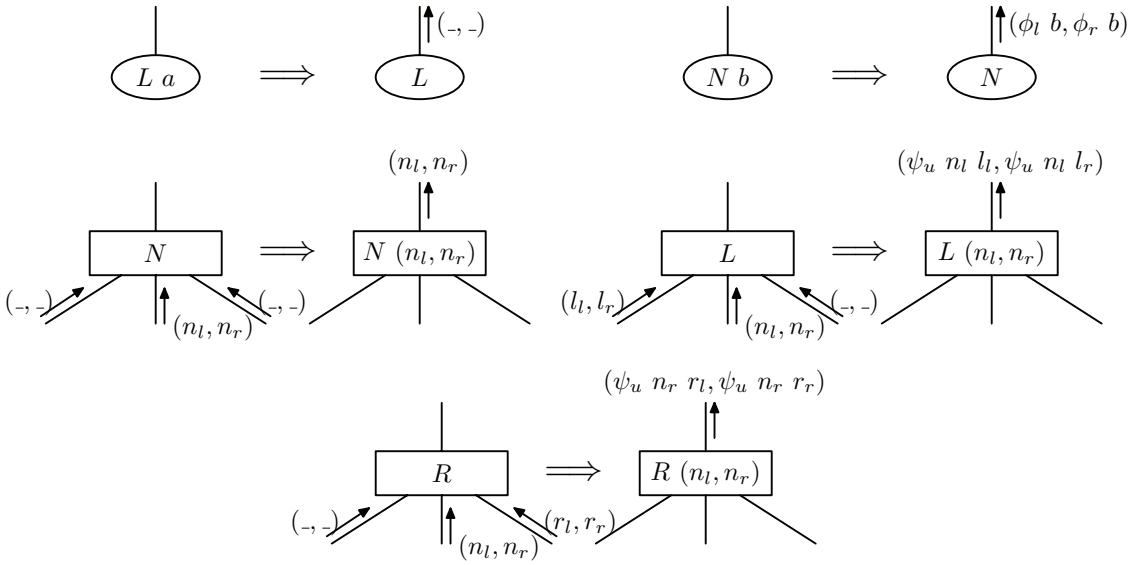


Figure 3.13. Illustration of the function $\mathsf{dAcc}^{u}_{\mathrm{b}}$.

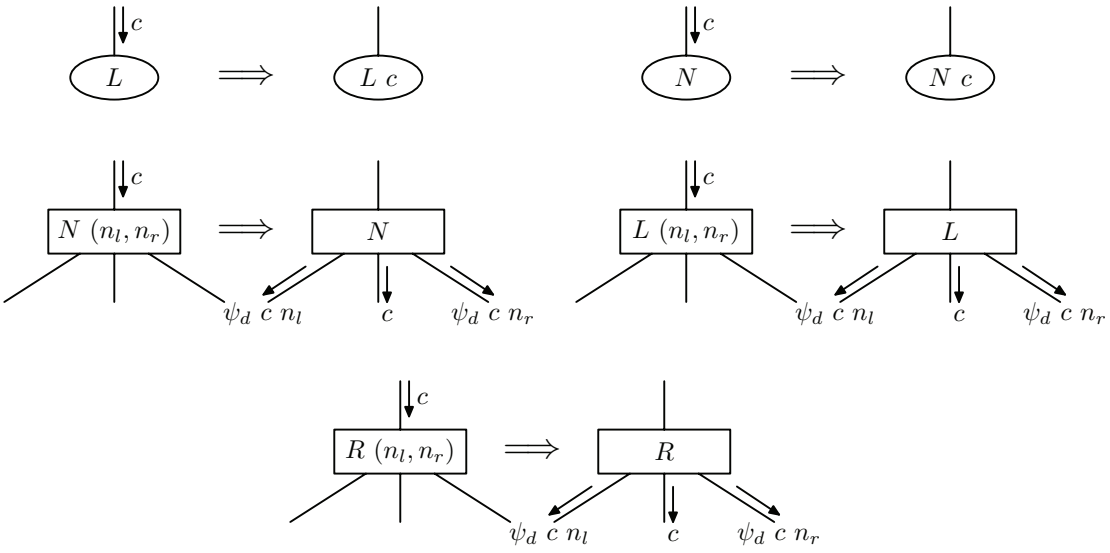


Figure 3.14. Illustration of the function $\mathsf{dAcc}^{d}_{\mathrm{b}}$.

Proof Sketch. The implementation of the $\mathsf{dAcc_b}$ on the ternary-tree representation has been shown so far. We therefore only need to prove the correctness of the implementation. Similar to the proof in Lemma 3.8, the proof consists of two parts: proving the correctness on the plain ternary trees, and verifying the tree associativity.

First, we show that the implementation is correct for the plain ternary tree, by proving the equation

$$\mathsf{dAcc_b}\ \langle \phi_l, \phi_r, \psi_u, \psi_d \rangle_{\mathrm{d}}\ c = \mathit{tt2bt} \circ \mathsf{dAcc_b^d}\ \psi_d\ c \circ \mathit{fst} \circ \mathsf{dAcc_b^u}\ \langle \phi_l, \phi_r, \psi_u, \psi_d \rangle_{\mathrm{d}} \circ \mathit{bt2tt}$$

by induction on the structure of binary trees: $(\mathsf{BLeaf}\ a)$ for the base case, and $(\mathsf{BNode}\ l\ b\ r)$ for the induction step. In the induction step, we use two equations $g_l\ c\ b = \psi_d\ c\ (\phi_l\ n)$ and $g_r\ c\ b = \psi_d\ c\ (\phi_r\ n)$.

Then, we verify that the implementation is tree associative modulo function *tt2bt*. Let $\mathsf{dAcc_t}$ be defined as

$$\mathsf{dAcc_t} = \mathsf{dAcc_b^d}\ \psi_d\ c \circ \mathit{fst} \circ \mathsf{dAcc_b^u}\ \langle \phi_l, \phi_r, \psi_u, \psi_d \rangle_{\mathrm{d}}\ ,$$

we prove the following two equations.

$$\begin{aligned}
&\mathit{tt2bt}\ (\mathsf{dAcc_t}\ (\mathsf{TNodeN}\ (\mathsf{TNodeN}\ a\ b\ c)\ d\ e)) \\
&\quad = \mathit{tt2bt}\ (\mathsf{uAcc_t}\ (\mathsf{TNodeN}\ a\ (\mathsf{TNodeL}\ b\ d\ e)\ c)) \\
&\mathit{tt2bt}\ (\mathsf{dAcc_t}\ (\mathsf{TNodeN}\ a\ b\ (\mathsf{TNodeN}\ c\ d\ e))) \\
&\quad = \mathit{tt2bt}\ (\mathsf{uAcc_t}\ (\mathsf{TNodeN}\ c\ (\mathsf{TNodeR}\ a\ b\ d)\ e))
\end{aligned}$$

We use equation $\psi_d\ (\psi_d\ c\ n)\ m = \psi_d\ c\ (\psi_u\ n\ m)$ in the calculations of the proof.

It is worth remarking that all the three equations of the condition for parallel implementations are essentially used in the proof of this lemma. □

3.4 Balanced Ternary-Tree Representation

In the previous sections we studies the basic property of tree associativity and ternary-tree representation and developed parallel implementations of tree homomorphisms on ternary trees. The parallel cost of the implementations developed in Section 3.3 is linear to the height of ternary trees, and it is important to reduce the height of ternary trees for efficient parallel implementations. In this section, we develop an algorithm for generating balanced ternary trees. Generating balanced ternary trees can be considered as a preprocessing of trees for efficient parallel computation.

We observed a recursive division of binary tree in Section 3.2. The recursive division generates a ternary tree in a top-down manner since a division specifies the root of the ternary tree. To obtain a well-balanced tree, we should find a node such that the three segments have almost the same size. Finding such a node with small cost is, however, not so easy.

We here develop another sequential algorithm that generates a balanced ternary-tree in a bottom-up manner in the sense that the ternary tree is constructed from leaves. Our

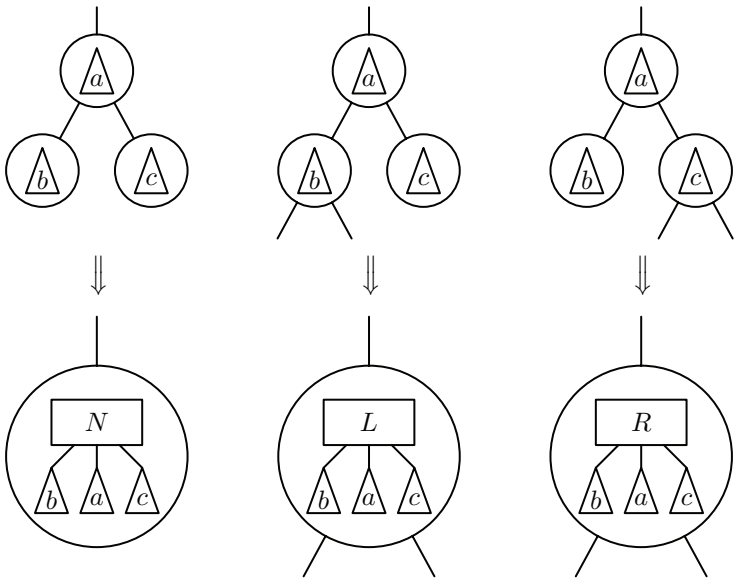


Figure 3.15. Three local merges and labels assigned.

algorithm uses the two contracting operations in the algorithm given by Abrahamson et al. [2]. By simulating Abrahamson et al.'s algorithm, we can obtain a ternary tree with its height at most $2 \log N$ where N denotes the number of nodes of the input binary tree. To study the balancing of ternary trees further, we develop a greedy algorithm and quantify the height of generated ternary trees.

The algorithm mainly consists of two steps. First, we put label TLeafL for each leaf node and label TLeafN for each internal node in a binary tree. Then, we iteratively merge three adjacent nodes into one using the two contracting operation. To keep the global shape to be binary, we merge a node and its two children if at least one child is a leaf. If either of the children is an internal node, then the node remains to be an internal node and will be merged again later. Therefore, this node can be considered as a terminal node of the merged segment. Based on this observation, we assign labels in the following rules when we merge three nodes. Figure 3.15 illustrates these local merges.

- TNodeN is assigned if both children are leaves.
- TNodeL is assigned if the left child is an internal node and the right child is a leaf.
- TNodeR is assigned if the right child is an internal node and the left child is a leaf.

We obtain a ternary tree by applying these merges repeatedly until the global binary tree consists of one node. To make the ternary tree balanced, we apply merges disjointly as much as possible in one step of the iteration, and a greedy algorithm for transforming a binary tree into a balanced ternary tree is given as follows.

Algorithm 3.1 (Greedy Balanced Ternary-Tree Generation)
Input: A binary tree.
Output: A balanced ternary-tree representation of the input binary tree.

1. Put label TLeafL to each leaf and label TLeafN to each internal node.
2. Iterate the following steps 2.1 and 2.2 until the global binary tree consists of only one leaf node.
   2.1 Do nothing for each leaf.
   2.2 Perform a bottom-up computation by applying one of the following rules to each internal node.
   - If either of children of an internal node has already been merged in the bottom-up computation, then do nothing for the node.
   - If no children are merged and both children are internal nodes, then do nothing for the node.
   - If no children are merged and at least either of children is a leaf, then merge the node with two children with the labels given in Figure 3.15.
3. The ternary-tree representation is given as the value of the remaining node. □

One implementation of the algorithm is given as follows. In the following program, function $bt2tt'$ performs the one iterative step of Step 2 in the algorithm above, and function $bt2tt''$ merges three nodes if possible. To check whether we can merge three nodes or not, we define function $bt2tt'$ to return a boolean value that represents where the node is merged or not.

$$
\begin{aligned}
&bt2tt :: \mathsf{BTree}\ a\ b \to \mathsf{TTree}\ a\ b \\
&bt2tt = root_{\mathrm{b}} \circ head \circ dropWhile\ (\lambda t.size_{\mathrm{b}}\ t > 1) \circ \\
&\qquad iterate\ (fst \circ bt2tt')\ \circ \mathsf{map}_{\mathrm{b}}\ (\lambda a.(\mathsf{TLeafL}\ a))\ (\lambda b.(\mathsf{TLeafN}\ b))
\end{aligned}
$$

$$
\begin{aligned}
&bt2tt'\ (\mathsf{BLeaf}\ n) &&= (\mathsf{BLeaf}\ n, \mathsf{False}) \\
&bt2tt'\ (\mathsf{BNode}\ n\ l\ r) &&= \mathbf{let}\ (l', flag_l) = bt2tt'\ l \\
&&&\qquad\quad (r', flag_r) = bt2tt'\ r \\
&&&\quad \mathbf{in\ if}\ flag_l \vee flag_r\ \mathbf{then}\ (\mathsf{BNode}\ n\ l'\ r', \mathsf{False}) \\
&&&\qquad\qquad\qquad\ \mathbf{else}\ bt2tt''\ n\ l'\ r'
\end{aligned}
$$

$$
\begin{aligned}
&bt2tt''\ n\ (\mathsf{BLeaf}\ l)\ (\mathsf{BLeaf}\ r) &&= (\mathsf{BLeaf}\ (\mathsf{TNodeN}\ l\ n\ r), \mathsf{True}) \\
&bt2tt''\ n\ (\mathsf{BLeaf}\ l)\ (\mathsf{BNode}\ rn\ rl\ rr) &&= (\mathsf{BNode}\ (\mathsf{TNodeR}\ l\ n\ rn)\ rl\ rr, \mathsf{True}) \\
&bt2tt''\ n\ (\mathsf{BNode}\ ln\ ll\ lr)\ (\mathsf{BLeaf}\ r) &&= (\mathsf{BNode}\ (\mathsf{TNodeL}\ ln\ n\ r)\ ll\ lr, \mathsf{True}) \\
&bt2tt''\ n\ l'\ r' &&= (\mathsf{BNode}\ n\ l'\ r', \mathsf{False})
\end{aligned}
$$

In the following, we analyze the height of resulting ternary trees and the computational time of the algorithm. First we count the number of disjoint merges in one step of the iteration.

Lemma 3.11 *Let L be the number of leaves in a binary tree, then the number of disjoint merges in one call of $bt2tt'$ is more than or equal to $L/3$.*

Proof. When a leaf node cannot be merged, its sibling must be an internal node and be already merged in the call of *bt2tt′*. In one merge, at most two leaves are involved. Therefore, for one merge at most two leaves are involved in the merge, and at most one leaf cannot be merged due to the merge. That is, let d be the number of disjoint merge, $(2+1) \times d \geq L$ holds. The lemma directly follows from the inequality. □

Let N be the number of nodes in a binary tree, then the number of leaves is $(N+1)/2$, and thus we can apply at least $\lceil (N+1)/6 \rceil$ merges disjointly. A single merge reduces the number of nodes by two. Therefore the number of nodes becomes less than or equal to $N - \lceil (N+1)/3 \rceil$ after one iterative step of Step 2 in the algorithm.

The height of the ternary tree is the number of iterations in Step 2, and thus we can estimate the height of the generated balanced ternary trees by the following theorem. Note that a binary tree with three nodes is transformed into a tree with one node at the end of iteration.

Theorem 3.1 *Let N be the number of nodes and h be the height of ternary trees generated by function bt2tt. Then the following equation holds.*

$$h \leq \frac{\log_2(N+1) - 2}{\log_2 3 - 1} + 2$$

Proof. Let a_i be the number of node after ith iteration of function *bt2tt′*. By definition, $a_0 = N$. The height of the generated ternary tree is given by the number of iteration of function *bt2tt′*, and thus $a_{h-1} = 1$. As stated above, at the last iteration, the number of nodes decreases from 3 to 1, that is, $a_{h-2} = 3$ holds. At other iterations of function *bt2tt′*, the following inequality holds.

$$a_i \leq a_{i-1} - \left\lceil \frac{a_{i-1}+1}{3} \right\rceil \leq \frac{2}{3} a_{i-1} - \frac{1}{3}$$

By solving this recurrence inequality, we obtain

$$a_i + 1 \leq \left(\frac{2}{3}\right)^i (a_0 + 1) \ ,$$

and substituting $h-2$ for i we obtain the following inequality between the number of nodes N and the height of ternary tree h.

$$3 + 1 \leq \left(\frac{2}{3}\right)^{h-2} (N+1)$$

Now we can solve this inequality and obtain the height of ternary tree as follows.

$$\begin{aligned} 3+1 &\leq \left(\frac{2}{3}\right)^{h-2} (N+1) \\ 2 &\leq (h-2)(1-\log_2 3) + log_2(N+1) \\ h &\leq \frac{log_2(N+1) - 2}{\log_2 3 - 1} + 2 \end{aligned}$$

□

Approximating $\log_2 3 \approx 1.585$, we can estimate the height of ternary tree as $1.71 \log_2(N + 1) - 1.42$. Compared with a simple simulation of Abrahamson et al.'s tree contraction algorithm that generates a ternary tree of height $2 \log_2 N$, our algorithm generates better-balanced ternary trees.

We then analyze the computational cost of the greedy balanced ternary-tree generation. The main part of the algorithm is Step 2, which consists of iteration of function *bt2tt′*. From Lemma 3.11 the number of nodes decreases to less than two thirds of the original number of nodes. Therefore, the computational cost of the algorithm is given as the following lemma says.

Lemma 3.12 *Let N be the number of nodes in an input binary tree. The computational cost of the function bt2tt is* $\mathrm{O}(N)$.

Proof. We can compute Step 1 by traversing the input binary tree once, which takes linear time to the size of the binary tree. The function *bt2tt′* also traverses a binary tree once. After one application of function *bt2tt′* reduces the number of nodes into less than two thirds of the original number of nodes. Therefore, we can estimate the number of nodes traversed in Step 2 as

$$N + \frac{2}{3}N + \left(\frac{2}{3}\right)^2 N + \cdots + 1 < N\left(1 + \frac{2}{3} + \left(\frac{2}{3}\right)^2 + \cdots\right) = 3N \ .$$

The cost of Step 3 is obviously constant.

In summary, the computational cost of the algorithm is linear to the number of nodes of the input binary trees. □

The balanced ternary trees guarantee the efficiency of the tree homomorphisms and the tree accumulations implemented by a simple divide-and-conquer on ternary trees as shown in Section 3.3. We conclude this section with the following theorem that states the cost of parallel programs implemented based on this ternary-tree representation.

Theorem 3.2 *Let N be the number of nodes of a binary tree. The tree homomorphisms and the tree accumulations can be executed in parallel in* $\mathrm{O}(\log N)$ *parallel time, after preprocessing of* $\mathrm{O}(N)$ *sequential time.*

Proof. It follows from Theorem 3.1 and Lemma 3.12. □

3.5 Discussion

The basic idea to represent a (binary) tree with a balanced tree structure has been studied so far. One of such representations is a balanced decomposition tree, where a decomposition tree is generated by recursive removal of an edge from a tree. There are many applications on this decomposition tree, especially in computational geometry [27]. There are also studies for deriving such a decomposition tree in parallel [130, 131]. The decomposition tree is binary tree and therefore is easy to manipulate, but due to the loss of

structural information the computation applicable on the decomposition trees is limited. Balanced ternary-tree representation in this chapter keeps structural information of binary trees and thus any computation can be mapped onto it if we do not matter efficiency.

In Section 3.3.2, we have shown an implementation of tree accumulations on the ternary-tree representation. There were some studies on the parallel implementation of the tree accumulations based on the parallel tree contraction algorithms.

Leiserson and Maggs [78] studied the parallel implementation of tree accumulations on a model of parallel computers named distributed random access machines. Their algorithms for tree accumulations are based on Miller and Reif's tree contraction algorithm, and they consist of the contraction phase and the expansion phase. We construct a *contraction tree* by merging two nodes recursively in the contraction phase, and then in the expansion phase we expand the contraction tree in the reversed order to the contractions. The contraction and expansion phase correspond to our bottom-up and top-down sweeps, respectively.

Gibbons et al. [50] developed a parallel implementation of tree accumulations based on Abrahamson et al's tree contraction algorithms. The implementation also consists of two phases where the contraction is extended with stacks. The stacks in Gibbons et al.'s implementation correspond to the paths from leaves labeled as TLeafN to the root node.

3.6 Short Summary

In this chapter, we have defined the ternary-tree representation of binary trees and formalized a new property named *tree-associativity* on this ternary-tree representation. We can implement tree homomorphisms and tree accumulations on the ternary-tree representation, where the implementation given in this chapter can be considered as a reformalization of the existing tree-contraction based parallel algorithms. We have also developed an algorithm that derives a balanced ternary tree from a given binary tree.

On the formalization of the ternary-tree representation, it would be interesting to study more complicated parallel algorithms, for example, generalized tree contraction algorithms proposed by Diks and Hagerup [41]. Developing a dynamic balancing algorithm that guarantees the balance after insertion or deletion of nodes is an open problem.

Chapter 4

Rose-Tree Skeletons

In many applications, we use not only binary trees but also general trees whose internal nodes may have more than two children. We can solve the two-dimensional N-body problem using quad-trees given by the recursive division of the field. An internal node of XML trees may have an arbitrary number of children. In this chapter, we consider the rose trees [96], a class of general trees of uniform elements, as the target data structure.

There have been many studies addressed on the parallel computation for binary trees. However, there have been only a few studies for rose trees. It is not clear how we can implement manipulations on rose trees efficiently in parallel or how we can map manipulations on general trees onto the manipulations on binary trees for which we have efficient implementations.

In this chapter we study parallel skeletons for rose trees. We formalize seven parallel rose-tree skeletons. Five rose-tree skeletons are straightforward extensions of those on the binary trees, and thus we can develop skeletal programs for rose trees in a similar way to those for binary trees. Additional two rose-tree skeletons are for manipulations among the siblings, which are essential in several algorithms for rose trees.

We implement rose-tree skeletons in parallel by using the parallel binary-tree skeletons. We first introduce a binary-tree representation of rose-trees, and then develop an implementation of rose-tree skeletons one by one. We formalize a new property on a pair of operators named *extended-distributivity* for the condition of parallel implementation of rose-tree skeletons.

The organization of this chapter is as follows. In Section 4.1, we formalize a new property held on a pair of operators by extending distributivity. In Section 4.2, we define seven parallel skeletons for rose trees and illustrate how we can solve problems with these rose-tree skeletons. These rose-tree skeletons can be implemented efficiently in parallel, and we show an implementation in Section 4.3. We summarize this chapter in Section 4.4.

The contents of this chapter is based on [90], and a preliminary version appeared in [85, 86, 89].

4.1 Extension of Distributive Law

In developing parallel programs, algebraic properties and rules on operators often play important roles. Associativity is one of the most important properties in discussing parallel computing, and many studies have clarified how we can derive and implement parallel programs based on the associativity of one operator used in the programs. In programs manipulating rose trees, two operators are essentially used; one among siblings and the other between a node and its children. In this section we formalize a new property on two operators as an extension of distributivity.

Distributivity is a well-known and important property on two operators. The distributive law together with the associative law enables us to simplify a complex expression defined with two operators. For example, expression $6 + 7 \times (8 + 4 \times x)$ can be simplified into $62 + 28 \times x$. It has been widely-known that first-order recurrences can be evaluated in parallel if two operators used satisfy the associative and the distributive laws [77, 112].

However, distributivity is a so strict property that there are many pairs of operators for which distributivity does not hold. A necessary condition of distributivity is that the two operators are defined on the same domain, but many operators in programs do not satisfy even this condition. We propose another property on pairs of operators to discuss derivation of parallel programs for rose trees in a general way.

In developing parallel programs we use the distributive law to simplify the given expressions as we have seen in the above. We introduce a generalized rule of distributivity by focusing on the closure property.

Definition 4.1 (Extended Distributivity [86]) Let $\otimes$ be an associative operator. Operator $\otimes$ is said to be *extended-distributive* over operator $\oplus$, if there exist functions p_1, p_2, and p_3 such that for any a, b, c, a', b', and c', the following equation holds.

$$\begin{aligned}(\lambda x.a \oplus (b \otimes x \otimes c)) \circ (\lambda x.a' \oplus (b' \otimes x \otimes c')) = \lambda x.A \oplus (B \otimes x \otimes C)\\ \textbf{where } A = p_1\ (a, b, c, a', b', c')\\ B = p_2\ (a, b, c, a', b', c')\\ C = p_3\ (a, b, c, a', b', c')\end{aligned}$$

We call the functions p_1, p_2, and p_3 *characteristic functions*. □

We may simplify this definition if the operator $\otimes$ is not only associative but also commutative.

Lemma 4.1 *Let $\otimes$ be an associative and commutative operator with its unit $\iota_\otimes$. If there exist functions p_1 and p_2 such that the following equation holds for any a, b, a', and b', then the operator $\otimes$ is extended-distributive over operator $\oplus$.*

$$\begin{aligned}(\lambda x.a \oplus (b \otimes x)) \circ (\lambda x.a' \oplus (b' \otimes x)) = \lambda x.A \oplus (B \otimes x)\\ \textbf{where } A = p_1\ (a, b, a', b')\\ B = p_2\ (a, b, a', b')\end{aligned}$$

Proof. We can derive the characteristic functions of extended-distributivity, p'_1, p'_2, and p'_3, as follows.

$$\begin{aligned}
&p'_1\ (a, b, c, a', b', c') = p_1\ (a, b \otimes c, a', b' \otimes c') \\
&p'_2\ (a, b, c, a', b', c') = p_2\ (a, b \otimes c, a', b' \otimes c') \\
&p'_3\ (a, b, c, a', b', c') = \iota_\otimes
\end{aligned}$$

□

Although the definition of extended-distributivity is a little complicated, it has the advantage of many applications in discussing derivation of parallel programs. In fact, many pairs of operators satisfy extended distributivity.

Firstly, extended-distributivity can replace associativity. Consider the case where two operators $\oplus$ and $\otimes$ are the same associative operator.

Lemma 4.2 *Associative operator $\otimes$ with its unit $\iota_\otimes$ is extended-distributive over $\otimes$ itself.*
Proof. The characteristic functions are given as follows.

$$\begin{aligned}
&p_1\ (a_1, b_1, c_1, a_2, b_2, c_2) = a_1 \otimes b_1 \otimes a_2 \otimes b_2 \\
&p_2\ (a_1, b_1, c_1, a_2, b_2, c_2) = \iota_\otimes \\
&p_3\ (a_1, b_1, c_1, a_2, b_2, c_2) = c_2 \otimes c_1
\end{aligned}$$

□

Secondly, extended-distributivity is an generalization of distributivity. If a pair of operators forms an algebraic semiring then extended-distributivity holds as well.

Lemma 4.3 *Let $\oplus$ be an associative operator, and $\otimes$ be an associative operator that also distributes over $\oplus$. Then, the operator $\otimes$ is extended distributive over $\oplus$.*
Proof. The characteristic functions are given as follows.

$$\begin{aligned}
&p_1\ (a_1, b_1, c_1, a_2, b_2, c_2) = a_1 \oplus (b_1 \otimes a_2 \otimes c_1) \\
&p_2\ (a_1, b_1, c_1, a_2, b_2, c_2) = b_1 \otimes b_2 \\
&p_3\ (a_1, b_1, c_1, a_2, b_2, c_2) = c_2 \otimes c_1
\end{aligned}$$

□

There are many pairs of operators that satisfy the extended distributivity but not the distributivity. One example is in serializing an XML tree to its string representation with open and close tags [86]. Let operator $\oplus$ be defined as follows where s denotes a start tag, e denotes a end tag, and t denotes serialized text of children.

$$(s, e) \oplus t = s +\!\!+ t +\!\!+ e$$

We can easily see that the distributivity does not hold between $\oplus$ and $+\!\!+$ since the domains of two arguments of $\oplus$ are different, i.e., a pair of string for the left and string for the right. The extended-distributivity holds as shown in the following calculations.

$$\begin{aligned}
&(\lambda x.(s, e) \oplus (t_1 +\!\!+ x +\!\!+ t_2)) \circ (\lambda x.(s', e') \oplus (t'_1 +\!\!+ x +\!\!+ t'_2) \\
=\ &\quad \{\text{definition of } \oplus\} \\
&(\lambda x.s +\!\!+ t_1 +\!\!+ x +\!\!+ t_2 +\!\!+ e) \circ (\lambda x.s' +\!\!+ t'_1 +\!\!+ x +\!\!+ t'_2 +\!\!+ e') \\
=\ &\quad \{\text{unfolding}\} \\
&\lambda x.s +\!\!+ t_1 +\!\!+ s' +\!\!+ t'_1 +\!\!+ x +\!\!+ t'_2 +\!\!+ e' +\!\!+ t_2 +\!\!+ e \\
=\ &\quad \{\text{extracting } s \text{ and } e \text{ for the left argument of } \oplus\} \\
&\lambda x.(s, e) \oplus ((t_1 +\!\!+ s' +\!\!+ t'_1) +\!\!+ x +\!\!+ (t'_2 +\!\!+ e' +\!\!+ t_2))
\end{aligned}$$

From the calculations above, we obtain the characteristic functions p_1, p_2, and p_3 for the operators $\oplus$ and $+\!\!+$ as follows.

$$
\begin{aligned}
&p_1\ ((s,e),t_1,t_2,(s',e'),t_1',t_2') = (s,e)\\
&p_2\ ((s,e),t_1,t_2,(s',e'),t_1',t_2') = t_1 +\!\!+ s' +\!\!+ t_1'\\
&p_3\ ((s,e),t_1,t_2,(s',e'),t_1',t_2') = t_2' +\!\!+ e' +\!\!+ t_2
\end{aligned}
$$

It is worth remarking that the definition of characteristic functions is not unique. Another definition of characteristic functions is given as follows, which was shown in our previous paper [86].

$$
\begin{aligned}
&p_1\ ((s,e),t_1,t_2,(s',e'),t_1',t_2') = (s +\!\!+ t_1 +\!\!+ s' +\!\!+ t_1',\ t_2' +\!\!+ e' +\!\!+ t_2 +\!\!+ e)\\
&p_2\ ((s,e),t_1,t_2,(s',e'),t_1',t_2') = [\,]\\
&p_3\ ((s,e),t_1,t_2,(s',e'),t_1',t_2') = [\,]
\end{aligned}
$$

4.2 Rose-Tree Skeletons

In this section, we formalize basic computational patterns on rose trees called *rose-tree skeletons* based on the idea of constructive algorithmics [15,18,67]. The rose-tree skeletons are given as extensions of the parallel binary-tree skeletons in Chapter 2 and the parallel list skeletons [19, 117, 118].

4.2.1 Specification of Rose-Tree Skeletons

The key idea of constructive algorithmics is that the computational structure of algorithms should be derivable from the data structures the algorithms manipulate. We have defined rose trees in Section 2.1 in such a way that an internal node has a list of children. Therefore, we specify computational patterns on rose trees as extensions of binary-tree skeletons with manipulations of lists. We formalize seven skeletons on rose trees, which are categorized into the following four groups. We denote rose-tree skeletons in the sans-serif font with a subscript r.

- Two Node-wise computations: map_r and $\mathsf{zipwith}_\mathrm{r}$
- Two Bottom-up computations: $\mathsf{reduce}_\mathrm{r}$ and uAcc_r (upwards accumulate)
- One Top-down computation: dAcc_r (downwards accumulate)
- Two Intra-siblings computations: rAcc_r (rightwards accumulate) and lAcc_r (leftwards accumulate)

Figure 4.1 summarizes the specification of the rose-tree skeletons defined intuitively by list comprehension. See our technical report [89] for another specification of rose-tree skeletons given as mutual recursive functions.

$$
\begin{aligned}
&\mathsf{map}_\mathrm{r} :: (\alpha \to \beta) \to \mathsf{RTree}\ \alpha \to \mathsf{RTree}\ \beta\\
&\mathsf{map}_\mathrm{r}\ k\ (\mathsf{RNode}\ a\ ts) = \mathsf{RNode}\ (k\ a)\ [\mathsf{map}_\mathrm{r}\ k\ t_i \mid i \in [1..\#ts]]\\
&\\
&\mathsf{zipwith}_\mathrm{r} :: (\alpha \to \alpha' \to \beta) \to \mathsf{RTree}\ \alpha \to \mathsf{RTree}\ \alpha' \to \mathsf{RTree}\ \beta\\
&\mathsf{zipwith}_\mathrm{r}\ k\ (\mathsf{RNode}\ a\ ts)\ (\mathsf{RNode}\ a'\ ts')\\
&\qquad = \mathsf{RNode}\ (k\ a\ a')\ [\mathsf{zipwith}_\mathrm{r}\ k\ t_i\ t'_i \mid i \in [1..\#ts]]\\
&\\
&\mathsf{reduce}_\mathrm{r} :: (\alpha \to \beta \to \beta) \to (\beta \to \beta \to \beta) \to \mathsf{RTree}\ \alpha \to \beta\\
&\mathsf{reduce}_\mathrm{r}\ (\oplus)\ (\otimes)\ (\mathsf{RNode}\ a\ ts) = a \oplus \textstyle\sum_{\otimes}[\mathsf{reduce}_\mathrm{r}\ (\oplus)\ (\otimes)\ t_i \mid i \in [1..\#ts]]\\
&\\
&\mathsf{reduce}'_\mathrm{r} :: (\alpha \to \alpha \to \alpha) \to (\alpha \to \alpha \to \alpha) \to \mathsf{RTree}\ \alpha \to \alpha\\
&\mathsf{reduce}'_\mathrm{r}\ (\oplus)\ (\otimes)\ (\mathsf{RNode}\ a\ [\,])\ \ \ \ = a\\
&\mathsf{reduce}'_\mathrm{r}\ (\oplus)\ (\otimes)\ (\mathsf{RNode}\ a\ (t : ts)) = \mathbf{let}\ ts' = (t : ts)\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{in}\ a \oplus \textstyle\sum_{\otimes}[\mathsf{reduce}'_\mathrm{r}\ (\oplus)\ (\otimes)\ t'_i \mid i \in [1..\#ts']]\\
&\\
&\mathsf{uAcc}_\mathrm{r} :: (\alpha \to \beta \to \beta) \to (\beta \to \beta \to \beta) \to \mathsf{RTree}\ \alpha \to \mathsf{RTree}\ \beta\\
&\mathsf{uAcc}_\mathrm{r}\ (\oplus)\ (\otimes)\ (\mathsf{RNode}\ a\ ts) = \mathbf{let}\ a' = \mathsf{reduce}_\mathrm{r}\ (\oplus)\ (\otimes)\ (\mathsf{RNode}\ a\ ts)\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{in}\ \mathsf{RNode}\ a'\ [\mathsf{uAcc}_\mathrm{r}\ (\oplus)\ (\otimes)\ t_i \mid i \in [1..\#ts]]\\
&\\
&\mathsf{uAcc}'_\mathrm{r} :: (\alpha \to \alpha \to \alpha) \to (\alpha \to \alpha \to \alpha) \to \mathsf{RTree}\ \alpha \to \mathsf{RTree}\ \alpha\\
&\mathsf{uAcc}'_\mathrm{r}\ (\oplus)\ (\otimes)\ (\mathsf{RNode}\ a\ ts) = \mathbf{let}\ a' = \mathsf{reduce}'_\mathrm{r}\ (\oplus)\ (\otimes)\ (\mathsf{RNode}\ a\ ts)\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{in}\ \mathsf{RNode}\ a'\ [\mathsf{uAcc}'_\mathrm{r}\ (\oplus)\ (\otimes)\ t_i \mid i \in [1..\#ts]]\\
&\\
&\mathsf{dAcc}_\mathrm{r} :: (\beta \to \alpha \to \beta) \to \beta \to \mathsf{RTree}\ \alpha \to \mathsf{RTree}\ \beta\\
&\mathsf{dAcc}_\mathrm{r}\ g\ c\ (\mathsf{RNode}\ a\ ts) = \mathsf{RNode}\ c\ [\mathsf{dAcc}_\mathrm{r}\ g\ (g\ c\ a)\ t_i \mid i \in [1..\#ts]]\\
&\\
&\mathsf{rAcc}_\mathrm{r} :: (\alpha \to \alpha \to \alpha) \to \mathsf{RTree}\ \alpha \to \mathsf{RTree}\ \alpha\\
&\mathsf{rAcc}_\mathrm{r}\ (\oplus)\ (\mathsf{RNode}\ a\ ts) = \mathbf{let}\ rs = scan\ (\oplus)\ [root_\mathrm{r}\ t_i \mid i \in [1..\#ts]]\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{in}\ \mathsf{RNode}\ \iota_\oplus\ [setroot_\mathrm{r}\ (\mathsf{rAcc}_\mathrm{r}\ (\oplus)\ t_i)\ r_i \mid i \in [1..\#ts]]\\
&\\
&\mathsf{lAcc}_\mathrm{r} :: (\alpha \to \alpha \to \alpha) \to \mathsf{RTree}\ \alpha \to \mathsf{RTree}\ \alpha\\
&\mathsf{lAcc}_\mathrm{r}\ (\oplus)\ (\mathsf{RNode}\ a\ ts) = \mathbf{let}\ rs = scan'\ (\oplus)\ [root_\mathrm{r}\ t_i \mid i \in [1..\#ts]]\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{in}\ \mathsf{RNode}\ \iota_\oplus\ [setroot_\mathrm{r}\ (\mathsf{lAcc}_\mathrm{r}\ (\oplus)\ t_i)\ r_i \mid i \in [1..\#ts]]
\end{aligned}
$$

Figure 4.1. Definition of Rose-Tree Skeletons

Node-wise computations: map and zipwith

We define two node-wise computations in a similar way to those on binary trees. Rose-tree skeleton map_r takes a function k and a rose tree, and applies the function to each node. An example of the map_r skeleton is shown in Figure 4.2.

$$\begin{aligned}&\mathsf{map}_\mathrm{r} :: (\alpha \to \beta) \to \mathsf{RTree}\ \alpha \to \mathsf{RTree}\ \beta\\ &\mathsf{map}_\mathrm{r}\ k\ (\mathsf{RNode}\ a\ ts) = \mathsf{RNode}\ (k\ a)\ [\mathsf{map}_\mathrm{r}\ k\ t_i \mid i \in [1..\#ts]]\end{aligned}$$

Rose-tree skeleton $\mathsf{zipwith}_\mathrm{r}$ takes a function k and two rose trees of the same shape, and zips them up by applying the function to each pair of corresponding nodes. An example of the $\mathsf{zipwith}_\mathrm{r}$ skeleton is shown in Figure 4.3.

$$\begin{aligned}&\mathsf{zipwith}_\mathrm{r} :: (\alpha \to \alpha' \to \beta) \to \mathsf{RTree}\ \alpha \to \mathsf{RTree}\ \alpha' \to \mathsf{RTree}\ \beta\\ &\mathsf{zipwith}_\mathrm{r}\ k\ (\mathsf{RNode}\ a\ ts)\ (\mathsf{RNode}\ a'\ ts')\\ &\qquad = \mathsf{RNode}\ (k\ a\ a')\ [\mathsf{zipwith}_\mathrm{r}\ k\ t_i\ t'_i \mid i \in [1..\#ts]]\end{aligned}$$

Bottom-up computations: reduce and upwards accumulate

Rose-tree skeleton $\mathsf{reduce}_\mathrm{r}$ takes two operators $\oplus$ and $\otimes$ and a rose tree, and collapses the tree into a value in a bottom-up manner. We specify rose-tree skeleton $\mathsf{reduce}_\mathrm{r}$ with two operators: one for folding the list of children, and the other for merging the results of the children into the parent. By definition, the operator $\otimes$ that folds the siblings must be associative.

$$\begin{aligned}&\mathsf{reduce}_\mathrm{r} :: (\alpha \to \beta \to \beta) \to (\beta \to \beta \to \beta) \to \mathsf{RTree}\ \alpha \to \beta\\ &\mathsf{reduce}_\mathrm{r}\ (\oplus)\ (\otimes)\ (\mathsf{RNode}\ a\ ts) = a \oplus \textstyle\sum_{\otimes}[\mathsf{reduce}_\mathrm{r}\ (\oplus)\ (\otimes)\ t_i \mid i \in [1..\#ts]]\end{aligned}$$

To guarantee the existence of efficient parallel implementations, we impose some conditions on the two operators. A sufficient condition is: The operator $\otimes$ is associative and extended distributive over the operator $\oplus$.

In the definition above, we may assume that $a \oplus \iota_\otimes = a$ holds for any a. However, the equation does not hold in some cases. For example, if two operators are given as $\oplus = \times$ and $\otimes = +$ then $a \times 0 = 0$. For such operators, we define another rose-tree skeleton.

$$\begin{aligned}&\mathsf{reduce}'_\mathrm{r} :: (\alpha \to \alpha \to \alpha) \to (\alpha \to \alpha \to \alpha) \to \mathsf{RTree}\ \alpha \to \alpha\\ &\mathsf{reduce}'_\mathrm{r}\ (\oplus)\ (\otimes)\ (\mathsf{RNode}\ a\ [\,]) = a\\ &\mathsf{reduce}'_\mathrm{r}\ (\oplus)\ (\otimes)\ (\mathsf{RNode}\ a\ (t : ts)) = \mathbf{let}\ ts' = (t : ts)\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{in}\ a \oplus \textstyle\sum_{\otimes}[\mathsf{reduce}'_\mathrm{r}\ (\oplus)\ (\otimes)\ t'_i \mid i \in [1..\#ts']]\end{aligned}$$

In the definition, the $\mathsf{reduce}'_\mathrm{r}$ skeleton returns the value of the node for each leaf. A sufficient condition for parallel implementation is: The operator $\otimes$ is associative, and the operator $\oplus$ is associative and distributive over the operator $\otimes$. An example of the $\mathsf{reduce}'_\mathrm{r}$ skeleton is shown in Figure 4.4.

We then define another bottom-up computational pattern that returns a rose tree instead of a value. Rose-tree skeleton uAcc_r (upwards accumulate) takes two operators and a rose tree, and applies the $\mathsf{reduce}_\mathrm{r}$ skeletons for each subtree.

$$\begin{aligned}&\mathsf{uAcc}_\mathrm{r} :: (\alpha \to \beta \to \beta) \to (\beta \to \beta \to \beta) \to \mathsf{RTree}\ \alpha \to \mathsf{RTree}\ \beta\\ &\mathsf{uAcc}_\mathrm{r}\ (\oplus)\ (\otimes)\ (\mathsf{RNode}\ a\ ts) = \mathbf{let}\ a' = \mathsf{reduce}_\mathrm{r}\ (\oplus)\ (\otimes)\ (\mathsf{RNode}\ a\ ts)\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{in}\ \mathsf{RNode}\ a'\ [\mathsf{uAcc}_\mathrm{r}\ (\oplus)\ (\otimes)\ t_i \mid i \in [1..\#ts]]\end{aligned}$$

Rose-tree skeleton $\mathsf{uAcc}'_{\mathrm{r}}$ is a variant that applies the $\mathsf{reduce}'_{\mathrm{r}}$ skeleton for each subtree.

$$\begin{aligned}&\mathsf{uAcc}'_{\mathrm{r}} :: (\alpha \to \alpha \to \alpha) \to (\alpha \to \alpha \to \alpha) \to \mathsf{RTree}\ \alpha \to \mathsf{RTree}\ \alpha\\&\mathsf{uAcc}'_{\mathrm{r}}\ (\oplus)\ (\otimes)\ (\mathsf{RNode}\ a\ ts) = \mathbf{let}\ a' = \mathsf{reduce}'_{\mathrm{r}}\ (\oplus)\ (\otimes)\ (\mathsf{RNode}\ a\ ts)\\&\qquad\qquad\qquad\qquad\qquad\qquad\mathbf{in}\ \mathsf{RNode}\ a'\ [\mathsf{uAcc}'_{\mathrm{r}}\ (\oplus)\ (\otimes)\ t_i \mid i \in [1..\#ts]]\end{aligned}$$

An example of the $\mathsf{uAcc}_{\mathrm{r}}$ skeleton is shown in Figure 4.5.

These skeletons return a rose-tree of the same shape as the input. For efficient parallel implementations the $\mathsf{uAcc}_{\mathrm{r}}$ and $\mathsf{uAcc}'_{\mathrm{r}}$ skeletons require the same conditions as the $\mathsf{reduce}_{\mathrm{r}}$ and $\mathsf{reduce}'_{\mathrm{r}}$ skeletons do, respectively.

Top-down computations: downwards accumulate

Rose-tree skeleton $\mathsf{dAcc}_{\mathrm{r}}$ (downwards accumulate) is a top-down computational pattern. With an analogy to the $\mathsf{dAcc}_{\mathrm{b}}$ skeleton, the $\mathsf{dAcc}_{\mathrm{r}}$ skeleton takes an accumulative parameter c and proceeds the computation in a top-down manner by updating the accumulative parameter on each node. Since an internal node of a rose tree may have an arbitrary number of children, it is not realistic to prepare functions for updating the accumulative parameter individually for each children. Therefore, in our definition the $\mathsf{dAcc}_{\mathrm{r}}$ skeleton takes only one function that is uniformly used in updating accumulative parameter for all the children. An example of the $\mathsf{dAcc}_{\mathrm{r}}$ skeleton is shown in Figure 4.6.

$$\begin{aligned}&\mathsf{dAcc}_{\mathrm{r}} :: (\beta \to \alpha \to \beta) \to \beta \to \mathsf{RTree}\ \alpha \to \mathsf{RTree}\ \beta\\&\mathsf{dAcc}_{\mathrm{r}}\ g\ c\ (\mathsf{RNode}\ a\ ts) = \mathsf{RNode}\ c\ [\mathsf{dAcc}_{\mathrm{r}}\ g\ (g\ c\ a)\ t_i \mid i \in [1..\#ts]]\end{aligned}$$

The condition for efficient parallel implementations is the existence of auxiliary functions ϕ, ψ_u and ψ_d satisfying the following two equations.

$$\begin{aligned}&g\ c\ n = \psi_d\ c\ (\phi\ n)\\&\psi_d\ (\psi_d\ c\ n)\ m = \psi_d\ c\ (\psi_u\ n\ m)\end{aligned}$$

This definition is rather simple, but we show that we can obtain enough expressiveness for top-down computations with the following intra-sibling computations.

Intra-siblings computations: rightwards accumulate and leftwards accumulate

The five rose-tree skeletons above are extensions of the five binary-tree skeletons. We add two skeletons that are specific to rose trees.

Rose-tree skeleton $\mathsf{rAcc}_{\mathrm{r}}$ (rightwards accumulate) takes an associative operator $\oplus$ and a rose tree, and applies the *scan* operation to each list of siblings. An example of the $\mathsf{rAcc}_{\mathrm{r}}$ skeleton is shown in Figure 4.7.

$$\begin{aligned}&\mathsf{rAcc}_{\mathrm{r}} :: (\alpha \to \alpha \to \alpha) \to \mathsf{RTree}\ \alpha \to \mathsf{RTree}\ \alpha\\&\mathsf{rAcc}_{\mathrm{r}}\ (\oplus)\ (\mathsf{RNode}\ a\ ts) = \mathbf{let}\ rs = scan\ (\oplus)\ [root_{\mathrm{r}}\ t_i \mid i \in [1..\#ts]]\\&\qquad\qquad\qquad\qquad\qquad\qquad\mathbf{in}\ \mathsf{RNode}\ \iota_{\oplus}\ [setroot_{\mathrm{r}}\ (\mathsf{rAcc}_{\mathrm{r}}\ (\oplus)\ t_i)\ r_i \mid i \in [1..\#ts]]\end{aligned}$$

Here, r_i denotes the ith element of the list rs.

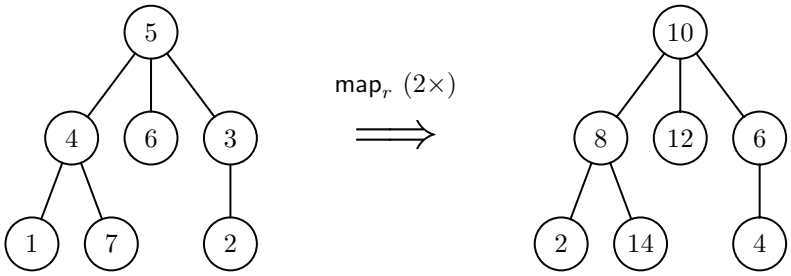


Figure 4.2. An example of the map_r skeleton. This example computes the doubled value for each node.

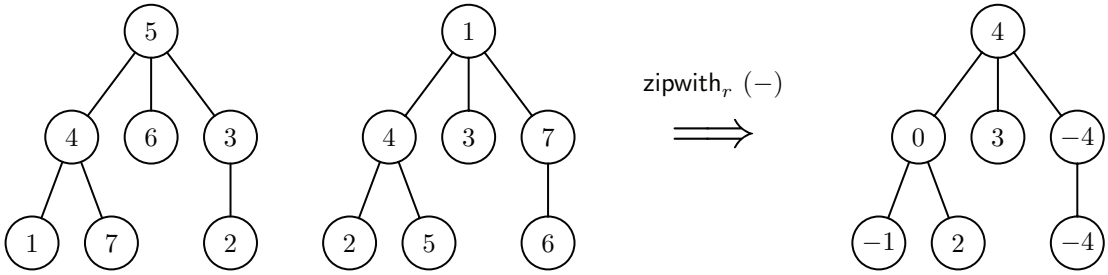


Figure 4.3. An example of the $\mathsf{zipwith}_\mathrm{r}$ skeleton. This example computes the difference for each pair of corresponding nodes.

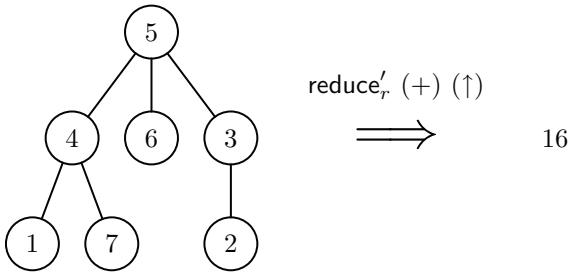


Figure 4.4. An example of the $\mathsf{reduce}'_\mathrm{r}$ skeleton. This example computes the maximum among sums of values on paths from the root.

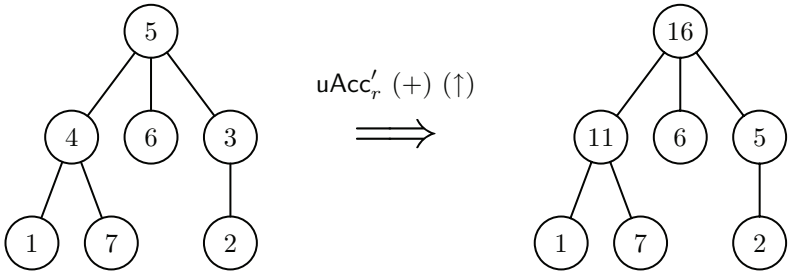


Figure 4.5. An example of the $\mathsf{uAcc}'_\mathrm{r}$ skeleton. This example uses the same operators as the example of the $\mathsf{reduce}'_\mathrm{r}$ skeleton.

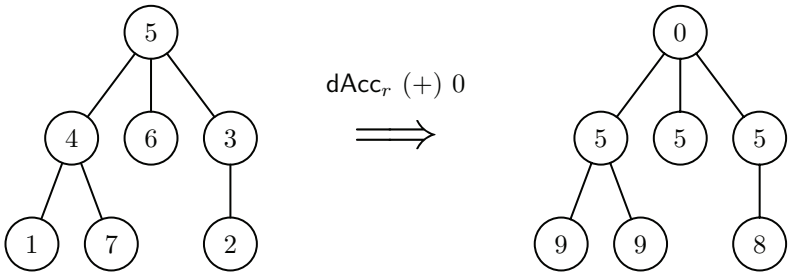


Figure 4.6. An example of the $\mathsf{dAcc_r}$ skeleton. This example computes for each node the sum from the root node to the parent node.

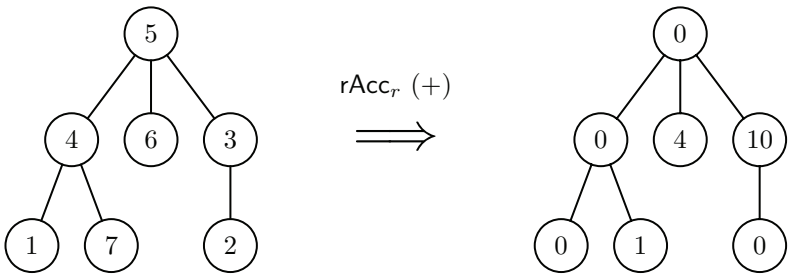


Figure 4.7. An example of the $\mathsf{lAcc_r}$ skeleton. This example computes accumulated sums from left to right for each set of siblings.

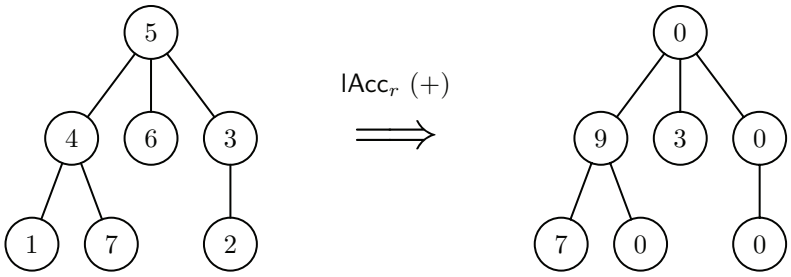


Figure 4.8. An example of the $\mathsf{rAcc_r}$ skeleton. This example computes accumulated sums from left to right for each set of siblings.

Rose-tree skeleton $\mathsf{lAcc_r}$ (leftwards accumulate) is symmetric to the $\mathsf{rAcc_r}$ skeleton. This skeleton applies the reversed *scan* operation to each list of siblings. An example of the $\mathsf{lAcc_r}$ skeleton is shown in Figure 4.8.

$$\begin{aligned}
&\mathsf{lAcc_r} :: (\alpha \to \alpha \to \alpha) \to \mathsf{RTree}\ \alpha \to \mathsf{RTree}\ \alpha \\
&\mathsf{lAcc_r}\ (\oplus)\ (\mathsf{RNode}\ a\ ts) = \mathbf{let}\ rs = scan'\ (\oplus)\ [root_{\mathrm{r}}\ t_i \mid i \in [1..\#ts]] \\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{in}\ \mathsf{RNode}\ \iota_{\oplus}\ [setroot_{\mathrm{r}}\ (\mathsf{lAcc_r}\ (\oplus)\ t_i)\ r_i \mid i \in [1..\#ts]]
\end{aligned}$$

4.2.2 Example: Prefix Numbering Problem on Rose Trees

To see how we can use these rose-tree skeletons for manipulating general trees, we develop a skeletal program that solves the prefix numbering problem on rose trees. In this problem, given a rose tree we want to assign for each node a number starting at zero in the order of the prefix traversal.

We can give a recursive algorithm pre_{r} for the prefix numbering as follows. The recursive algorithm is based on two facts of the prefix numbering. The value of a leftmost child is larger by one than that of its parent. The value of one child is larger than its left sibling by the size of the left-sibling's subtree. Auxiliary function $size_{\mathrm{r}}$ counts the number of nodes in the given rose tree.

$$\begin{aligned}
&pre_{\mathrm{r}}\ t = pre'_{\mathrm{r}}\ 0\ t \\
&pre'_{\mathrm{r}}\ c\ (\mathsf{RNode}\ a\ ts) = \mathsf{RNode}\ c\ [pre'_{\mathrm{r}}\ (c + 1 + l_i)\ t_i \mid i \in [1..\#ts]] \\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{where}\ l_i = \textstyle\sum_{+}[size_{\mathrm{r}}\ t_j \mid j \in [1..i-1]] \\
&size_{\mathrm{r}}\ (\mathsf{RNode}\ a\ ts) = 1 + \textstyle\sum_{+}[size_{\mathrm{r}}\ t_i \mid i \in [1..\#ts]]
\end{aligned}$$

The recursive program is very similar to the recursive program for the prefix numbering of binary trees. A difference is that the accumulative parameter c is updated by a value l_i, which is computed from the values of left siblings. Note that we can compute the value for each node by the $\mathsf{rAcc_r}$ skeleton.

From the recursive program above, we can give the following skeletal program that consists of three steps. We first compute the result of the $size_{\mathrm{r}}$ function for each node using the $\mathsf{map_r}$ and $\mathsf{uAcc_r}$ skeletons. We then apply the $\mathsf{rAcc_r}$ skeleton to accumulate the results from left to right among each list of siblings. Finally, we compute the prefix numbering by a top-down computation with the $\mathsf{map_r}$, $\mathsf{dAcc_r}$, and $\mathsf{zipwith_r}$ skeletons.

$$\begin{aligned}
pre_{\mathrm{r}}\ t = \mathbf{let}\ &st = \mathsf{uAcc_r}\ (+)\ (+)\ (\mathsf{map_r}\ (\lambda x.1)\ t) \\
&lt\ = \mathsf{rAcc_r}\ (+)\ st \\
\mathbf{in}\ &\mathsf{zipwith_r}\ (+)\ (\mathsf{dAcc_r}\ (+)\ 0\ (\mathsf{map_r}\ (1+)\ lt))\ lt
\end{aligned}$$

Since the operator used in the program is $+$, we can easily verify that the conditions for efficient parallel implementations are satisfied.

In section 5.1 we will give a systematic method of deriving skeletal programs from recursive functions.

4.3 Parallelizing Rose-Tree Skeletons with Binary-Tree Skeletons

In this section, we develop a parallel implementation of the rose-tree skeletons. The main idea is that we implement the rose-tree skeletons with the parallel binary-tree skeletons on a binary-tree representation of rose-trees. Since we can implement the binary-tree skeletons efficiently in parallel, our implementation of the rose-tree skeletons also run efficiently in parallel.

4.3.1 Binary-Tree Representation of Rose Trees

Many researchers have studied parallel manipulations of binary trees based on tree contraction algorithms [2, 8, 95, 98], and we have shown a novel implementation of tree manipulations on ternary-tree representation in Chapter 3. We can utilize these parallel implementations, if we represent rose trees in the form of binary trees.

Here, we adopt a binary-tree representation of rose trees shown in Figure 4.9. This binary-tree representation is a widely-used one in sequential programming [36] especially for manipulating XML trees, but it has been rarely used for parallel programming. There are other binary-tree representations of rose trees [2,35,89,119]. In this binary-tree representation, every internal node comes from a node in the original rose tree, and all leaves are dummy nodes. The left child of a node in the binary-tree representation indicates the leftmost child in the rose tree; the right child of a node in the binary-tree representation indicates the next right sibling in the rose tree.

To formalize the binary-tree representation, we define function *rt2bt* to transform a rose tree into its binary-tree representation. This function uses auxiliary functions that transform a forest (a list of rose trees) into a binary tree. Note that the dummy value on a leaf is given by a don't-care value $\_$ in the following function.

$$
\begin{aligned}
&rt2bt :: \mathsf{RTree}\ \alpha \to \mathsf{BTree}\ \_\ \alpha \\
&rt2bt\ t &&= rt2bt'\ t\ [\,] \\
&rt2bt'\ (\mathsf{RNode}\ a\ ts)\ ss &&= \mathsf{BNode}\ (rt2bt''\ ts)\ a\ (rt2bt''\ ss) \\
&rt2bt''\ [\,] &&= \mathsf{BLeaf}\ \_ \\
&rt2bt''\ (t : ts) &&= rt2bt'\ t\ ts
\end{aligned}
$$

The inverse function *bt2rt*, which restores a rose tree from its binary-tree representation, is defined as follows. The auxiliary function *bt2rt* transforms a binary tree to a forest, and we pick the resulting rose tree out by function *head*. The value of a leaf is omitted.

$$
\begin{aligned}
&bt2rt :: \mathsf{BTree}\ \_\ \alpha \to \mathsf{RTree}\ \alpha \\
&bt2rt\ t &&= head\ (bt2rt'\ t) \\
&bt2rt'\ (\mathsf{BNode}\ l\ b\ r) &&= (\mathsf{RNode}\ b\ (bt2rt'\ l)) : bt2rt'\ r \\
&bt2rt'\ (\mathsf{BLeaf}\ \_) &&= [\,]
\end{aligned}
$$

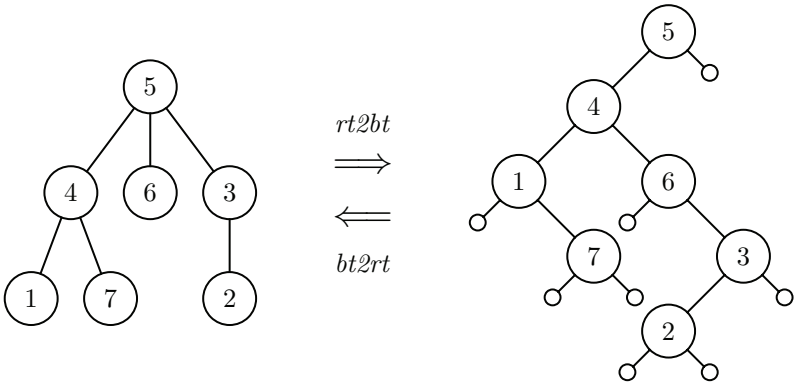


Figure 4.9. Binary-tree representation of rose trees.

Here, $bt2rt \circ rt2bt = id$ always holds, but $rt2bt \circ bt2rt$ may change the values of dummy nodes. It is worth noting that if we ignore the values of dummy nodes then we can treat the composition $rt2bt \circ bt2rt$ as the identity function.

We remark on the size of the binary-tree representation.

Lemma 4.4 *Let N be the number of nodes of the original rose tree, then the number of nodes in the binary-tree representation is $2N + 1$.*

Proof. All the internal nodes in the binary-tree representation correspond to the nodes in the rose tree. In general, the number of leaves is given by the number of internal nodes plus one. It follows from these facts that the lemma holds. □

This lemma guarantees the asymptotic cost with respect to the size of rose trees when we utilize parallel binary-tree skeletons on the binary-tree representation.

In the following sections, we implement the rose-tree skeletons on the binary-tree representation using parallel binary-tree skeletons. Generally speaking, our implementation of a rose-tree skeleton consists of three steps: firstly function *rt2bt* transforms a rose tree to its binary-tree representation; then we apply binary-tree skeletons to the binary-tree representation; and finally function *bt2rt* restores the rose-tree structure if necessary. Note that if two rose-tree skeletons are called successively we can remove the transformations from and to the binary-tree representation.

4.3.2 Parallelizing Node-wise Computations

Since every node in a rose tree is an internal node in the binary-tree representation and there are no dependencies in the computation of the map_r skeleton, we implement the map_r skeleton by simply using the map_b skeleton to apply the function to each internal node. Here, we do not care about the parameter function for leaves.

$$\mathsf{map}_\mathrm{r}\ k = bt2rt \circ (\mathsf{map}_\mathrm{b}\ \_\ k) \circ rt2bt$$

Similarly, we implement the $\mathsf{zipwith}_\mathrm{r}$ skeleton using the $\mathsf{zipwith}_\mathrm{b}$ skeleton.

$$\mathsf{zipwith}_\mathrm{r}\ k\ t\ t' = bt2rt\ (\mathsf{zipwith}_\mathrm{b}\ \_\ k\ (rt2bt\ t)\ (rt2bt\ t'))$$

4.3.3 Parallelizing Bottom-up Computations

The bottom-up computation on a rose tree can be mapped to a bottom-up computation on the binary-tree representation. We implement the $\mathsf{reduce_r}$ skeleton using the $\mathsf{map_b}$ and $\mathsf{reduce_b}$ skeletons as follows.

$$\begin{aligned}\mathsf{reduce_r}\ (\oplus)\ (\otimes) = (\mathsf{reduce_b}\ k) \circ (\mathsf{map_b}\ (\lambda x.\iota_\otimes)\ id) \circ rt2bt\\ \textbf{where}\ k\ l\ b\ r = (b \oplus l) \otimes r\end{aligned}$$

Note that we do not apply the *bt2rt* function since the $\mathsf{reduce_r}$ skeleton returns a value that is not a tree.

For the parallel implementation, the function k above should satisfy the condition of the $\mathsf{reduce_b}$ skeleton. As stated in Lemma 3.7, we can always derive functions for parallel implementation of the $\mathsf{reduce_b}$ skeleton if we do not care about the cost. In the following we show that we can implement the $\mathsf{reduce_r}$ skeleton efficiently in parallel.

Sufficient conditions for an efficient implementation are the following two: the operator $\otimes$ is associative and extended-distributive over $\oplus$ with its unit $\iota_\otimes$, and the operator $\oplus$ has the left unit $\iota_\oplus$. Let p_1, p_2, and p_3 be the characteristic functions of the extended-extended distributivity. For the function k of the $\mathsf{reduce_b}$ skeleton, we choose the following form of functions defined with four parameters.

$$f_{(a,b,c,d)} = \lambda x\ y.a \oplus (b \otimes (c \oplus x) \otimes y \otimes d)$$

We can confirm that this functional form is closed through tree contractions as the following calculations show.

$$\begin{aligned}&\phi\ n = \lambda x\ y.(n \oplus x) \otimes y\\ &\quad = \quad \{\text{introducing units } \iota_\oplus \text{ and } \iota_\otimes\}\\ &\quad\ \ \lambda x\ y.\iota_\oplus \oplus (\iota_\otimes \otimes (n \oplus x) \otimes y \otimes \iota_\otimes)\\ &\quad = \quad \{\text{folding to the functional form}\}\\ &\quad\ \ f_{(\iota_\oplus,\iota_\otimes,n,\iota_\oplus)}\end{aligned}$$

$$\begin{aligned}&\psi_l\ f_{(a_l,b_l,c_l,d_l)}\ f_{(a_n,b_n,c_n,d_n)}\ r\\ &\quad = \quad \{\text{unfolding functions}\}\\ &\quad\ \ \lambda x\ y.a_n \oplus (b_n \otimes (c_n \oplus (a_l \oplus (b_l \otimes (c_l \oplus x) \otimes y \otimes d_l))) \otimes r \otimes d)\\ &\quad = \quad \{\text{folding to the form of extended distributivity}\}\\ &\quad\ \ \lambda x\ y.((\lambda z.a_n \oplus (b_n \otimes z \otimes (r \otimes d))) \circ (\lambda z.c_n \oplus (\iota_\otimes \otimes z \otimes \iota_\otimes))\\ &\qquad\qquad \circ (\lambda z.a_l \oplus (b_l \otimes z \otimes d_l)))\ ((c_l \oplus x) \otimes y)\\ &\quad = \quad \{\text{extended distributivity}\}\\ &\quad\ \ \lambda x\ y.(\lambda z.a' \oplus (b' \otimes z \otimes c'))\ ((c_l \oplus x) \otimes y)\\ &\quad\ \ \textbf{where}\ (a', b', c') = (p_1 \vartriangle p_2 \vartriangle p_3)\ (a_n, b_n, (r \otimes d), a'', b'', c'')\\ &\qquad\qquad (a'', b'', c'') = (p_1 \vartriangle p_2 \vartriangle p_3)\ (c_n, \iota_\otimes, \iota_\otimes, a_l, b_l, d_l)\\ &\quad = \quad \{\text{unfolding and folding to the functional form}\}\\ &\quad\ \ \lambda x\ y.f_{(a',b',c_l,c')}\ x\ y\end{aligned}$$

$$
\begin{aligned}
&\psi_r\ l\ f_{(a_n,b_n,c_n,d_n)}\ f_{(a_r,b_r,c_r,d_r)}\\
&\quad = \quad \{\text{unfolding functions}\}\\
&\qquad \lambda x\ y.a_n \oplus (b_n \otimes (c_n \oplus l) \otimes (a_r \oplus (b_r \otimes (c_r \oplus x) \otimes y \otimes d_r)) \otimes d_n)\\
&\quad = \quad \{\text{folding to the form of extended distributivity}\}\\
&\qquad \lambda x\ y.((\lambda z.a_n \oplus ((b_n \otimes (c_n \oplus l)) \otimes z \otimes d_n)) \circ (\lambda z.a_r \oplus (b_r \otimes z \otimes d_r)))\\
&\qquad\qquad\qquad ((c_r \otimes x) \otimes y)\\
&\quad = \quad \{\text{extended distributivity}\}\\
&\qquad \lambda x\ y.(\lambda z.a' \oplus (b' \otimes z \otimes c'))\ ((c_r \otimes x) \otimes y)\\
&\qquad \textbf{where}\ (a', b', c') = (p_1 \vartriangle p_2 \vartriangle p_3)\ (a_n, b_n \otimes (c_n \oplus l), d_n, a_r, b_r, d_r)\\
&\quad = \quad \{\text{unfolding and folding to the functional form}\}\\
&\qquad \lambda x\ y.f_{(a',b',c_r,c')}\ x\ y
\end{aligned}
$$

Based on the calculations above, we can derive the auxiliary functions for the $\mathsf{reduce}_\mathrm{b}$ skeleton. Therefore, parallel implementation of the $\mathsf{reduce}_\mathrm{r}$ skeleton is given as follows.

$$
\begin{aligned}
&\mathsf{reduce}_\mathrm{r}\ (\oplus)\ (\otimes)\\
&\quad = (\mathsf{reduce}_\mathrm{b}\ \langle \phi, \psi_n, \psi_l, \psi_r \rangle_\mathrm{u}) \circ (\mathsf{map}_\mathrm{b}\ (\lambda x.\iota_\otimes)\ id) \circ rt2bt\\
&\qquad \textbf{where}\ \phi\ n = (\iota_\oplus, \iota_\otimes, n, \iota_\oplus)\\
&\qquad\qquad \phi_n\ l\ (a_n, b_n, c_n, d_n)\ r = a_n \oplus (b_n \otimes (c_n \oplus l) \otimes r \otimes d)\\
&\qquad\qquad \psi_l\ (a_l, b_l, c_l, d_l)\ (a_n, b_n, c_n, d_n)\ r\\
&\qquad\qquad\quad = \textbf{let}\ (a'', b'', c'') = (p_1 \vartriangle p_2 \vartriangle p_3)\ (c_n, \iota_\otimes, \iota_\otimes, a_l, b_l, d_l)\\
&\qquad\qquad\qquad\qquad (a', b', c') = (p_1 \vartriangle p_2 \vartriangle p_3)\ (a_n, b_n, (r \otimes d), a'', b'', c'')\\
&\qquad\qquad\qquad\ \textbf{in}\ (a', b', c_l, c')\\
&\qquad\qquad \psi_r\ l\ (a_n, b_n, c_n, d_n)\ (a_r, b_r, c_r, d_r)\\
&\qquad\qquad\quad = \textbf{let}\ (a', b', c') = (p_1 \vartriangle p_2 \vartriangle p_3)\ (a_n, b_n \otimes (c_n \oplus l), d_n, a_r, b_r, d_r)\\
&\qquad\qquad\qquad\ \textbf{in}\ (a', b', c_r, c')
\end{aligned}
$$

The $\mathsf{reduce}'_\mathrm{r}$ skeleton deals with the leaves and the internal nodes in a different way. Therefore, to implement the $\mathsf{reduce}'_\mathrm{r}$ skeleton, we mark the internal nodes whether or not they are from leaves in the original rose tree, by using the map_b, $\mathsf{getchl}_\mathrm{b}$ and $\mathsf{zipwith}_\mathrm{b}$ skeletons. We implement the $\mathsf{reduce}'_\mathrm{r}$ skeleton as follows.

$$
\begin{aligned}
\mathsf{reduce}'_\mathrm{r}\ (\oplus)\ (\otimes)\ t = \textbf{let}\ &bt\ = rt2bt\ t\\
&mt = \mathsf{getchl}_\mathrm{b}\ \_\ (\mathsf{map}_\mathrm{b}\ (\lambda x.\mathsf{True})\ (\lambda x.\mathsf{False})\ t)\\
\textbf{in}\ &\mathsf{reduce}_\mathrm{b}\ k\ (\mathsf{zipwith}_\mathrm{b}\ (\lambda x\ y.\iota_\otimes)\ (,)\ mt\ bt)\\
\textbf{where}\ &k\ l\ (\mathsf{True}, n)\ r\ = n \otimes r\\
&k\ l\ (\mathsf{False}, n)\ r = (n \oplus l) \otimes r
\end{aligned}
$$

For the parallel implementation of the $\mathsf{reduce}'_\mathrm{r}$ skeleton, a sufficient condition is: the operator $\oplus$ is associative with its unit $\iota_\oplus$ and distributive over $\otimes$, and the operator $\otimes$ is associative with its unit $\iota_\otimes$. Under this condition, we have $\iota_\otimes \oplus a = \iota_\otimes$ for any a. With this fact, we design the functional form for a parallel implementation of the $\mathsf{reduce}_\mathrm{b}$ skeleton as follows.

$$
f_{(a,b,c,d)} = \lambda x\ y.a \otimes (b \oplus x) \otimes (c \oplus x) \otimes d
$$

We confirm that this form is closed through tree contractions by the following calculations.

$$
\begin{aligned}
\phi\ (\mathsf{True}, n) &= \lambda x\ y.n \otimes y \\
&= \quad \{\text{introducing units } \iota_\oplus \text{ and } \iota_\otimes\} \\
&\quad \lambda x\ y.n \otimes (\iota_\otimes \oplus x) \otimes (\iota_\oplus \oplus y) \otimes \iota_\otimes \\
&= \quad \{\text{folding to the functional form}\} \\
&\quad f_{(n,\iota_\otimes,\iota_\oplus,\iota_\otimes)}
\end{aligned}
$$

$$
\begin{aligned}
\phi\ (\mathsf{False}, n) &= \lambda x\ y.(n \oplus x) \otimes y \\
&= \quad \{\text{introducing units } \iota_\oplus \text{ and } \iota_\otimes\} \\
&\quad \lambda x\ y.\iota_\otimes \otimes (n \oplus x) \otimes (\iota_\oplus \oplus y) \otimes \iota_\otimes \\
&= \quad \{\text{folding to the functional form}\} \\
&\quad f_{(\iota_\otimes,n,\iota_\oplus,\iota_\otimes)}
\end{aligned}
$$

$$
\begin{aligned}
&\psi_l\ f_{(a_l,b_l,c_l,d_l)}\ f_{(a_n,b_n,c_n,d_n)}\ r \\
&\quad = \quad \{\text{unfolding functions}\} \\
&\qquad \lambda x\ y.a_n \otimes (b_n \oplus (a_l \otimes (b_l \oplus x) \otimes (c_l \oplus y) \otimes d_l)) \otimes (c_n \oplus r) \otimes d_n \\
&\quad = \quad \{\text{the associative law and the distributive law}\} \\
&\qquad \lambda x\ y.(a_n \otimes (b_n \oplus a_l)) \otimes ((b_n \oplus b_l) \oplus x) \otimes ((b_n \oplus c_l) \oplus y) \\
&\qquad\qquad \otimes ((b_n \oplus d_l) \otimes (c_n \oplus r) \otimes d_n) \\
&\quad = \quad \{\text{folding to the functional form}\} \\
&\qquad f_{(a_n \otimes (b_n \oplus a_l),\ b_n \oplus b_l,\ b_n \oplus c_l,\ (b_n \oplus d_l) \otimes (c_n \oplus r) \otimes d_n)}
\end{aligned}
$$

$$
\begin{aligned}
&\psi_r\ l\ f_{(a_n,b_n,c_n,d_n)}\ f_{(a_r,b_r,c_r,d_r)} \\
&\quad = \quad \{\text{unfolding functions}\} \\
&\qquad \lambda x\ y.a_n \otimes (b_n \oplus l) \otimes (c_n \oplus (a_r \otimes (b_r \oplus x) \otimes (c_r \oplus y) \otimes d_r)) \otimes d_n \\
&\quad = \quad \{\text{the associative law and the distributive law}\} \\
&\qquad \lambda x\ y.(a_n \otimes (b_n \oplus l) \otimes (c_n \oplus a_r)) \otimes ((c_n \oplus b_r) \oplus x) \otimes ((c_n \oplus c_r) \oplus y) \\
&\qquad\qquad \otimes ((c_n \oplus d_r) \otimes d_n) \\
&\quad = \quad \{\text{folding to the functional form}\} \\
&\qquad f_{(a_n \otimes (b_n \oplus l) \otimes (c_n \oplus a_r),\ c_n \oplus b_r,\ c_n \oplus c_r,\ (c_n \oplus d_r) \otimes d_n)}
\end{aligned}
$$

Based on the calculations above, we can implement the $\mathsf{reduce}'_{\mathrm{r}}$ skeleton efficiently in parallel.

$$
\begin{aligned}
&\mathsf{reduce}'_{\mathrm{r}}\ (\oplus)\ (\otimes)\ t \\
&\quad = \mathbf{let}\ bt\ = rt2bt\ t \\
&\qquad\quad mt = \mathsf{getchl}_{\mathrm{b}}\ \_\ (\mathsf{map}_{\mathrm{b}}\ (\lambda x.\mathsf{True})\ (\lambda x.\mathsf{False})\ t) \\
&\qquad \mathbf{in}\ \mathsf{reduce}_{\mathrm{b}}\ \langle \phi, \psi_n, \psi_l, \psi_r \rangle_{\mathrm{u}}\ (\mathsf{zipwith}_{\mathrm{b}}\ (\lambda x\ y.\iota_\otimes)\ (,)\ mt\ bt) \\
&\qquad \mathbf{where}\ \phi\ (\mathsf{True}, n) = (n, \iota_\otimes, \iota_\oplus, \iota_\otimes) \\
&\qquad\qquad\quad \phi\ (\mathsf{False}, n) = (\iota_\otimes, n, \iota_\oplus, \iota_\otimes) \\
&\qquad\qquad\quad \psi_n\ l\ (a_n, b_n, c_n, d_n)\ r = a_n \otimes (b_n \oplus l) \otimes (c_n \oplus r) \otimes d \\
&\qquad\qquad\quad \psi_l\ (a_l, b_l, c_l, d_l)\ (a_n, b_n, c_n, d_n)\ r \\
&\qquad\qquad\qquad = (a_n \otimes (b_n \oplus a_l),\ b_n \oplus b_l,\ b_n \oplus c_l,\ (b_n \oplus d_l) \otimes (c_n \oplus r) \otimes d_n) \\
&\qquad\qquad\quad \psi_r\ l\ (a_n, b_n, c_n, d_n)\ (a_r, b_r, c_r, d_r) \\
&\qquad\qquad\qquad = (a_n \otimes (b_n \oplus l) \otimes (c_n \oplus a_r),\ c_n \oplus b_r,\ c_n \oplus c_r,\ (c_n \oplus d_r) \otimes d_n)
\end{aligned}
$$

Now we turn to develop parallel implementations of the $\mathsf{uAcc}_{\mathrm{r}}$ and $\mathsf{uAcc}'_{\mathrm{r}}$ skeletons. Since the $\mathsf{uAcc}_{\mathrm{r}}$ skeleton is similar to the $\mathsf{reduce}_{\mathrm{r}}$ skeleton, first let us consider applying

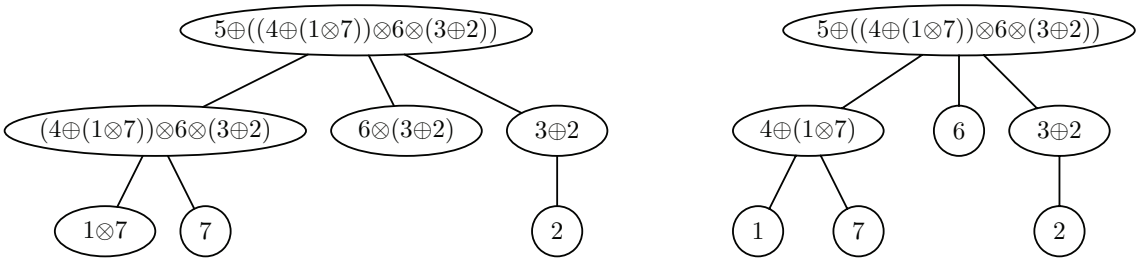


Figure 4.10. Left: the result of $bt2rt \circ (\mathsf{uAcc_b}\ (\lambda b\ l\ r.(b \oplus l) \otimes r)) \circ (\mathsf{map_b}\ (\lambda x.\iota_\otimes)\ id) \circ rt2bt$. Right: the desired result for $\mathsf{uAcc_r}\ (\oplus)\ (\otimes)$.

the $\mathsf{uAcc_b}$ skeleton instead of the $\mathsf{reduce_b}$ skeleton. In the following program, let t be a input rose tree, $\oplus$ and $\otimes$ be operators passed to the $\mathsf{uAcc_r}$ skeleton.

$$\begin{aligned} bt' = {} & \mathsf{uAcc_b}\ k\ (\mathsf{map_b}\ (\lambda x.\iota_\otimes)\ id\ (rt2bt\ t)) \\ & \textbf{where}\ k\ l\ b\ r = (b \oplus l) \otimes r \end{aligned}$$

Unfortunately, the result is not what we want for the $\mathsf{uAcc_r}$ skeleton as shown in Figure 4.10. Since in the binary-tree representation an internal node has a right-child that was the next right sibling in the original rose tree, the result value includes the siblings' values. We need to compute again the desired result for each internal node from the original value and the value of left child in bt'. We can implement this computation using the $\mathsf{zipwith_b}$ and $\mathsf{getchl_b}$ skeletons.

$$bt'' = \mathsf{zipwith_b}\ \_\ (\oplus)\ bt\ (\mathsf{getchl_b}\ \_\ bt')$$

Therefore, an implementation of the $\mathsf{uAcc_r}$ skeleton on the binary-tree representation is given as follows. Note that the condition and the four auxiliary functions for the $\mathsf{uAcc_b}$ skeleton, ϕ, ψ_n, ψ_l, and ψ_r, are the same as those derived for implementing the $\mathsf{reduce_r}$ skeleton.

$$\begin{aligned} \mathsf{uAcc_r}\ (\oplus)\ (\otimes)\ t = {} & \textbf{let}\ bt = rt2bt\ t \\ & \qquad bt' = \mathsf{uAcc_b}\ \langle \phi, \psi_n, \psi_l, \psi_r \rangle_\mathrm{u}\ (\mathsf{map_b}\ (\lambda x.\iota_\otimes)\ id\ bt) \\ & \textbf{in}\ bt2rt\ (\mathsf{zipwith_b}\ \_\ (\oplus)\ bt\ (\mathsf{getchl_b}\ \_\ bt')) \end{aligned}$$

Similarly, the $\mathsf{uAcc'_r}$ can be implemented based on the implementation of the $\mathsf{reduce'_r}$ skeleton. An implementation of the $\mathsf{uAcc'_r}$ skeleton is given as follows, where the auxiliary functions ϕ, ψ_n, ψ_l and ψ_r are those derived for the implementation of the $\mathsf{reduce'_r}$ skeleton.

$$\begin{aligned} \mathsf{uAcc'_r}\ (\oplus)\ (\otimes)\ t = {} & \textbf{let}\ bt = rt2bt\ t \\ & \qquad mt = \mathsf{getchl_b}\ \_\ (\mathsf{map_b}\ (\lambda x.\mathsf{True})\ (\lambda x.\mathsf{False})\ t) \\ & \qquad bt' = \mathsf{uAcc_b}\ \langle \phi, \psi_n, \psi_l, \psi_r \rangle_\mathrm{u}\ (\mathsf{zipwith_b}\ (\lambda x\ y.\iota_\otimes)\ (,)\ mt\ bt) \\ & \textbf{in}\ bt2rt\ (\mathsf{zipwith_b}\ \_\ (\odot)\ (\mathsf{zipwith_b}\ \_\ (,)\ mt\ bt)\ bt') \\ & \textbf{where}\ (\mathsf{True}, n) \odot l = n \\ & \qquad\quad (\mathsf{False}, n) \odot l = n \oplus l \end{aligned}$$

4.3.4 Parallelizing Top-down Computations

The top-down computation on rose trees can be mapped to a top-down computation on the binary-tree representation. We thus implement the $\mathsf{dAcc_r}$ skeleton using the $\mathsf{dAcc_b}$ skeleton as follows.

$$\begin{aligned}&\mathsf{dAcc_r}\ g\ c = \mathit{bt2rt} \circ \mathsf{dAcc_b}\ g_l\ g_r\ c \circ \mathit{rt2bt}\\ &\quad\ \textbf{where}\ g_l\ c\ n = g\ c\ n\\ &\qquad\qquad g_r\ c\ n = c\end{aligned}$$

The parameter functions g_l and g_r require some conditions for efficient parallel implementations. In the following, we show that we can define the auxiliary functions under the sufficient condition introduced in the specification. Let ϕ, ψ_d, and ψ_u be functions satisfying the following equations with g.

$$\begin{aligned}&g\ c\ n = \psi_d\ c\ (\phi\ n)\\ &\psi_d\ (\psi_d\ c\ n)\ m = \psi_d\ c\ (\psi_u\ n\ m)\end{aligned}$$

We consider the following form of functions defined with two parameters. The True case corresponds to the function g_l and the False case corresponds to the function g_r.

$$\begin{aligned}f_{(\mathsf{True},n)} &= \lambda c.\psi_d\ c\ n\\ f_{(\mathsf{False},\_)} &= \lambda c.c\end{aligned}$$

We can confirm that the functional form is closed under function composition. In the following calculations, function ψ'_u is one of the auxiliary functions for the $\mathsf{dAcc_b}$ skeleton.

$$\begin{aligned}\psi'_u\ f_{(\mathsf{True},n)}\ f_{(\mathsf{True},m)} &= \lambda c.\psi_d\ (\psi_d\ c\ n)\ m\\ &= \quad \{\text{the second equation of the condition from left to right}\}\\ &\quad\ \lambda c.\psi_d\ c\ (\psi_u\ n\ m)\\ &= \quad \{\text{folding to the functional form}\}\\ &\quad\ f_{(\mathsf{True},\psi_u\ n\ m)}\end{aligned}$$

$$\psi'_u\ f_{(\mathsf{False},\_)}\ f_{(\mathsf{True},m)} = \lambda c.\psi_d\ c\ m = f_{(\mathsf{True},m)}$$

$$\psi'_u\ f_{(\mathsf{True},n)}\ f_{(\mathsf{False},\_)} = \lambda c.\psi_d\ c\ n = f_{(\mathsf{True},n)}$$

$$\psi'_u\ f_{(\mathsf{False},\_)}\ f_{(\mathsf{False},\_)} = \lambda c.c = f_{(\mathsf{False},\_)}$$

Based on the calculations so far, we can derive the auxiliary functions for the $\mathsf{dAcc_b}$ skeleton and obtain the following parallel implementation of the $\mathsf{dAcc_r}$ skeleton.

$$\begin{aligned}&\mathsf{dAcc_r}\ g\ c = \mathit{bt2rt} \circ \mathsf{dAcc_b}\ \langle \phi'_l, \phi'_r, \psi'_u, \psi'_d \rangle_{\mathrm{d}}\ c \circ \mathit{rt2bt}\\ &\quad\ \textbf{where}\ \phi'_l\ n = (\mathsf{True}, \phi\ n)\\ &\qquad\qquad \phi'_r\ n = (\mathsf{False}, \_)\\ &\qquad\qquad \psi'_d\ c\ (\mathsf{True}, n) = \psi_d\ c\ n\\ &\qquad\qquad \psi'_d\ c\ (\mathsf{False}, n) = c\\ &\qquad\qquad \psi'_u\ (\mathsf{True}, n)\ (\mathsf{True}, m) = (\mathsf{True}, \psi_u\ n\ m)\\ &\qquad\qquad \psi'_u\ (\mathsf{False}, \_)\ (\mathsf{True}, m) = (\mathsf{True}, m)\\ &\qquad\qquad \psi'_u\ (\mathsf{True}, n)\ (\mathsf{False}, \_) = (\mathsf{True}, n)\\ &\qquad\qquad \psi'_u\ (\mathsf{False}, \_)\ (\mathsf{False}, \_) = (\mathsf{False}, \_)\end{aligned}$$

4.3.5 Parallelizing Intra-Siblings Computations

The skeleton $\mathsf{rAcc_r}$ traverses each list of siblings from left to right on rose trees. This traversal corresponds to a top-down one on the binary-tree representation. We can implement the $\mathsf{rAcc_r}$ skeleton using the $\mathsf{dAcc_b}$ skeleton as follows. In the computation of the $\mathsf{dAcc_b}$ skeleton, we update the accumulative parameter using the operator passed to the $\mathsf{rAcc_r}$ skeleton for the right child, and reset the accumulative parameter to the unit for the left child.

$$\begin{aligned}\mathsf{rAcc_r}\ (\oplus) = {} & \mathit{bt2rt} \circ (\mathsf{dAcc_b}\ g_l\ g_r\ \iota_\oplus) \circ \mathit{rt2bt} \\ & \textbf{where}\ g_l\ c\ n = \iota_\oplus \\ & \qquad\ g_r\ c\ n = c \oplus n\end{aligned}$$

We derive the auxiliary functions for the parallel implementation of the $\mathsf{dAcc_b}$ skeleton based on the associativity of the operator $\oplus$. We consider the following form of functions defined with two parameters. The True case corresponds to the function g_l, and the False case corresponds to the function g_r.

$$\begin{aligned}f_{(\mathsf{True},n)} &= \lambda c.n \\ f_{(\mathsf{False},n)} &= \lambda c.c \oplus n\end{aligned}$$

We can confirm that the functional form is closed under function composition as the following calculations show. Function ψ_u is one of the auxiliary functions for the $\mathsf{dAcc_b}$ skeleton.

$$\begin{aligned}\psi_u\ f_{(\mathsf{True},n)}\ f_{(\mathsf{True},m)} &= \lambda c.m = f_{(\mathsf{True},m)} \\ \psi_u\ f_{(\mathsf{True},n)}\ f_{(\mathsf{False},m)} &= \lambda c.n \oplus m = f_{(\mathsf{True},n\oplus m)} \\ \psi_u\ f_{(\mathsf{False},n)}\ f_{(\mathsf{True},m)} &= \lambda c.m = f_{(\mathsf{True},m)} \\ \psi_u\ f_{(\mathsf{False},n)}\ f_{(\mathsf{False},m)} &= \lambda c.c \oplus n \oplus m = f_{(\mathsf{False},n\oplus m)}\end{aligned}$$

Based on this functional form, we obtain the following parallel implementation of the $\mathsf{rAcc_r}$ skeleton.

$$\begin{aligned}\mathsf{rAcc_r}\ (\oplus) = {} & \mathit{bt2rt} \circ \mathsf{dAcc_b}\ \langle \phi_l, \phi_r, \psi_u, \psi_d \rangle_\mathrm{d}\ \iota_\oplus \circ \mathit{rt2bt} \\ & \textbf{where}\ \phi_l\ n = (\mathsf{True}, n) \\ & \qquad\ \phi_r\ n = (\mathsf{False}, n) \\ & \qquad\ \psi_u\ (\mathsf{True}, n)\ (\mathsf{True}, m)\ = (\mathsf{True}, m) \\ & \qquad\ \psi_u\ (\mathsf{True}, n)\ (\mathsf{False}, m)\ = (\mathsf{True}, n \oplus m) \\ & \qquad\ \psi_u\ (\mathsf{False}, n)\ (\mathsf{True}, m)\ = (\mathsf{True}, m) \\ & \qquad\ \psi_u\ (\mathsf{False}, n)\ (\mathsf{False}, m) = (\mathsf{False}, n \oplus m) \\ & \qquad\ \psi_d\ c\ (\mathsf{True}, n)\ = n \\ & \qquad\ \psi_d\ c\ (\mathsf{False}, n) = c \oplus n\end{aligned}$$

The skeleton $\mathsf{lAcc_r}$ traverses each list of siblings from right to left. This traversal corresponds to a bottom-up one on the binary-tree representation. Therefore, an implementation of the $\mathsf{lAcc_r}$ skeleton should be given with the $\mathsf{uAcc_b}$ skeleton. We first consider the following definition with the $\mathsf{uAcc_b}$ and $\mathsf{map_b}$ skeletons.

$$\begin{aligned}& \mathit{bt2rt} \circ (\mathsf{uAcc_b}\ k) \circ (\mathsf{map_b}\ (\lambda x.\iota_\oplus)\ \mathit{id}) \circ \mathit{rt2bt} \\ & \textbf{where}\ k\ l\ b\ r = b \oplus r\end{aligned}$$

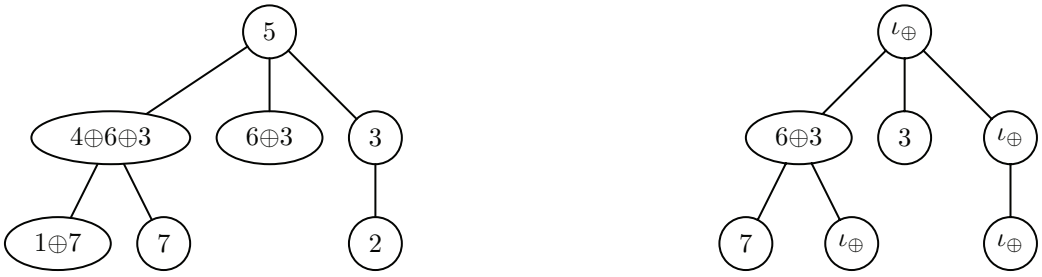


Figure 4.11. Left: the result of $bt2rt \circ (\mathsf{uAcc_b}\ (\lambda b\ l\ r.b \oplus r)) \circ (\mathsf{map_b}\ (\lambda x.\iota_\otimes)\ id) \circ rt2bt$. Right: the desired result for $\mathsf{lAcc_r}\ (\oplus)\ (\otimes)$.

The results of this computation are slightly different from what we want for the $\mathsf{lAcc_r}$ skeleton: the results should be shifted to the left by one on the rose tree as shown in Figure 4.11. We resolve this inconsistency by applying the $\mathsf{getchr_b}$ skeleton to the result before restoring the rose-tree structure, and we obtain an implementation of the $\mathsf{lAcc_r}$ skeleton as follows.

$$\begin{aligned}&\mathsf{lAcc_r}\ (\oplus) = bt2rt \circ (\mathsf{getchr_b}\ \_) \circ (\mathsf{uAcc_b}\ k) \circ (\mathsf{map_b}\ (\lambda x.\iota_\otimes)\ id) \circ rt2bt\\ &\qquad\textbf{where}\ k\ l\ b\ r = b \oplus r\end{aligned}$$

Next, we derive the auxiliary functions for the parallel implementation of the $\mathsf{uAcc_b}$ skeleton. We introduce the following form of functions defined with two parameters.

$$\begin{aligned}f_{(\mathsf{True},a)} &= \lambda l\ r.a \oplus r\\ f_{(\mathsf{False},a)} &= \lambda l\ r.a\end{aligned}$$

We can confirm that the functional form is closed through the computation of tree contraction algorithms as the following calculations show.

$$\begin{aligned}\psi_l\ f_{(\_,\_)}\ f_{(\mathsf{True},a_n)}\ r &= \lambda x\ y.a_n \oplus r = \lambda x\ y.f_{(\mathsf{False},a_n \oplus r)}\\ \psi_l\ f_{(\_,\_)}\ f_{(\mathsf{False},a_n)}\ r &= \lambda x\ y.a_n = \lambda x\ y.f_{(\mathsf{False},a_n)}\end{aligned}$$

$$\begin{aligned}\psi_r\ l\ f_{(\mathsf{True},a_n)}\ f_{(\mathsf{True},a_r)} &= \lambda x\ y.a_n \oplus a_r \oplus y = \lambda x\ y.f_{(\mathsf{True},a_n \oplus a_r)}\\ \psi_r\ l\ f_{(\mathsf{True},a_n)}\ f_{(\mathsf{False},a_r)} &= \lambda x\ y.a_n \oplus a_r = \lambda x\ y.f_{(\mathsf{False},a_n \oplus a_r)}\\ \psi_r\ l\ f_{(\mathsf{False},a_n)}\ f_{(\_,\_)} &= \lambda x\ y.a_n = \lambda x\ y.f_{(\mathsf{False},a_n)}\end{aligned}$$

Based on the functional form and the calculations above, we can derive the auxiliary functions for the $\mathsf{uAcc_b}$ skeleton and obtain the following parallel implementation of the $\mathsf{lAcc_r}$ skeleton.

$$\begin{aligned}&\mathsf{lAcc_r}\ (\oplus)\\ &\quad= bt2rt \circ (\mathsf{getchr_b}\ \_) \circ (\mathsf{uAcc_b}\ \langle \phi, \psi_n, \psi_l, \psi_r \rangle_\mathrm{u}) \circ (\mathsf{map_b}\ (\lambda x.\iota_\otimes)\ id) \circ rt2bt\\ &\qquad\textbf{where}\ \phi\ n = (\mathsf{True}, n)\\ &\qquad\qquad \psi_n\ l\ (\mathsf{True}, n)\ r = n \oplus r\\ &\qquad\qquad \psi_n\ l\ (\mathsf{False}, n)\ r = n\\ &\qquad\qquad \psi_l\ (\_,\_)\ (\mathsf{True}, a_n)\ r = (\mathsf{False}, a_n \oplus r)\\ &\qquad\qquad \psi_l\ (\_,\_)\ (\mathsf{False}, a_n)\ r = (\mathsf{False}, a_n)\\ &\qquad\qquad \psi_r\ l\ (\mathsf{True}, a_n)\ (\mathsf{True}, a_r) = (\mathsf{True}, a_n \oplus a_r)\\ &\qquad\qquad \psi_r\ l\ (\mathsf{True}, a_n)\ (\mathsf{False}, a_r) = (\mathsf{False}, a_n \oplus a_r)\\ &\qquad\qquad \psi_r\ l\ (\mathsf{False}, a_n)\ (\_,\_) = (\mathsf{False}, a_n)\end{aligned}$$

4.3.6 Parallel Cost of the Implementation

Now we briefly discuss the efficiency of our parallel implementation of the skeletons. We used two functions *rt2bt* and *bt2rt* for specifying the computation on the binary-tree representation, but we can remove these two functions away if two rose-tree skeletons are called successively. Thus we give the parallel cost of the rose-tree skeletons without these two functions.

Let N denote the number of nodes of the rose trees, and P the number of processors. Here we assume the EREW PRAMs for the parallel computing model, for which we have efficient implementation as discussed in Section 2.4. The parallel computation time of the map_r and $\mathsf{zipwith}_\mathrm{r}$ skeletons are $O(N/P)$, and the parallel costs of the other five skeletons are $O(N/P + \log P)$. Note that the implementation of the rose-tree skeletons is cost optimal, and it achieves linear speedups under the condition of $P \leq N/\log N$.

We may develop more involved implementations of the rose-tree skeletons by removing unnecessary intermediate structures and optimizing sequential parts. To develop such an implementation is our future work, and we believe the fusion transformations [51, 126] are useful for this optimization.

We summarize this section with the following theorem.

Theorem 4.1 *The seven parallel skeletons for rose trees defined in Figure 4.1 can be implemented in parallel based on the binary-tree representation with the parallel binary-tree skeletons.*

Proof. The correctness of the implementation of the rose-tree skeletons is almost self-evident from the derivation so far. □

4.4 Short Summary

In this chapter we have formalized a set of parallel skeletons for rose trees, and shown their parallel implementation. The rose-tree skeletons are defined based on the idea of constructive algorithmics. Five rose-tree skeletons are straightforward extensions of five binary-tree skeletons in Chapter 2, and two new rose-tree skeletons are introduced to represent rose-tree specific computational patterns. We have shown that the rose-tree skeletons can be implemented efficiently in parallel based on a binary-tree representation of rose trees.

Chapter 5

Theorems for Deriving Skeletal Parallel Programs

We have defined parallel tree skeletons for binary trees in Chapter 2, and for rose trees in Chapter 4. Each parallel skeleton provides a simple computational pattern that can be implemented efficiently in parallel, and we can build many parallel programs manipulating trees by composing tree skeletons with appropriate functions.

However, there is still a gap between sequential programs and skeletal parallel programs. Two problems are as follows.

- Difference of programming style: We need to develop parallel programs with parallel skeletons in a compositional style, whereas we develop sequential programs by recursive functions.
- Need of auxiliary functions: We require to derive auxiliary functions that are necessary to guarantee the parallel implementation of the skeletons.

These two problems let sequential programmers hesitate to develop parallel programs using tree skeletons.

In this chapter, we bridge the gap by proposing a systematic methodology of skeletal programming. Our methodology mainly consists of two sets of theorems: diffusion theorems and theorems on three algebraic properties. We first decompose a sequential program by diffusion theorems into composition of tree skeletons. We then apply theorems on algebraic properties to each call of a skeleton and derive auxiliary functions.

In Section 5.1, we develop diffusion theorems gradually on binary trees and rose trees. In Section 5.2, we show how we can derive auxiliary functions for binary skeletons by focusing on three algebraic properties of the operators. In Section 5.3, we illustrate our methodology of skeletal-program development by using the party planning problem in Introduction as the running example. We summarize this chapter in Section 5.4.

This chapter composes several results of our papers together. Section 5.1 is based on [88–90]. Section 5.2 is an extended version of [91].

5.1 Diffusion Theorems

We often develop programs manipulating trees as recursive functions. Since each parallel skeleton only provides a simple computational pattern, it is not easy to decide which parallel skeletons we should use to develop parallel programs from the recursively defined sequential algorithms. In this section, we develop diffusion theorems that decompose a complex recursive function into simpler ones that can be represented by parallel tree skeletons.

5.1.1 Diffusion Theorems for Binary Trees

We first develop diffusion theorems for binary trees. This section is an extended version of our previous paper [88].

We start by decomposing the tree homomorphism into a composition of binary-tree skeletons. It is worth noting that the following lemma is an extension of the first homomorphism theorem [15] proposed for lists.

Lemma 5.1 *Tree homomorphism* $h = (\![k_l, k_n]\!)_{\mathrm{b}}$ *can be implemented by the* $\mathsf{map}_{\mathrm{b}}$ *and* $\mathsf{reduce}_{\mathrm{b}}$ *skeletons.*

$$h = (\mathsf{reduce}_{\mathrm{b}}\ k_n) \circ (\mathsf{map}_{\mathrm{b}}\ k_l\ id)$$

Proof. The lemma can be easily proved by induction on the structure of binary trees. □

Many computations on trees are specified with an accumulative parameter. As an example of such recursive functions with an accumulative parameter, we first deal with functions in the following form.

$$\begin{aligned} h\ c\ (\mathsf{BLeaf}\ a) &= k_l\ (a, c) \\ h\ c\ (\mathsf{BNode}\ l\ b\ r) &= k_n\ (h\ (g_l\ c\ b)\ l)\ (b, c)\ (h\ (g_r\ c\ b)\ r) \end{aligned}$$

The main idea for decomposing a function with an accumulative parameter into skeletons is to compute in advance the accumulative parameter passed to each node using the $\mathsf{dAcc}_{\mathrm{b}}$ skeleton. After computing the accumulative parameter for each node, the function becomes a simple tree homomorphism.

$$\begin{aligned} h\ c\ t = \mathbf{let}\ & ct = \mathsf{zipwith}_{\mathrm{b}}\ (,)\ (,)\ t\ (\mathsf{dAcc}_{\mathrm{b}}\ (g_l, g_r)\ c\ t) \\ \mathbf{in}\ & (\![k_l, k_n]\!)_{\mathrm{b}}\ ct \end{aligned}$$

Now we apply Lemma 5.1 and obtain the following diffusion theorem for the decomposition of the recursive function defined with an accumulative parameter.

Theorem 5.1 (Binary-Tree Diffusion) *Let function* h *be defined as follows with an accumulative parameter.*

$$\begin{aligned} h\ c\ (\mathsf{BLeaf}\ a) &= k_l\ (a, c) \\ h\ c\ (\mathsf{BNode}\ l\ b\ r) &= k_n\ (h\ (g_l\ c\ b)\ l)\ (b, c)\ (h\ (g_r\ c\ b)\ r) \end{aligned}$$

This function h can be written with the binary-tree skeletons.

Proof. The decomposed skeletal program for the function h is given as follows.

$$\begin{aligned} h\ c\ t = \textbf{let}\ & ct = \mathsf{zipwith_b}\ (,)\ (,)\ t\ (\mathsf{dAcc_b}\ (g_l, g_r)\ c\ t) \\ \textbf{in}\ & \mathsf{reduce_b}\ k_n\ (\mathsf{map_b}\ k_l\ id\ ct) \end{aligned}$$

The correctness of this decomposed function can be proved by induction on the structure of binary trees. □

We can extend this diffusion theorem into paramorphic ones that has another recursive call of tree homomorphism. In the following we consider function h defined as follows with auxiliary tree homomorphisms $h' = (\![k'_l, k'_n]\!)_\mathrm{b}$ and $h'' = (\![k''_l, k''_n]\!)_\mathrm{b}$, where tree homomorphism h' is used to compute sub-results and tree homomorphism h'' is used to update the accumulative parameter.

$$\begin{aligned} h\ c\ (\mathsf{BLeaf}\ a) &= k_l\ (a, c, h'\ (\mathsf{BLeaf}\ a)) \\ h\ c\ (\mathsf{BNode}\ l\ b\ r) &= \textbf{let}\ t = \mathsf{BNode}\ l\ b\ r \\ &\quad\ \textbf{in}\ k_n\ (h\ (g_l\ c\ (b, h''\ t))\ l)\ (b, c, h'\ t)\ (h\ (g_r\ c\ (b, h''\ t)\ r)) \end{aligned}$$

A naive implementation of the function above is inefficient since it may have $\mathrm{O}(N^2)$ calls of functions h' and h'' where N denotes the number of nodes in a tree. To obtain an efficient implementation we first apply the auxiliary functions h' and h'' to each node by a bottom-up computation of the $\mathsf{uAcc_b}$ skeleton. We then apply Theorem 5.1 after this preprocessing. The following theorem summarize the discussion so far.

Theorem 5.2 *Let function h be defined as follows with auxiliary tree homomorphisms $h' = (\![k'_l, k'_n]\!)_\mathrm{b}$ and $h'' = (\![k''_l, k''_n]\!)_\mathrm{b}$.*

$$\begin{aligned} h\ c\ (\mathsf{BLeaf}\ a) &= k_l\ (a, c, h'\ (\mathsf{BLeaf}\ a)) \\ h\ c\ (\mathsf{BNode}\ l\ b\ r) &= \textbf{let}\ t = \mathsf{BNode}\ l\ b\ r \\ &\quad\ \textbf{in}\ k_n\ (h\ (g_l\ c\ (b, h''\ t))\ l)\ (b, c, h'\ t)\ (h\ (g_r\ c\ (b, h''\ t)\ r)) \end{aligned}$$

The function h can be implemented with the binary-tree skeletons.

Proof. The skeletal implementation of the function h is given as follows.

$$\begin{aligned} h\ c\ t = \textbf{let}\ & t' = \mathsf{uAcc_b}\ k'_n\ (\mathsf{map_b}\ k'_l\ id\ t) \\ & t'' = \mathsf{uAcc_b}\ k''_n\ (\mathsf{map_b}\ k''_l\ id\ t) \\ & ct = \mathsf{dAcc_b}\ (g_l, g_r)\ c\ (\mathsf{zipwith_b}\ \_\ (,)\ t\ t'') \\ & zt = \mathsf{zipwith_b}\ (\lambda a\ (b, c).(a, b, c))\ (\lambda a\ (b, c).(a, b, c)) \\ & \qquad\qquad\qquad\qquad\qquad\qquad t\ (\mathsf{zipwith_b}\ (,)\ (,)\ ct\ t') \\ \textbf{in}\ & \mathsf{reduce_b}\ k_n\ (\mathsf{map_b}\ k_l\ id\ zt) \end{aligned}$$

We can prove the correctness by a simple induction on the structure of binary trees. □

We can develop several variations of this theorem. Firstly, we can consider a function in which the last tree homomorphism returns a tree rather than a simple value. In this case, we simply use the $\mathsf{map_b}$ skeleton instead of the last $\mathsf{reduce_b}$ skeleton. Secondly, functions sometimes require the partial results of left subtree or right subtree rather than the whole

subtree. In this case, we add a call of $\mathsf{getchl}_\mathrm{b}$ or $\mathsf{getchr}_\mathrm{b}$ after the computation of the corresponding uAcc_b skeleton. Considering a variation where the last tree homomorphism returns a tree instead of a value and the accumulative parameter is updated with the left subtree's value, we have the following corollary of Theorem 5.2.

Corollary 5.1 *Let k_l, k_n, g_l and g_r be given functions, h' be a tree homomorphism $h' = (\![k'_l, k'_n]\!)_\mathrm{b}$. The function h defined as:*

$$
\begin{aligned}
h\ c\ (\mathsf{BLeaf}\ a) &= \mathsf{BLeaf}\ (k_l\ (a, c)) \\
h\ c\ (\mathsf{BNode}\ l\ b\ r) &= \mathsf{BNode}\ (h\ (g_l\ c\ (b, h'\ l)\ l))\ (k_n\ (b, c))\ (h\ (g_r\ c\ (b, h'\ l)\ l))
\end{aligned}
$$

can be decomposed into the binary-tree skeletons.

Proof. The skeletal implementation of the function h is given as follows.

$$
\begin{aligned}
h\ c\ t = \mathbf{let}\ t' &= \mathsf{getchl}_\mathrm{b}\ \_\ (\mathsf{uAcc}_\mathrm{b}\ k'_n\ (\mathsf{map}_\mathrm{b}\ k'_l\ id\ t)) \\
ct &= \mathsf{dAcc}_\mathrm{b}\ (g_l, g_r)\ c\ (\mathsf{zipwith}_\mathrm{b}\ \_\ (,)\ t\ t') \\
\mathbf{in}\ &\mathsf{zipwith}_\mathrm{b}\ (\lambda a\ c.k_l\ (a, c))\ (\lambda b\ c.k_n\ (b, c))\ t\ ct
\end{aligned}
$$

Note that the last $\mathsf{zipwith}_\mathrm{b}$ skeleton is fused from the $\mathsf{zipwith}_\mathrm{b}$ skeleton followed by the map_b*skeleton*. □

To illustrate how the diffusion theorems work, we develop a skeletal program for the prefix numbering on binary trees. The definition of recursive function *pre* that computes the prefix numbering is given as follows.

$$
\begin{aligned}
&pre_\mathrm{b}\ t = pre'_\mathrm{b}\ 0\ t \\
&pre'_\mathrm{b}\ c\ (\mathsf{BLeaf}\ a) = \mathsf{BLeaf}\ a \\
&pre'_\mathrm{b}\ c\ (\mathsf{BNode}\ l\ b\ r) = \mathsf{BNode}\ (pre'_\mathrm{b}\ (c+1)\ l)\ c\ (pre'_\mathrm{b}\ (c+1+size_\mathrm{b}\ l)\ r) \\
&size_\mathrm{b}\ (\mathsf{BLeaf}\ a) = 1 \\
&size_\mathrm{b}\ (\mathsf{BNode}\ l\ b\ r) = 1 + size_\mathrm{b}\ l + size_\mathrm{b}\ r
\end{aligned}
$$

The function $size_\mathrm{b}$ is by definition a homomorphism $size_\mathrm{b} = (\![size_l, size_n]\!)_\mathrm{b}$ where two functions are defined as $size_l\ a = 1$ and $size_n\ l\ b\ r = 1 + l + r$.

The definition of pre'_b for the BNode case can be transformed into the form of Corollary 5.1 by the introduction of functions k_n, g_l and g_r.

$$
\begin{aligned}
pre'_\mathrm{b}\ c\ (\mathsf{BNode}\ b\ l\ r) = \mathsf{BNode}\ &(k_n\ (b, c))\ (pre'_\mathrm{b}\ (g_l\ c\ (b, size_\mathrm{b}\ l))\ l) \\
&(pre'_\mathrm{b}\ (g_r\ c\ (b, size_\mathrm{b}\ l))) \\
\mathbf{where}\ &k_n\ (b, c) = c \\
&g_l\ c\ (b, s) = c + 1 \\
&g_r\ c\ (b, s) = c + 1 + s
\end{aligned}
$$

Therefore, we can apply Corollary 5.1 to obtain the following skeletal program, where we derive the skeletal program just by substituting the parameter functions k_n, g_l, g_r, $size_l$, and $size_n$ for those in the corollary.

$$
\begin{aligned}
pre'_\mathrm{b}\ c\ t = \mathbf{let}\ t' &= \mathsf{getchl}_\mathrm{b}\ \_\ (\mathsf{uAcc}_\mathrm{b}\ size_n\ (\mathsf{map}_\mathrm{b}\ size_l\ id\ t)) \\
ct &= \mathsf{dAcc}_\mathrm{b}\ (\lambda c\ (b, s).c + 1, \lambda c\ (b, s).c + 1 + s)\ c\ (\mathsf{zipwith}_\mathrm{b}\ \_\ (,)\ t\ t') \\
\mathbf{in}\ &\mathsf{zipwith}_\mathrm{b}\ (\lambda a\ c.c)\ (\lambda b\ c.c)\ t\ ct
\end{aligned}
$$

We can optimize the generated skeletal program by removing unnecessary computation. Since the functions g_l and g_r do not use the first value of the second argument, we can remove the first $\mathsf{zipwith}_\mathrm{b}$ skeleton with minor changes to the functions g_l and g_r. Since the two functions $(\lambda a\ c.c)$ and $(\lambda b\ c.c)$ used in the last $\mathsf{zipwith}_\mathrm{b}$ skeleton do not use the first argument either, the last $\mathsf{zipwith}_\mathrm{b}$ skeleton can be replaced with the map_b skeleton.

$$\begin{aligned} pre'_\mathrm{b}\ c\ t = \mathbf{let}\ & t' = \mathsf{getchl}_\mathrm{b}\ \_\ (\mathsf{uAcc}_\mathrm{b}\ size_n\ (\mathsf{map}_\mathrm{b}\ size_l\ id\ t)) \\ & ct = \mathsf{dAcc}_\mathrm{b}\ (\lambda c\ s.c+1, \lambda c\ s.c+1+s)\ c\ t' \\ \mathbf{in}\ & \mathsf{map}_\mathrm{b}\ (\lambda c.c)\ (\lambda c.c)\ ct \end{aligned}$$

The last map_b is in fact equal to the identity function, and therefore we can remove it.

$$\begin{aligned} pre'_\mathrm{b}\ c\ t = \mathbf{let}\ & t' = \mathsf{getchl}_\mathrm{b}\ \_\ (\mathsf{uAcc}_\mathrm{b}\ size_n\ (\mathsf{map}_\mathrm{b}\ size_l\ id\ t)) \\ \mathbf{in}\ & \mathsf{dAcc}_\mathrm{b}\ (\lambda c\ s.c+1, \lambda c\ s.c+1+s)\ c\ t' \end{aligned}$$

Finally, we substitute the initial value 0 of the accumulative parameter c, and succeed in deriving the following skeletal program.

$$\begin{aligned} pre_\mathrm{b}\ t = \mathbf{let}\ & t' = \mathsf{getchl}_\mathrm{b}\ \_\ (\mathsf{uAcc}_\mathrm{b}\ size_n\ (\mathsf{map}_\mathrm{b}\ size_l\ id\ t)) \\ \mathbf{in}\ & \mathsf{dAcc}_\mathrm{b}\ (\lambda c\ s.c+1, \lambda c\ s.c+1+s)\ 0\ t' \end{aligned}$$

5.1.2 Generalized Top-Down Computation on Rose Trees

Before developing diffusion theorems for rose trees, we generalize the dAcc_r skeleton for another top-down computational pattern.

The dAcc_r skeleton in Chapter 4 is a top-down computation on a rose tree where the accumulative parameter is updated only with the value of the node, and the accumulative parameters passed to the children are the same in the dAcc_r skeleton. In this section, we consider another top-down computation f modeled as follows, in which we update the accumulative parameter for each child with the child's value in addition to the node's value.

$$f\ c\ (\mathsf{RNode}\ a\ ts) = \mathsf{RNode}\ c\ [f\ (g\ c\ (a, root_\mathrm{r}\ t_i))\ t_i \mid i \in [1..\#ts]]$$

Let function g satisfy the condition of the dAcc_r skeleton, that is, the following equations hold for some auxiliary functions ϕ, ψ_u, and ψ_d.

$$\begin{aligned} & g\ c\ (a,b) = \psi_d\ c\ (\phi\ (a,b))\ \text{, and} \\ & \psi_d\ (\psi_d\ c\ ab)\ ab' = \psi_d\ c\ (\psi_u\ ab\ ab')\ . \end{aligned}$$

What makes this general top-down computation complicated is the existence of the function $root_\mathrm{r}$ in the update of the accumulative parameter. We first remove the function $root_\mathrm{r}$ by shifting computation of the accumulative parameter to the child. We can transform the function f above into the following one, in which the value of a node is passed to the children as a part of the accumulative parameter.

$$\begin{aligned} f\ c\ (\mathsf{RNode}\ a\ ts) &= \mathsf{RNode}\ c\ [f'\ (c,a)\ t_i \mid i \in [1..\#ts]] \\ f'\ (c,p)\ (\mathsf{RNode}\ a\ ts) &= \mathsf{RNode}\ (g\ c\ (p,a))\ [f'\ (g\ c\ (p,a), a)\ t_i \mid i \in [1..\#ts]] \end{aligned}$$

In the newly defined computation, the function f is applied to the root node and the function f' is applied to the other nodes.

To simplify the program we introduce two values ι_c and ι_p such that $g\ \iota_c\ (\iota_p, \_) = c$. If there are no values ι_c and ι_p satisfying this equation, we need to extend the definition of function g. With these two values, the function f can be simplified into the following definition.

$$
\begin{aligned}
&f\ c\ t = f'\ (\iota_c, \iota_p)\ t \\
&f'\ (c,p)\ (\mathsf{RNode}\ a\ ts) = \mathsf{RNode}\ (g\ c\ (p,a))\ [f'\ (g\ c\ (p,a), a)\ t_i \mid i \in [1..\#ts]]
\end{aligned}
$$

Now we implement the function f' with the rose-tree skeletons $\mathsf{dAcc_r}$ and $\mathsf{zipwith_r}$. First we compute the accumulative parameter passed to each node by the $\mathsf{dAcc_r}$ skeleton, and then compute the result value using the original value and the computed accumulative value by the $\mathsf{zipwith_r}$ skeleton.

$$
\begin{aligned}
f'\ (c,p)\ t = \mathsf{zipwith_r}\ k\ t\ (\mathsf{dAcc_r}\ g'\ (c,p)\ t) \\
\quad \textbf{where}\ k\ a\ (c,p) = g\ c\ (p,a) \\
\qquad\qquad g'\ (c,p)\ a = (g\ c\ (p,a), a)
\end{aligned}
$$

To prove that the implementation can be executed in parallel, we confirm that the function g' satisfies the condition of the $\mathsf{dAcc_r}$ skeleton. We consider the following form of functions defined with four parameters where the True case corresponds to the definition of g'.

$$
\begin{aligned}
f_{(\mathsf{True},a,\_,\_)} &= \lambda(c,p).(\psi_d\ c\ (\phi\ (p,a)), a) \\
f_{(\mathsf{False},a,b,d)} &= \lambda(c,p).(\psi_d\ c\ (\psi_u\ (\phi\ (p,a))\ b), d)
\end{aligned}
$$

The functional form is closed under function composition as the following calculations shows.

$$
\begin{aligned}
&f_{(\mathsf{True},a',\_,\_)} \circ f_{(\mathsf{True},a,\_,\_)} \\
&\quad = \lambda(c,p).(\psi_d\ (\psi_d\ c\ (\phi\ (p,a)))\ (\phi\ (a,a')), a') \\
&\quad = \lambda(c,p).(\psi_d\ c\ (\psi_u\ (\phi\ (p,a))\ (\phi\ (a,a'))), a') \\
&\quad = f_{(\mathsf{False},a,\phi\ (a,a'),a')}
\end{aligned}
$$

$$
\begin{aligned}
&f_{(\mathsf{True},a',\_,\_)} \circ f_{(\mathsf{False},a,b,d)} \\
&\quad = \lambda(c,p).(\psi_d\ (\psi_d\ c\ (\psi_u\ (\phi\ (p,a))\ b))\ (\phi\ (d,a')), a') \\
&\quad = \lambda(c,p).(\psi_d\ (\psi_d\ (\psi_d\ c\ (\phi\ (p,a)))\ b)\ (\phi\ (d,a')), a') \\
&\quad = \lambda(c,p).(\psi_d\ (\psi_d\ c\ (\phi\ (p,a)))\ (\psi_u\ b\ (\phi\ (d,a'))), a') \\
&\quad = \lambda(c,p).(\psi_d\ c\ (\psi_u\ (\phi\ (p,a))\ (\psi_u\ b\ (\phi\ (d,a')))), a') \\
&\quad = f_{(\mathsf{False},a,\psi_u\ b\ (\phi\ (d,a')),a')}
\end{aligned}
$$

$$
\begin{aligned}
&f_{(\mathsf{False},a',b',d')} \circ f_{(\mathsf{True},a,\_,\_)} \\
&\quad = \lambda(c,p).(\psi_d\ (\psi_d\ c\ (\phi\ (p,a)))\ (\psi_u\ (\phi\ (a,a'))\ b'), d') \\
&\quad = \lambda(c,p).(\psi_d\ c\ (\psi_u\ (\phi\ (p,a))\ (\psi_u\ (\phi\ (a,a'))\ b')), d') \\
&\quad = f_{(\mathsf{False},a,\psi_u\ (\phi\ (a,a'))\ b',d')}
\end{aligned}
$$

$$
\begin{aligned}
&f_{(\mathsf{False},a',b',d')} \circ f_{(\mathsf{False},a,b,d)} \\
&\quad = \lambda(c,p).(\psi_d\ (\psi_d\ c\ (\psi_u\ (\phi\ (p,a))\ b))\ (\psi_u\ (\phi\ (d,a'))\ b'),d') \\
&\quad = \lambda(c,p).(\psi_d\ (\psi_d\ (\psi_d\ c\ (\phi\ (p,a)))\ b)\ (\psi_u\ (\phi\ (d,a'))\ b'),d') \\
&\quad = \lambda(c,p).(\psi_d\ (\psi_d\ c\ (\phi\ (p,a)))\ (\psi_u\ b\ (\psi_u\ (\phi\ (d,a'))\ b')),d') \\
&\quad = \lambda(c,p).(\psi_d\ c\ (\psi_u\ (\phi\ (p,a))\ (\psi_u\ b\ (\psi_u\ (\phi\ (d,a'))\ b'))),d') \\
&\quad = f_{(\mathsf{False},a,(\psi_u\ b\ (\psi_u\ (\phi\ (d,a'))\ b')),d')}
\end{aligned}
$$

Based on the calculations for function composition, we can derive the auxiliary functions for the parallel implementation of the $\mathsf{dAcc_r}$ skeleton. Therefore, the skeletal program for the general downwards accumulation can be parallelized as stated by the following lemma.

Lemma 5.2 *Let g be a given function satisfying the following equations with some auxiliary functions ϕ, ψ_u, and ψ_d.*

$$
\begin{aligned}
&g\ c\ (a,b) = \psi_d\ c\ (\phi\ (a,b)) \\
&\psi_d\ (\psi_d\ c\ ab)\ ab' = \psi_d\ c\ (\psi_u\ ab\ ab')
\end{aligned}
$$

Under this condition, the function f defined as

$$
f\ c\ (\mathsf{RNode}\ a\ ts) = \mathsf{RNode}\ c\ [f\ (g\ c\ (a, root_{\mathrm{r}}\ t_i))\ t_i \mid i \in [1..\#ts]]
$$

can be decomposed into the rose-tree skeletons.

Proof. Let ι_c and ι_p be two values satisfying $g\ \iota_c\ (\iota_p, \_) = c$. Based on the discussion above, the skeletal program for the function f is given as follows.

$$
\begin{aligned}
&f\ c\ t = \mathsf{zipwith_r}\ k\ t\ (\mathsf{dAcc_r}\ g'\ (\iota_c,\iota_p)\ t) \\
&\qquad \textbf{where}\ k\ a\ (c,p) = g\ c\ (p,a) \\
&\qquad\qquad g'\ (c,p)\ a = (g\ c\ (p,a),a)
\end{aligned}
$$

The auxiliary functions for the parallel implementation of $\mathsf{dAcc_r}$ are given as follows.

$$
\begin{aligned}
&\phi'\ a = (\mathsf{True},a,\_,\_) \\
&\psi'_u\ (\mathsf{True},a,\_,\_)\ (\mathsf{True},a',\_,\_) = (\mathsf{False},a,\phi\ (a,a'),a') \\
&\psi'_u\ (\mathsf{False},a,b,d)\ (\mathsf{True},a',\_,\_) = (\mathsf{False},a,\psi_u\ b\ (\phi\ (d,a')),a') \\
&\psi'_u\ (\mathsf{True},a,\_,\_)\ (\mathsf{False},a',b',d') = (\mathsf{False},a,\psi_u\ (\phi\ (a,a'))\ b',d') \\
&\psi'_u\ (\mathsf{False},a,b,d)\ (\mathsf{False},a',b',d') = (\mathsf{False}_u\ (\phi\ (d,a'))\ b')),d') \\
&\psi'_d\ (c,p)\ (\mathsf{True},a,\_,\_) = (\psi_d\ c\ (\phi\ (p,a)),a) \\
&\psi'_d\ (c,p)\ (\mathsf{False},a,b,d) = (\psi_d\ c\ (\psi_u\ (\phi\ (p,a))\ b),d)
\end{aligned}
$$

□

5.1.3 Diffusion Theorems for Rose Trees

In this section we develop diffusion theorems for rose trees in a similar way as we have done for binary trees. In the definition of rose-tree skeletons, we have defined two reductions and two upwards accumulations. In the following discussion, we deal with computational patterns in which we apply the same computation both for internal nodes and for leaves.

First, we define the rose-tree homomorphism as a general bottom-up computation for rose trees. Since the data structure of rose trees is defined with lists, the homomorphism on rose trees is defined with the list homomorphism.

Definition 5.1 (List Homomorphism) Let k be a given function and $\oplus$ be an associative operator. Function h is a list homomorphism if it is defined as follows.

$$\begin{aligned} h\ [\,] &= \iota_\oplus \\ h\ [a] &= k\ a \\ h\ (x +\!\!+ y) &= h'\ x \oplus h'\ y \end{aligned}$$

□

By the list comprehension notation, we can denote the list homomorphism h above as $h\ as = \sum_\oplus [k\ a \mid i \in [1..\#as]]$.

Definition 5.2 (Rose-Tree Homomorphism) Let h' be a list homomorphism, and k be a function. The following function h defined with h' is called rose-tree homomorphism.

$$h\ (\mathsf{RNode}\ a\ ts) = k\ a\ (h'\ [h\ t_i \mid i \in [1..\#ts]])$$

□

The rose-tree homomorphism expresses a wide class of bottom-up computations, but in general it is hard or impossible to derive efficient parallel programs from any of them. Therefore, we define a subclass of rose-tree homomorphisms that can be implemented in parallel with the parallel skeletons.

Definition 5.3 (Parallelizable Homomorphism) Let $\oplus$ be an operator, $\otimes$ be an associative operator and be extended distributive over $\oplus$, and k be a function. A function h is said to be a parallelizable homomorphism if it is defined as follows

$$h\ (\mathsf{RNode}\ a\ ts) = k\ a \oplus \sum_{\otimes} [h\ t_i \mid i \in [1..\#ts]]\ .$$

We denote the parallelizable homomorphism h defined with function k and operators $\oplus$ and $\otimes$, as $h = (\!|k, \oplus, \otimes|\!)_{\mathrm{r}}$. □

This parallelizable homomorphism is a computational pattern generalized from the $\mathsf{reduce}_{\mathrm{r}}$ skeleton at the point that the function k is applied to all the nodes. Therefore, we can compute the parallelizable homomorphism by the $\mathsf{map}_{\mathrm{r}}$ skeleton followed by the $\mathsf{reduce}_{\mathrm{r}}$ skeleton.

Lemma 5.3 *A parallelizable homomorphism* $(\!|k, \oplus, \otimes|\!)_{\mathrm{r}}$ *can be implemented in parallel with the rose-tree skeletons.*

$$(\!|k, \oplus, \otimes|\!)_{\mathrm{r}} = (\mathsf{reduce}_{\mathrm{r}}\ (\oplus)\ (\otimes)) \circ (\mathsf{map}_{\mathrm{r}}\ k)$$

Proof. We can prove this theorem by induction on the structure of rose trees. □

It is worth noting that the theorem is an extension of the first homomorphism theorem for list [15] and Lemma 5.1 for binary trees.

To study the expressiveness of the rose-tree skeletons further, we next consider recursive computational patterns where the value of a node depends on not only the values of

the descendants but also those of the ancestors. We formalize such a computational pattern by describing the top-down dependency with an accumulative parameter c updated by a function g, and the bottom-up computation with a parallelizable homomorphism $(\!|k, \oplus, \otimes|\!)_{\mathrm{r}}$.

$$f\ c\ (\mathsf{RNode}\ a\ ts) = k\ (a, c) \oplus \textstyle\sum_{\otimes}[f\ t_i\ (g\ c\ a) \mid i \in [1..\#ts]]$$

We compute this function using rose-tree skeletons. We first generate a pair of the original value and the value of the accumulative parameter for each node using the $\mathsf{dAcc}_{\mathrm{r}}$ skeleton followed by the $\mathsf{zipwith}_{\mathrm{r}}$ skeleton. We then perform the over-all bottom-up computation which is exactly a parallelizable homomorphism. According to Lemma 5.3, we obtain the following skeletal program defined with the rose-tree skeletons.

$$\begin{aligned} f\ c\ t = {} & \mathbf{let}\ t' = \mathsf{zipwith}_{\mathrm{r}}\ (,)\ t\ (\mathsf{dAcc}_{\mathrm{r}}\ g\ c\ t) \\ & \mathbf{in}\ \mathsf{reduce}_{\mathrm{r}}\ (\oplus)\ (\otimes)\ (\mathsf{map}_{\mathrm{r}}\ k\ t') \end{aligned}$$

We may simplify the program above by fusing the $\mathsf{map}_{\mathrm{r}}$ and $\mathsf{zipwith}_{\mathrm{r}}$ skeletons,

$$\mathsf{map}_{\mathrm{r}}\ k\ (\mathsf{zipwith}_{\mathrm{r}}\ (,)\ t\ t') = \mathsf{zipwith}_{\mathrm{r}}\ (\lambda a\ b.k\ (a, b))\ t\ t'$$

and in summary we obtain the following theorem.

Theorem 5.3 (Rose-Tree Diffusion) *Let $(\!|k, \oplus, \otimes|\!)_{\mathrm{r}}$ be a parallelizable homomorphism, and g be a function. Function f defined as*

$$f\ c\ (\mathsf{RNode}\ a\ ts) = k\ a\ c \oplus \textstyle\sum_{\otimes}[f\ (g\ c\ a)\ t_i \mid i \in [1..\#ts]]$$

can be implemented by rose-tree skeletons.

Proof. The skeletal program for the function f is given as follows.

$$\begin{aligned} f\ c\ t = {} & \mathbf{let}\ ct = \mathsf{dAcc}_{\mathrm{r}}\ g\ t \\ & \mathbf{in}\ \mathsf{reduce}_{\mathrm{r}}\ (\oplus)\ (\otimes)\ (\mathsf{zipwith}_{\mathrm{r}}\ (\lambda a\ b.k\ (a, b))\ t\ ct) \end{aligned}$$

The correctness can be proved by induction on the structure of rose trees. □

We then extend Theorem 5.3 into a paramorphic one. Consider another computational pattern in which the accumulative parameter is updated not only the value of the node but also the reduced value of the subtree. Let the reduction be computed by a parallelizable homomorphism $h' = (\!|k', \oplus', \otimes'|\!)_{\mathrm{r}}$. We formalize such a computational pattern as the following recursive function.

$$\begin{aligned} f\ c\ (\mathsf{RNode}\ a\ ts) = {} & k\ (a, c) \oplus \textstyle\sum_{\otimes}[f\ c'\ t_i \mid i \in [1..\#ts]] \\ & \mathbf{where}\ c' = g\ c\ (a, h'\ (\mathsf{RNode}\ a\ ts)) \end{aligned}$$

This function is an instance of paramorphism [97] defined on rose trees.

As we decomposed the paramorphism on binary trees, we decompose this function into a procedure with three steps. In the first step, we preprocess the parallelizable

homomorphism h' with the map_r and uAcc_r skeletons for the whole rose tree. In the following, we denote the input tree as t.

$$t' = \mathsf{uAcc}_\mathrm{r}\ (\oplus')\ (\otimes')\ (\mathsf{map}_\mathrm{r}\ k'\ t)$$

In the second step, we compute the accumulative parameter for each node in a top-down manner. By Lemma 5.2, we can implement this computation using the $\mathsf{zipwith}_\mathrm{r}$ and dAcc_r skeletons.

$$t'' = \mathsf{dAcc}_\mathrm{r}\ g\ (\mathsf{zipwith}_\mathrm{r}\ (,)\ t\ t')$$

In the last step, we compute globally the parallelizable homomorphism using the $\mathsf{zipwith}_\mathrm{r}$ and $\mathsf{reduce}_\mathrm{r}$ skeleton.

$$\mathsf{reduce}_\mathrm{r}\ (\oplus)\ (\otimes)\ (\mathsf{zipwith}_\mathrm{r}\ (ab.k(a,b))\ t\ t'')$$

With these three steps, we can derive a skeletal parallel program for the computational pattern above.

Theorem 5.4 *Let h' be a parallelizable homomorphism $h' = (\![k', \oplus', \otimes']\!)_\mathrm{r}$, and g and k be functions. The function f defined as*

$$\begin{aligned} f\ c\ (\mathsf{RNode}\ a\ ts) = k\ (a,c) \oplus \textstyle\sum_{\otimes}[f\ c'\ t_i \mid i \in [1..\#ts]] \\ \textbf{where}\ c' = g\ c\ (a, h'\ (\mathsf{RNode}\ a\ ts)) \end{aligned}$$

can be decomposed into the composition of the rose-tree skeletons.

Proof. The skeletal program for the function f is given as follows.

$$\begin{aligned} f\ c\ t = \textbf{let}\ & t' = \mathsf{uAcc}_\mathrm{r}\ (\oplus')\ (\otimes')\ (\mathsf{map}_\mathrm{r}\ k'\ t) \\ & t'' = \mathsf{dAcc}_\mathrm{r}\ g\ (\mathsf{zipwith}_\mathrm{r}\ (,)\ t\ t') \\ \textbf{in}\ & \mathsf{reduce}_\mathrm{r}\ (\oplus)\ (\otimes)\ (\mathsf{zipwith}_\mathrm{r}\ (\lambda a\ b.k\ (a,b))) \end{aligned}$$

To prove the correctness of the skeletal program, we use the following equation. The equation states the relation between the reduction and the upwards accumulation.

$$\mathsf{reduce}_\mathrm{r}\ (\oplus)\ (\otimes) = \mathit{root}_\mathrm{r} \circ (\mathsf{uAcc}_\mathrm{r}\ (\oplus)\ (\otimes))$$

With this equation, we can prove the correctness by induction on the structure of rose trees. □

We can also develop variations of this theorem. The following corollary is for the functions that return a rose tree instead of a value.

Corollary 5.2 *Let h' be a parallelizable homomorphism $h' = (\![k', \oplus', \otimes']\!)_\mathrm{r}$, and g and k be functions. The function f defined as*

$$\begin{aligned} f\ c\ (\mathsf{RNode}\ a\ ts) = \mathsf{RNode}\ (k\ (a,c))\ [f\ c'\ t_i \mid i \in [1..\#ts]] \\ \textbf{where}\ c' = g\ c\ (a, h'\ (\mathsf{RNode}\ a\ ts)) \end{aligned}$$

can be decomposed into the composition of the rose-tree skeletons.

Proof. We can give the decomposed skeletal program as follows

$$\begin{aligned} f\ c\ t = \textbf{let}\ & t' = \mathsf{uAcc_r}\ (\oplus')\ (\otimes')\ (\mathsf{map_r}\ k'\ t) \\ & t'' = \mathsf{dAcc_r}\ g\ (\mathsf{zipwith_r}\ (,)\ t\ t') \\ \textbf{in}\ & \mathsf{zipwith_r}\ (\lambda a\ b.k\ (a,b))\ t\ t'' \end{aligned}$$

where the last $\mathsf{reduce_r}$ in Theorem 5.4 is omitted in the skeletal program above. □

Finally, we study a more general and complicated computational pattern defined as a top-down computation with dependencies not only on the values of subtrees but also among siblings. We formalize such a computational pattern as the following recursive function: the overall top-down computation is specified by function k and accumulative parameter c updated with function g; the bottom-up dependencies are specified with three parallelizable homomorphisms $f'_l = (\![k'_l, \oplus'_l, \otimes'_l]\!)_\mathrm{r}$, $f' = (\![k', \oplus', \otimes']\!)_\mathrm{r}$, and $f'_r = (\![k'_r, \oplus'_r, \otimes'_r]\!)_\mathrm{r}$; and the inter-sibling dependencies are specified as summations of the values of its left siblings or its right siblings with associative operators $\circledast_l$ and $\circledast_r$. The accumulative parameter is updated for the ith child with the value of the node (a), the i-th child's subtree (t'_i), summation of left siblings' values (l_i), and summation of right siblings' values (r_i). It is worth noting that the values of the accumulative parameter passed to children may differ in this specification.

$$\begin{aligned} f\ c\ (\mathsf{RNode}\ a\ ts) = \mathsf{RNode}\ (k\ a\ c)\ & [f\ c_i\ t_i \mid i \in [1..\#ts]] \\ \textbf{where}\ & c_i = g\ c\ (a, (l_i, t'_i, r_i)) \\ & l_i = \textstyle\sum_{\circledast_l} [f'_l\ t_j \mid j \in [1..i-1]] \\ & t'_i = f'\ t_i \\ & r_i = \textstyle\sum_{\circledast_r} [f'_r\ t_j \mid j \in [i+1..\#ts]] \end{aligned}$$

Since it is not efficient to compute the rose-tree homomorphisms for each node independently, we should compute them for all node at a time and put them together by the tupling technique [62]. We implement the bottom-up computations specified as parallelizable homomorphisms using the $\mathsf{uAcc_r}$ skeleton. For inter-sibling dependencies, we use the $\mathsf{rAcc_r}$ and $\mathsf{lAcc_r}$ skeletons. Therefore, we can compute the tuple (l_i, t'_i, r_i) for each node by the following four steps.

1. Compute l_i for all the nodes at once using the $\mathsf{rAcc_r}$ skeleton after the $\mathsf{map_r}$ and $\mathsf{uAcc_r}$ skeletons (line 1).

2. Compute t'_i for all the nodes at once using the $\mathsf{map_r}$, and $\mathsf{uAcc_r}$ skeletons (line 2).

3. Compute r_i for all the nodes at once using the $\mathsf{lAcc_r}$ skeleton after the $\mathsf{map_r}$ and $\mathsf{uAcc_r}$ skeletons (line 3).

4. Zip the results up by the $\mathsf{zipwith3_r}$ skeleton (line 4). The $\mathsf{zipwith3_r}$ skeleton is an extension of $\mathsf{zipwith_r}$ skeleton for zipping three rose trees of the same shape.

$$
\begin{aligned}
t'' = \textbf{let}\ & lt = \mathsf{rAcc_r}\ (\circledast_l)\ (\mathsf{uAcc_r}\ (\oplus'_l)\ (\otimes'_l)\ (\mathsf{map_r}\ k'_l\ t)) \\
& t' = \mathsf{uAcc_r}\ (\oplus')\ (\otimes')\ (\mathsf{map_r}\ k'\ t) \\
& rt = \mathsf{lAcc_r}\ (\circledast_r)\ (\mathsf{uAcc_r}\ (\oplus'_r)\ (\otimes'_r)\ (\mathsf{map_r}\ k'_r\ t)) \\
\textbf{in}\ & \mathsf{zipwith3_r}\ (\lambda l\ t'\ r.(l, t', r))\ lt\ t'\ rt
\end{aligned}
$$

After this preprocessing, we can compute the general computational pattern by the following function f' over the original rose tree and the rose tree of tuples generated by the preprocess. In the following program, $(\mathsf{RNode}\ a\ ts)$ denotes the original tree and $(\mathsf{RNode}\ a''\ ts'')$ denotes the generated tree.

$$
\begin{aligned}
f'\ c\ (\mathsf{RNode}\ a\ ts)\ (\mathsf{RNode}\ a''\ ts'') = \mathsf{RNode}\ (k\ a\ c)\ [f'\ c_i\ t_i\ t''_i \mid i \in [1..\#ts]] \\
\textbf{where}\ c_i = g\ c\ (a, (root\ t''_i))
\end{aligned}
$$

This function f' is a top-down computation where the accumulative parameter is updated for each child using the child's value in rose tree t''. We derive a skeletal program by applying the technique of Theorem 5.2. Let values ι_c and ι_p be values satisfying the following equation for the value a of the root and the initial accumulative parameter c.

$$g\ \iota_c\ (\iota_p, a) = c$$

Using these values, we can implement the function f' using rose-tree skeletons as follows.

$$
\begin{aligned}
f'\ c\ t\ t'' = \textbf{let}\ & dt = \mathsf{dAcc_r}\ g'\ (\iota_c, \iota_p)\ (\mathsf{zipwith_r}\ (,)\ t\ t'') \\
& ct = \mathsf{zipwith_r}\ (\lambda ltr\ (c, p).g\ c\ (p, ltr))\ t''\ dt \\
\textbf{in}\ & \mathsf{zipwith_r}\ (\lambda a\ c.k\ (a, c))\ t\ ct \\
\textbf{where}\ & g'\ (c, p)\ (a, ltr) = (g\ c\ (p, ltr), a)
\end{aligned}
$$

The auxiliary functions for the parallel implementation of $\mathsf{dAcc_r}$ can be derived from the following form of functions defined with the auxiliary functions ϕ, ψ_u, and ψ_d of g. We omit the derivation here since it is the same as the derivation in Section 5.1.2.

$$
\begin{aligned}
f_{(\mathsf{True}, ltr, \_, a)} &= \lambda(c, p).(\psi_d\ c\ (\phi\ (p, ltr)), a) \\
f_{(\mathsf{False}, ltr, b, a)} &= \lambda(c, p).(\psi_d\ c\ (\psi_u\ (\phi\ (p, ltr)\ b)), a)
\end{aligned}
$$

Theorem 5.5 *Let $\circledast_l$ and $\circledast_r$ be associative operators, f'_l, f', and f'_r be parallelizable homomorphisms defined as $f'_l = (\!|k'_l, \oplus'_l, \otimes'_l|\!)_{\mathrm{r}}$, $f' = (\!|k', \oplus', \otimes'|\!)_{\mathrm{r}}$, and $f'_r = (\!|k'_r, \oplus'_r, \otimes'_r|\!)_{\mathrm{r}}$, k be a function, and g be a function satisfying the following equations:*

$$
\begin{aligned}
& g\ c\ (a, ltr) = \psi_d\ c\ (\phi\ (a, ltr)) \\
& \psi_d\ (\psi_d\ c\ altr)\ altr' = \psi_d\ c\ (\psi_u\ altr\ altr')
\end{aligned}
$$

with auxiliary functions ϕ, ψ_u, and ψ_d.

The function f defined as follows can be decomposed into the rose-tree skeletons.

$$
\begin{aligned}
f\ c\ (\mathsf{RNode}\ a\ ts) = \mathsf{RNode}\ (k\ a\ c)\ [f\ c_i\ t_i \mid i \in [1..\#ts]] \\
\textbf{where}\ c_i = g\ c\ (a, (l_i, t'_i, r_i)) \\
l_i = \textstyle\sum_{\circledast_l} [f'_l\ t_j \mid j \in [1..i-1]] \\
t'_i = f'\ t_i \\
r_i = \textstyle\sum_{\circledast_r} [f'_r\ t_j \mid j \in [i+1..\#ts]]
\end{aligned}
$$

Proof. Let ι_c and ι_p be values satisfying equation $g\ \iota_c\ (\iota_p, (l, t', r)) = c$ for the initial value of associative operator c and the tupled value (l, t', r) of the root of zt.

The skeletal program for the function f is given as follows.

$$
\begin{aligned}
f\ c\ t = \textbf{let}\ & lt = \mathsf{rAcc_r}\ (\circledast_l)\ (\mathsf{uAcc_r}\ (\oplus'_l)\ (\otimes'_l)\ (\mathsf{map_r}\ k'_l\ t)) \\
& t' = \mathsf{uAcc_r}\ (\oplus')\ (\otimes')\ (\mathsf{map_r}\ k'\ t) \\
& rt = \mathsf{lAcc_r}\ (\circledast_r)\ (\mathsf{uAcc_r}\ (\oplus'_r)\ (\otimes'_r)\ (\mathsf{map_r}\ k'_r\ t)) \\
& zt = \mathsf{zipwith3_r}\ (\lambda l\ t\ r.(l, t, r))\ lt\ t'\ rt \\
& dt = \mathsf{dAcc_r}\ g'\ (\iota_c, \iota_p)\ (\mathsf{zipwith_r}\ (,)\ t\ t'') \\
\textbf{in}\ & \mathsf{zipwith3_r}\ (\lambda a\ ltr\ (c, p).k\ (a, g\ c\ (p, ltr)))\ t\ t''\ dt \\
\textbf{where}\ & g'\ (c, p)\ (a, ltr) = (g\ c\ (a, ltr), a)
\end{aligned}
$$

The auxiliary functions ϕ', ψ'_u and ψ'_d for the $\mathsf{dAcc_r}$ skeleton are given as follows using the auxiliary functions for g.

$$
\begin{aligned}
& \phi'\ (a, ltr) = (\mathsf{True}, ltr, \_, a) \\
& \psi'_u\ (\mathsf{True}, ltr, \_, a)\ (\mathsf{True}, ltr', \_, a') = (\mathsf{False}, ltr, \phi\ (a, ltr'), a') \\
& \psi'_u\ (\mathsf{False}, ltr, b, a)\ (\mathsf{True}, ltr', \_, a') = (\mathsf{False}, ltr, \psi_u\ b\ (\phi\ (a, ltr')), a') \\
& \psi'_u\ (\mathsf{True}, ltr, \_, a)\ (\mathsf{False}, ltr', b', a') = (\mathsf{False}, ltr, \psi_u\ (\phi\ (a, ltr'))\ b', a') \\
& \psi'_u\ (\mathsf{False}, ltr, b, a)\ (\mathsf{False}, ltr', b', a') = (\mathsf{False}, ltr, (\psi_u\ b\ (\psi_u\ (\phi\ (a, ltr'))\ b')), a') \\
& \psi'_d\ (c, p)\ (\mathsf{True}, ltr, \_, a) = (\psi_d\ c\ (\phi\ (p, ltr)), a) \\
& \psi'_d\ (c, p)\ (\mathsf{False}, ltr, b, a) = (\psi_d\ c\ (\psi_u\ (\phi\ (p, ltr))\ b), a)
\end{aligned}
$$

We can prove the correctness of the skeletal program above by induction on the structure of rose trees. □

Now we illustrate how these diffusion theorems work by the prefix numbering problem on rose trees. A sequential recursive program that solves the prefix numbering problem is given as follows.

$$
\begin{aligned}
& pre_{\mathrm{r}}\ t = pre'_{\mathrm{r}}\ 0\ t \\
& pre'_{\mathrm{r}}\ c\ (\mathsf{RNode}\ a\ ts) = \mathsf{RNode}\ c\ [pre'_{\mathrm{r}}\ (c + 1 + l_i)\ t_i \mid i \in [1..\#ts]] \\
& \qquad\qquad \textbf{where}\ l_i = \textstyle\sum_+ [size_{\mathrm{r}}\ t_j \mid j \in [1..i-1]] \\
& size_{\mathrm{r}}\ (\mathsf{RNode}\ a\ ts) = 1 + \textstyle\sum_+ [size_{\mathrm{r}}\ t_i \mid i \in [1..\#ts]]
\end{aligned}
$$

By comparing the definition above with the recursive definition of f in Theorem 5.5, we notice the following facts.

1. The values t'_i and r_i in Theorem 5.5 are not used in the specification of pre'_{r}.
2. The value a is not used both in function k and g.
3. The parameter functions in Theorem 5.5 can be specified by comparing the definition of h in the theorem and the recursive function pre_{r}.

$$
\begin{aligned}
& k\ \_\ c = c \\
& g\ c\ (\_, (l_i, \_, \_)) = c + 1 + l_i \\
& \circledast_l = + \\
& f'_l = (\![\lambda x.1, +, +]\!)_{\mathrm{r}}
\end{aligned}
$$

Now we derive the skeletal program for the function pre'_{r} by substituting the parameter functions to those in the theorem. Here, from the first fact we remove the computation for the t'_i and r_i, remove the $\mathsf{zipwith3}_{\mathrm{r}}$ skeleton, and simplify the definition of g' in Theorem 5.5.

$$\begin{aligned} pre'_{\mathrm{r}}\ c\ t = \mathbf{let}\ & lt = \mathsf{rAcc}_{\mathrm{r}}\ (+)\ (\mathsf{uAcc}_{\mathrm{r}}\ (+)\ (+)\ (\mathsf{map}_{\mathrm{r}}\ (\lambda x.1)\ t)) \\ & dt = \mathsf{dAcc}_{\mathrm{r}}\ g'\ (\iota_c, \iota_p)\ (\mathsf{zipwith}_{\mathrm{r}}\ (,)\ t\ lt) \\ & ct = \mathsf{zipwith}_{\mathrm{r}}\ (\lambda l\ (c, \_).c + 1 + l)\ lt\ dt \\ \mathbf{in}\ & \mathsf{zipwith}_{\mathrm{r}}\ (\lambda \_\ c.c)\ t\ ct \\ \mathbf{where}\ & g'\ (c, p)\ (a, l) = (g\ c\ (p, a), l) \end{aligned}$$

From the second fact we can remove the second value of the accumulative parameter and the last $\mathsf{zipwith}_{\mathrm{r}}$ skeleton, and we obtain the following simplified skeletal program.

$$\begin{aligned} pre'_{\mathrm{r}}\ c\ t = \mathbf{let}\ & lt = \mathsf{rAcc}_{\mathrm{r}}\ (+)\ (\mathsf{uAcc}_{\mathrm{r}}\ (+)\ (+)\ (\mathsf{map}_{\mathrm{r}}\ (\lambda x.1)\ t)) \\ & dt = \mathsf{dAcc}_{\mathrm{r}}\ g''\ \iota_c\ lt \\ \mathbf{in}\ & \mathsf{zipwith}_{\mathrm{r}}\ (\lambda l\ c.c + 1 + l)\ lt\ dt \\ \mathbf{where}\ & g''\ c\ l = c + 1 + l \end{aligned}$$

Finally, we derive the suitable value ι_c for the initial value of the downwards accumulation. By definition, we can easily obtain the value.

$$\begin{aligned} g''\ \iota_c\ 0 &= 0 \\ \iota_c + 1 + 0 &= 0 \\ \iota_c &= -1 \end{aligned}$$

Therefore, we obtain the following skeletal program for the prefix numbering problem. The auxiliary functions can be easily derived for this case since the function g'' is defined with an associative operator $+$.

$$\begin{aligned} pre_{\mathrm{r}}\ t = \mathbf{let}\ & lt = \mathsf{rAcc}_{\mathrm{r}}\ (+)\ (\mathsf{uAcc}_{\mathrm{r}}\ (+)\ (+)\ (\mathsf{map}_{\mathrm{r}}\ (\lambda x.1)\ t)) \\ & dt = \mathsf{dAcc}_{\mathrm{r}}\ (\lambda c\ l.c + l + 1)\ -1\ lt \\ \mathbf{in}\ & \mathsf{zipwith}_{\mathrm{r}}\ (\lambda l\ c.c + 1 + l)\ lt\ dt \end{aligned}$$

5.2 Properties for Deriving Parallelism

In this section we introduce three algebraic properties for parallel computing on trees. These properties guarantee the existence of auxiliary functions for the parallel implementation of binary-tree skeleton skeletons.

5.2.1 Semi-Associativity for Parallel Implementation of Skeletons

We first show the central idea of deriving the auxiliary functions for the parallel implementation of tree skeletons. We define semi-associativity of operator as follows.

Definition 5.4 (Semi-Associativity) Let $\oplus$ be an associative operator. Operator $\ominus$ is called semi-associative if it satisfies the following equation.

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

Here, the associative operator $\oplus$ is called the *associative complement* of operator $\ominus$. □

An example of the semi-associative operator is mirrored $-$ operator defined as

$$a -' b = b - a.$$

This operator $-'$ is semi-associative with its associative complement being operator $+$.

$$a -' (b -' c) \quad = \quad (c - b) - a \quad = \quad c - (b + a) \quad = \quad (a + b) -' c$$

Semi-associativity is a generalization of associativity. An associative operator is also semi-associative with its associative complement being itself. The unit of the associative operator is also the unit of the semi-associative operator.

Lemma 5.4 *Let $\ominus$ be a semi-associative operator with its associative complement being an associative operator $\oplus$. The unit of the associative operator, $\iota_\oplus$, is the left unit of the operator $\ominus$.*

Proof. It follows from the following calculations for any a and b that the lemma holds.

$$\begin{aligned} \iota_\oplus \ominus (a \ominus b) &= (\iota_\oplus \oplus a) \ominus b \\ &= a \ominus b \end{aligned}$$

□

Firstly let us consider derivation of auxiliary functions for the $\mathsf{reduce_b}$ and $\mathsf{uAcc_b}$ skeletons. Let the parameter function k used for the skeletons satisfy the following equations with semi-associative operator $\ominus$ and some functions k_l and k_r.

$$\begin{aligned} k\ b\ l\ r &= k_l\ n\ r \ominus l \\ k\ b\ l\ r &= k_r\ n\ l \ominus r \end{aligned}$$

To derive auxiliary functions under this condition, we use the following form of functions defined with two parameters a and b.

$$f_{(a,b)} = \lambda x\ y.a \ominus k\ b\ x\ y$$

By definition, we can write the function k in the form. In the following, let operator $\oplus$ be the associative complement of the operator $\ominus$. By the following definition, the associative operator $\oplus$ should have its unit $\iota_\oplus$.

$$\begin{aligned} k\ l\ b\ r &= (\lambda x\ y.\iota_\oplus \ominus k\ x\ b\ y)\ l\ r \\ &= f_{(\iota_\oplus, b)}\ l\ r \end{aligned}$$

We can confirm that the form of functions is preserved through contraction as the following calculations show.

$$\begin{aligned} \psi_l\ f_{(a_l,b_l)}\ f_{(a_n,b_n)}\ r &= \lambda x\ y.a_n \ominus k\ b_n\ (a_l \ominus k\ b_l\ x\ y)\ r \\ &= \quad \{\text{assumption on function } k\} \\ &\quad \lambda x\ y.a_n \ominus (k_l\ b_n\ r \ominus (a_l \ominus k\ b_l\ x\ y)) \\ &= \quad \{\text{semi-associativity}\} \\ &\quad \lambda x\ y.(a_n \oplus k_l\ b_n\ r \oplus a_l) \ominus k\ b_l\ x\ y \\ &= \quad \{\text{folding to the functional form}\} \\ &\quad f_{(a_n \oplus k_l\ b_n\ r \oplus a_l,\ b_l)} \end{aligned}$$

$$
\begin{aligned}
\psi_r\ l\ f_{(a_n,b_n)}\ f_{(a_r,b_r)} &= \lambda x\ y.a_n \oplus k\ b_n\ l\ (a_r \oplus k\ b_r\ x\ y) \\
&= \quad \{\text{assumption on function } k\} \\
&\quad \lambda x\ y.a_n \ominus (k_r\ b_n\ l \ominus (a_r \ominus k\ b_r\ x\ y)) \\
&= \quad \{\text{semi-associativity}\} \\
&\quad \lambda x\ y.(a_n \oplus k_r\ b_n\ r \oplus a_r) \ominus k\ b_r\ x\ y \\
&= \quad \{\text{folding to the functional form}\} \\
&\quad f_{(a_n \oplus k_r\ b_n\ r \oplus a_r,\ b_r)}
\end{aligned}
$$

Based on these calculations, we can derive auxiliary functions ϕ, ψ_n, ψ_l, and ψ_r as follows. In the following definition, we simply denote functional arguments as pairs.

Theorem 5.6 *Let $\ominus$ be a semi-associative operator, $\oplus$ be a associative complement of $\ominus$ with its unit $\iota_\oplus$, k_l and k_r be some functions. If function k satisfy the following functions,*

$$
\begin{aligned}
&k\ l\ b\ r = k_l\ b\ r \ominus l\ , \text{ and} \\
&k\ l\ b\ r = k_r\ b\ l \ominus r\ ,
\end{aligned}
$$

then there exist auxiliary functions ϕ, ψ_n, ψ_l and ψ_r such that $k = \langle \phi, \psi_n, \psi_l, \psi_r \rangle_{\mathrm{u}}$ holds.

Proof. We can define the auxiliary functions as follows based on the calculations above.

$$
\begin{aligned}
&\phi\ b = (\iota_\oplus, b) \\
&\psi_n\ l\ (a_n, b_n)\ r = a_n \ominus k\ b_n\ l\ r \\
&\psi_l\ (a_l, b_l)\ (a_n, b_n)\ r = (a_n \oplus k_l\ b_n\ r \oplus a_l, b_l) \\
&\psi_r\ l\ (a_n, b_n)\ (a_r, b_r) = (a_n \oplus k_r\ b_n\ l \oplus a_r, b_r)
\end{aligned}
$$

It can be easily shown that the auxiliary functions satisfy the condition of the $\mathsf{reduce}_{\mathrm{b}}$ and $\mathsf{uAcc}_{\mathrm{b}}$ skeletons. □

Secondly, let us consider the $\mathsf{dAcc}_{\mathrm{b}}$ skeleton called with functions g_l and g_r, which can be written with an associative operator $\ominus$ and some functions g'_l and g'_r as follows.

$$
\begin{aligned}
&g_l\ c\ b = g'_l\ b \ominus c \\
&g_r\ c\ b = g'_r\ b \ominus c
\end{aligned}
$$

To derive auxiliary functions for the $\mathsf{dAcc}_{\mathrm{b}}$ skeleton, we use the following form of functions.

$$
f_b = \lambda c.b \ominus c
$$

By definition, we can embed functions g_l and g_r into the form of functions.

$$
\begin{aligned}
&g_l\ c\ b = (\lambda c.g'_l\ b \ominus c)\ c = f_{(g'_l\ b)}\ c \\
&g_r\ c\ b = (\lambda c.g'_r\ b \ominus c)\ c = f_{(g'_r\ b)}\ c
\end{aligned}
$$

The preservation of the form through the contractions can be easily confirmed by the following calculation.

$$
\begin{aligned}
\psi_u\ f_b\ f'_b &= \lambda c.b' \ominus (b \ominus c) \\
&= \quad \{\text{semi-associativity}\} \\
&\quad \lambda c.(b' \oplus b) \ominus c) \\
&= \quad \{\text{folding to the functional form}\} \\
&\quad f_{(b' \oplus b)}
\end{aligned}
$$

Therefore, we can derive auxiliary functions for the $\mathsf{dAcc}_{\mathrm{b}}$ skeleton as stated by the following theorem.

Theorem 5.7 *Let $\ominus$ be a semi-associative operator, $\oplus$ be the associative complement of $\ominus$ with its unit $\iota_{\oplus}$, g'_l and g'_r be some functions. For functions g_l and g_r that can be written as*

$$g_l\ c\ b = g'_l\ b \ominus c\ , \text{ and}$$
$$g_r\ c\ b = g'_r\ b \ominus c\ ,$$

there exist auxiliary functions ϕ_l, ϕ_r, ψ_u and ψ_d such that $(g_l, g_r) = \langle \phi_l, \phi_r, \psi_u, \psi_d \rangle_{\mathrm{d}}$.

Proof. We can define the auxiliary functions as follows based on the calculations above.

$$\phi_l\ b = g'_l\ b$$
$$\phi_r\ b = g'_r\ b$$
$$\psi_u\ b\ b' = b' \oplus b$$
$$\psi_d\ c\ b = b \ominus c$$

□

So far, we have discussed the derivation of auxiliary functions for the parallel implementation under the existence of a semi-associative operator. If we can find a semi-associative operator in the definition of the functions, then we can easily obtain the auxiliary functions with the two theorems.

Now the question is how to find the semi-associative operator in the definition of functions. We here adopt the *context preservation technique* [32] that derives an associative operator by finding a function form closed under function composition. Since function composition is associative, we map the closed functional form with identity function to an isomorphic algebra with an associative operator with its unit.

In the following three sections, we show three algebraic properties under which we can systematically derive semi-associative operators.

5.2.2 Finiteness Property

There have been several studies [35, 57, 68, 121] that derive parallel algorithms based on the finiteness of the domain and range of functions. The common idea in these derivations is to compute the results for all the values in the domain.

Let $\mathcal{C}$ be a finite set that consists of l values $\{c_1, c_2, \ldots, c_l\}$. We can derive a semi-associative operator after finding a set of functions that are closed under function composition. We first specify such a set of functions defined on the finite set $\mathcal{C}$.

Definition 5.5 (Tabled Function) Let $\mathcal{C}$ be a finite set $\mathcal{C} = \{c_1, c_2, \ldots, c_l\}$. A function $g :: \mathcal{C} \to \mathcal{C}$ is said to be a *tabled function* on domain $\mathcal{C}$, if it is defined in the following form

$$g = \lambda x.\ \mathbf{case}\ x\ \mathbf{of}\ c_1 \to c'_1; c_2 \to c'_2; \cdots; c_l \to c'_l$$

where $c'_1, c'_2, \ldots c'_l \in \mathcal{C}$. □

An example of the tabled functions is a function *not* that reverses the boolean input.

$$not :: \mathsf{Bool} \to \mathsf{Bool}$$
$$not = \lambda x.\mathbf{case}\ x\ \mathbf{of}\ \mathsf{True} \to \mathsf{False}; \mathsf{False} \to \mathsf{True}$$

The following two lemmas prove that the identity function can be defined in the form of tabled function (Lemma 5.5) and that the tabled functions are closed under function composition (Lemma 5.6).

Lemma 5.5 *Let $\mathcal{C}$ be a finite set $\mathcal{C} = \{c_1, c_2, \ldots, c_l\}$. A set of tabled functions on domain $\mathcal{C}$ includes the identity function.*

Proof. The identity function *id* is given in the form of the tabled function as follows.

$$id = \lambda x.\ \mathbf{case}\ x\ \mathbf{of}\ c_1 \to c_1; c_2 \to c_2; \cdots; c_l \to c_l$$

□

Lemma 5.6 *Let $\mathcal{C}$ be a finite set $\mathcal{C} = \{c_1, c_2, \ldots, c_l\}$. A set of tabled functions on domain $\mathcal{C}$ is closed under function composition.*

Proof. Let g_1 and g_2 be defined as:

$$g_1 = \lambda x.\ \mathbf{case}\ x\ \mathbf{of}\ c_1 \to c_1^1; c_2 \to c_2^1; \cdots; c_l \to c_l^1\ ,\ \text{and}$$
$$g_2 = \lambda x.\ \mathbf{case}\ x\ \mathbf{of}\ c_1 \to c_1^2; c_2 \to c_2^2; \cdots; c_l \to c_l^2\ ,$$

where $c_i^1, c_i^2 \in \mathcal{C}$ for any $i \in \{1, 2, \ldots, l\}$. Let g be the composed function, $g = g_2 \circ c_1$. By definition, we have

$$g\ c_i = g_2\ (g_1\ c_i) = g_2\ c_i^1$$

for any $i \in \{1, 2, \ldots, l\}$, and therefore we can define g as the following tabled function:

$$g = \lambda x.\ \mathbf{case}\ x\ \mathbf{of}\ c_1 \to g_2\ c_1^1; c_2 \to g_2\ c_2^1; \cdots; c_l \to g_2\ c_l^1\ ,$$

where every $g_2\ c_i^1$ can be partially evaluated and the result value is in $\mathcal{C}$. □

Based on these two lemmas, we define operators for parallelization. We map tabled function $g :: \mathcal{C} \to \mathcal{C}$ onto a tuple of l values. More concretely, tabled function g defined as

$$g = \lambda x.\ \mathbf{case}\ x\ \mathbf{of}\ c_1 \to c_1'; c_2 \to c_2'; \cdots; c_l \to c_l'$$

is mapped to a tuple of l elements $(c_1', c_2', \ldots, c_l')$. On this mapping, semi-associative operator $\ominus$, its associative complement $\oplus$, and the unit $\iota_\oplus$ are given as shown in Figure 5.1.

We then consider the derivation of auxiliary functions for binary-tree skeletons.

Let k be a parameter function for the $\mathsf{reduce_b}$ and $\mathsf{uAcc_b}$ skeletons, and consider the case when the range of the function k is finite. In this case, we can denote the type of function k as $k :: \mathcal{C} \to \alpha \to \mathcal{C} \to \mathcal{C}$, where α is a certain type.

By fixing the first and the second arguments of k and partially evaluating the function, we can obtain a tabled function as follows. In the following function, term $(k\ l\ b\ c_i)$ can be computed into a value in $\mathcal{C}$.

$$\lambda r.k\ l\ b\ r = \lambda r.\mathbf{case}\ r\ \mathbf{of}\ c_1 \to k\ l\ b\ c_1; c_2 \to k\ l\ b\ c_2; \cdots; c_l \to k\ l\ b\ c_l$$

$$(c'_1, c'_2, \ldots, c'_k) \ominus x = \textbf{case } x \textbf{ of } c_1 \to c'_1; c_2 \to c'_2; \cdots; c_l \to c'_l$$

$$(c^2_1, c^2_2, \ldots, c^2_l) \oplus (c^1_1, c^1_2, \ldots, c^1_l) = (c'_1, c'_2, \ldots, c'_l)$$
$$\textbf{where } c'_i = \textbf{case } c^1_i \textbf{ of } c_1 \to c^2_1; c_2 \to c^2_2; \cdots; c_l \to c^2_l$$

$$\iota_\oplus = (c_1, c_2, \ldots, c_l)$$

Figure 5.1. The definition of the semi-associative operator $\ominus$, its associative complement $\oplus$, and the unit $\iota_\oplus$ derived from the mapping of tabled functions on domain $\mathcal{C} = \{c_1, c_2, \ldots, c_l\}$.

Based on this tabled function and the operator $\ominus$ defined above, we can derive the following equation,

$$k\ l\ b\ r = (k\ l\ b\ c_1, k\ l\ b\ c_2, \ldots, k\ l\ b\ c_l) \ominus r\ ,$$

which is the second equation in the assumption of Theorem 5.6 with substitution of

$$k_r\ b\ l = (k\ l\ b\ c_1, k\ l\ b\ c_2, \ldots, k\ l\ b\ c_l)\ .$$

By fixing the second and the third arguments of k and partially evaluating the function, we can derive the first equation in the assumption of Theorem 5.6. Based on the discussion so far, we can derive auxiliary functions for the $\mathsf{reduce_b}$ and $\mathsf{uAcc_b}$ skeletons if the range of the parameter function is finite.

Theorem 5.8 (Finiteness Property for $\mathsf{reduce_b}$ and $\mathsf{uAcc_b}$) *Let $\mathcal{C} = \{c_1, c_2, \ldots, c_l\}$ be a finite set. If the range of function k is the set $\mathcal{C}$, that is, the type of the function k is $k :: \mathcal{C} \to \alpha \to \mathcal{C} \to \mathcal{C}$ with a certain type α, then there exist auxiliary functions for the parallel implementation of $\mathsf{reduce_b}\ k$ and $\mathsf{uAcc_b}\ k$.*

Proof. Based on the partial evaluation of k into the tabled functions and Theorem 5.6, we can derive the auxiliary functions as follows. The operators $\ominus$ and $\oplus$ and the unit $\iota_\oplus$ are those in Figure 5.1.

$$\begin{aligned}
&\phi\ b = ((c_1, c_2, \ldots, c_l), b)\\
&\psi_n\ l\ ((c_{n1}, c_{n2}, \ldots, c_{nl}), b_n)\ r = \textbf{case } k\ l\ b_n\ r \textbf{ of } c_1 \to c_{n1}; c_2 \to c_{n2}; \cdots; c_l \to c_{nl}\\
&\psi_l\ ((c_{l1}, c_{l2}, \ldots, c_{ll}), b_l)\ ((c_{n1}, c_{n2}, \ldots, c_{nl}), b_n)\ r = ((l_1, l_2, \ldots, l_l), b_l)\\
&\quad \textbf{where } l_i = \textbf{case } k\ c_{li}\ b_n\ r \textbf{ of } c_1 \to c_{n1}; c_2 \to c_{n2}; \cdots; c_l \to c_{nl}\\
&\psi_r\ l\ ((c_{n1}, c_{n2}, \ldots, c_{nl}), b_n)\ ((c_{r1}, c_{r2}, \ldots, c_{rl}), b_r) = ((r_1, r_2, \ldots, r_l), b_r)\\
&\quad \textbf{where } r_i = \textbf{case } k\ l\ b_n\ c_{ri} \textbf{ of } c_1 \to c_{n1}; c_2 \to c_{n2}; \cdots; c_l \to c_{nl}
\end{aligned}$$

□

If the range of the two parameter functions g_l and g_r for the $\mathsf{dAcc_b}$ skeleton is finite, we can derive auxiliary functions in a similar way.

Theorem 5.9 (Finiteness Property for $\mathsf{dAcc_b}$) *Let $\mathcal{C} = \{c_1, c_2, \ldots, c_l\}$ be a finite set. If the range of function g_l and g_r is the set $\mathcal{C}$, then there exist auxiliary functions for the parallel implementation of $\mathsf{dAcc_b}\ (g_l, g_r)$.*

Proof. We can rewrite the functions g_l and g_r as tabled functions as follows.

$$g_l\ c\ b = \textbf{case}\ c\ \textbf{of}\ c_1 \to g_l\ c_1\ b; c_2 \to g_l\ c_2\ b; \cdots; c_l \to g_l\ c_l\ b$$
$$g_r\ c\ b = \textbf{case}\ c\ \textbf{of}\ c_1 \to g_r\ c_1\ b; c_2 \to g_r\ c_2\ b; \cdots; c_l \to g_r\ c_l\ b$$

Based on these tabled functions, we obtain the following two equations using the semi-associative operator $\ominus$.

$$g_l\ c\ b = (g_l\ c_1\ b, g_l\ c_2\ b, \ldots g_l\ c_l\ b) \ominus c$$
$$g_r\ c\ b = (g_r\ c_1\ b, g_r\ c_2\ b, \ldots g_r\ c_l\ b) \ominus c$$

By applying Theorem 5.7 to the equations above, we can derive the auxiliary functions as follows. The operators $\ominus$ and $\oplus$ and the unit $\iota_\oplus$ are those in Figure 5.1.

$$\phi_l\ b = (g_l\ c_1\ b, g_l\ c_2\ b, \ldots, g_l\ c_l\ b)$$
$$\phi_r\ b = (g_r\ c_1\ b, g_r\ c_2\ b, \ldots, g_r\ c_l\ b)$$
$$\psi_u\ (b_1^1, b_2^1, \ldots, b_l^1)\ (b_1^2, b_2^2, \ldots, b_l^2)$$
$$\quad = (b_1', b_2', \ldots, b_l')\ \textbf{where}\ b_i' = \textbf{case}\ b_i^1\ \textbf{of}\ c_1 \to b_1^2; c_2 \to b_2^2; \cdots; c_l \to b_l^2$$
$$\psi_d\ c\ (b_1, b_2, \ldots, b_l) = \textbf{case}\ c\ \textbf{of}\ c_1 \to b_1; c_2 \to b_2; \cdots; c_l \to b_l$$ □

The finiteness property is simple and powerful in deriving parallel programs, and several development of parallel algorithms developed so far [57, 121] used this property. We use this property later in parallelizing the party planning problem in Section 5.3 and the maximum marking problem in Chapter 8.

5.2.3 Extended-Ring Property

Distributivity is one of important properties in deriving parallel programs. For example, linear recurrence equations of the form $x_{n+1} = a \times x_n + b$ can be computed efficiently in parallel based on the distributivity of $\times$ over $+$ in addition to the associativity of both operators. Xu et al. [129] formalized an interesting property named *extended ring*, and developed a parallelizing system for programs manipulating lists. We adopt their idea for parallelization of tree programs.

First, we introduce algebraic property on two or more operators by extending the algebraic ring.

Definition 5.6 (Extended Ring [129]) Let $\mathcal{D}$ is a set of elements. Algebra $\mathcal{A} = \{\mathcal{D}, \oplus_1, \cdots, \oplus_k\}$ is said to be an *extended ring*, if

- for each $i \in \{1, \ldots, k\}$, operator $\oplus_i$ is associative with the unit $\iota_{\oplus_i}$; and
- for any i and j such that $1 \leq i < j \leq k$, $\oplus_j$ distributes over $\oplus_i$. □

The definition of extended ring above is a bit different from that in [129]: each operator $\oplus_i$ should be associative but not semi-associative. We strengthen the condition on operators to allow the normal form below has some variables after the parameter x.

We then specify the form of functions that are closed under function composition.

Definition 5.7 (NF-function on Extended Ring) A unary function g is said to be a *normal-form function (NF-function) on an extended ring*, if it is defined in the following form.

$$g = \lambda x.\ a_1^l \oplus_1 (\ \cdots\ (a_k^l \oplus_k x \oplus_k a_k^r)\ \cdots\ ) \oplus_1 a_1^r$$

Here, $a_1^l, a_2^l, \ldots, a_k^l, a_k^r, \ldots, a_1^r$ are some values that do not include the parameter of function x. □

When an operator $\oplus_i$ is commutative, we can consider a function simplified by merging terms a_i^l and a_i^r as the normal-form function.

The identity function can be defined as an NF-function, and NF-functions are closed under function composition.

Lemma 5.7 *The identity function is an NF-function on the given extended ring.*

Proof. Let given extended ring be $\{\mathcal{D}, \oplus_1, \cdots, \oplus_k\}$, the identity function id can be defined as an NF-function as shown in the following.

$$id = \lambda x.\ \iota_{\oplus_1} \oplus_1 (\cdots (\iota_{\oplus_k} \oplus_k x \oplus_k \iota_{\oplus_k}) \cdots) \oplus_1 \iota_{\oplus_1}$$

□

Lemma 5.8 *A set of NF-functions on an extended ring is closed under function composition.*

Proof. Let $\{\mathcal{D}, \oplus_1, \cdots, \oplus_k\}$ be an extended ring, and g_1 and g_2 be NF-functions on the extended ring defined as follows.

$$g_1 = \lambda x.\ a_1^l \oplus_1 (\cdots (a_k^l \oplus_k x \oplus_k a_k^r) \cdots) \oplus_1 a_1^r$$
$$g_2 = \lambda x.\ b_1^l \oplus_1 (\cdots (b_k^l \oplus_k x \oplus_k b_k^r) \cdots) \oplus_1 b_1^r$$

The function composition of g_1 and g_2 yields the following new NF-function on the extended ring.

$$\begin{aligned} g_2 \circ g_1 = \lambda x.\ & c_1^l \oplus_1 (\cdots (c_k^l \oplus_k x \oplus_k c_k^r) \cdots) \oplus_1 c_1^r \\ & \textbf{where}\ c_1^l = b_1^l \oplus_1 (b_2^l \oplus_2 (\cdots (b_k^l \oplus_k a_1^l) \cdots)) \\ & \qquad c_2^l = b_2^l \oplus_2 (\cdots (b_k^l \oplus_k a_2^l) \cdots) \\ & \qquad\quad \vdots \\ & \qquad c_k^l = b_k^l \oplus_k a_k^l \\ & \qquad c_k^r = a_k^r \oplus_k b_k^r \\ & \qquad\quad \vdots \\ & \qquad c_k^1 = ((\cdots (a_1^r \oplus_k b_k^r) \cdots) \oplus_2 b_2^r) \oplus_1 b_1^r \end{aligned}$$

The function above is derived from the definition of the NF-function with the distributivity and associativity of operators. □

$$(a^l_1, a^l_2, \ldots a^l_k, a^r_k, \ldots, a^r_1) \ominus x = a^l_1 \oplus_1 (\cdots (a^l_k \oplus_k x \oplus_k a^r_k) \cdots) \oplus_1 a^r_1$$

$$(b^l_1, b^l_2, \ldots b^l_k, b^r_k, \ldots, b^r_1) \otimes (a^l_1, a^l_2, \ldots a^l_k, a^r_k, \ldots, a^r_1) = (c^l_1, c^l_2, \ldots c^l_k, c^r_k, \ldots, c^r_1)$$

$$\begin{aligned}
\textbf{where } c^l_1 &= b^l_1 \oplus_1 (b^l_2 \oplus_2 (\cdots (b^l_k \oplus_k a^l_1) \cdots)) \\
c^l_2 &= b^l_2 \oplus_2 (\cdots (b^l_k \oplus_k a^l_2) \cdots) \\
&\vdots \\
c^l_k &= b^l_k \oplus_k a^l_k \\
c^r_k &= a^r_k \oplus_k b^r_k \\
&\vdots \\
c^1_k &= ((\cdots (a^r_1 \oplus_k b^r_k) \cdots) \oplus_2 b^r_2) \oplus_1 b^r_1
\end{aligned}$$

$$\iota_\oplus = (\iota_{\oplus_1}, \iota_{\oplus_2}, \ldots, \iota_{\oplus_k}, \iota_{\oplus_k}, \ldots, \iota_{\oplus_1})$$

Figure 5.2. The definition of the semi-associative operator $\ominus$, its associative complement $\oplus$, and the unit $\iota_\oplus$ derived from the mapping of NF-functions on extended ring $\{\mathcal{D}, \oplus_1, \cdots, \oplus_k\}$.

Based on these two lemmas, we can derive an associative operator for parallelizing computation defined with operators on an extended ring. Let $\{\mathcal{D}, \oplus_1, \cdots, \oplus_k\}$ be a given extended ring, we map an NF-function defined as

$$g = \lambda x.\ a^l_1 \oplus_1 (\ \cdots\ (a^l_k \oplus_k x \oplus_k a^r_k)\ \cdots\ ) \oplus_1 a^r_1$$

to a tuple of $2 \times k$ elements

$$(a^l_1, \ldots, a^l_k, a^r_k, \ldots, a^r_1)\ .$$

On this mapping, we can define semi-associative operator $\ominus$, its associative complement $\oplus$, and the unit $\iota_\oplus$ as shown in Figure 5.2.

We then formalize classes of functions that can be systematically parallelized based on the NF-functions on an extended ring.

Definition 5.8 A function f is said to be *linear with respect to its argument* x, if its definition contains exactly one occurrence of x. □

For example, the following function k is linear with respect to the first argument l, but not with respect to the third argument r.

$$k\ l\ b\ r = (l + b + r) \uparrow (b + r)$$

We may transform the function definition so that it becomes linear. The example function above can be transformed into the following function that is linear with respect to both the first and the third argument.

$$k\ l\ b\ r = (l \uparrow 0) + b + r$$

We can utilize several techniques to make the given function linear. Three of which are tupling transformation [30, 58], conditional normalization [29], unfolding and folding transformation.

If a function defined only with the operators on an extend ring is linear with respect to its parameter x, then we can systematically transform the function into an NF-function. The transformation consists of unfolding of expressions by distributivity between two operators and folding of expressions by associativity. We insert the units if necessary. The following is an example of the transformation: the function is defined on extended-ring $\{\mathsf{Num}, +, \times\}$.

$$
\begin{aligned}
&\lambda x.a + (b \times (c + (d \times x))) \\
&= \lambda x.a + (b \times c) + (b \times d \times x) \\
&= \lambda x.(a + (b \times c)) + ((b \times d) \times x) \\
&= \lambda x.p + (q \times x) \\
&\quad \textbf{where } p = a + (b \times c) \\
&\qquad\qquad q = b \times d
\end{aligned}
$$

Note that in the example above two right parameters are omitted since the two operators are associative. If we do not omit them, the NF-function is defined as follows.

$$\lambda x.p + (q \times x \times 1) + 0$$

Now we show a sufficient condition for the parallelization of binary-tree skeletons if the parameter functions are defined on an extended ring.

Theorem 5.10 (Extended-Ring Property for $\mathsf{reduce_b}$ and $\mathsf{uAcc_b}$) *Let $\{\mathcal{D}, \oplus_1, \ldots, \oplus_k\}$ be an extended ring. If function g is defined only with the operators $\oplus_1, \ldots, \oplus_k$, and g is linear with respect to each of the first and the third arguments, then there exists efficient parallel implementation for $\mathsf{reduce_b}$ g and $\mathsf{uAcc_b}$ g.*

Proof. From the linearity, we can transform two functions $\lambda l.g\ l\ b\ r$ and $\lambda r.g\ l\ b\ r$ into NF-functions on the extended ring. By mapping the NF-functions to tuples and applying Theorem 5.6, we can derive the auxiliary functions for the $\mathsf{reduce_b}$ and $\mathsf{uAcc_b}$ skeletons. The semi-associative operator $\ominus$, its associative complement $\oplus$, and the unit $\iota_\oplus$ are given in Figure 5.2. □

Theorem 5.11 (Extended-Ring Property for $\mathsf{dAcc_b}$) *Let $\{\mathcal{D}, \oplus_1, \ldots, \oplus_k\}$ be an extended ring. If functions g_l and g_r are defined only with the operators $\oplus_1, \ldots, \oplus_k$, and they are linear with respect to the first argument, then there exists efficient parallel implementation for $\mathsf{dAcc_b}$ (g_l, g_r).*

Proof. From the linearity, we can transform two functions g_l and g_r into NF-functions on the extended ring. By mapping the NF-functions to tuples and applying Theorem 5.7, we can derive the auxiliary functions for the $\mathsf{dAcc_b}$ skeleton. The semi-associative operator $\ominus$, its associative complement $\oplus$, and the unit $\iota_\oplus$ are given in Figure 5.2. □

In the following, we illustrate how we can derive auxiliary functions by these theorems. We use, as our running example, evaluation of computational tree whose internal node has one of operators $\downarrow$, $\uparrow$, $+$, and $\times$. This problem was also dealt with by Miller and Teng [101].

If the values on leaves are non-negative then we can derive an efficient parallel implementation, since the algebra $\{\mathbb{R}^+ \cup \{0, +\infty\}, \downarrow, \uparrow, +, \times\}$ is an extended ring. We can implement the evaluation by the $\mathsf{reduce}_\mathrm{b}$ skeleton called with parameter function g defined as $g\ l\ (\odot)\ r = l \odot r$ where $\odot$ is an operator attached to internal nodes. The function g is obviously linear with respect to each of the first and third arguments. By inserting units, $\iota_\downarrow = +\infty$, $\iota_\uparrow = 0$, $\iota_+ = 0$, and $\iota_\times = 1$, we obtain the following NF-functions. Note that the we omit the right parameters because every operator is commutative.

$$\lambda l.k_n\ l\ (\downarrow)\ r = \lambda l.r \downarrow (0 \uparrow (0 + (1 \times l)))$$
$$\lambda l.k_n\ l\ (\uparrow)\ r = \lambda l.{+\infty} \downarrow (l \uparrow (0 + (1 \times l)))$$
$$\lambda l.k_n\ l\ (+)\ r = \lambda l.{+\infty} \downarrow (0 \uparrow (l + (1 \times l)))$$
$$\lambda l.k_n\ l\ (\times)\ r = \lambda l.{+\infty} \downarrow (0 \uparrow (0 + (l \times l)))$$

We can give the definition in the same way for the case that the value of r is missing.

The semi-associative operator $\ominus$, its associative complement $\oplus$, and the unit $\iota_\oplus$ are given as follows for the extended ring.

$$(a_1, a_2, a_3, a_4) \ominus x = a_1 \downarrow (a_2 \uparrow (a_3 + (a_4 \times x)))$$

$$\begin{aligned}(b_1, b_2, b_3, b_4) \oplus (a_1, a_2, a_3, a_4) = (c_1, c_2, c_3, c_4)& \\ \textbf{where}\ c_1 &= b_1 \downarrow (b_2 \uparrow (b_3 + (b_4 \times a_1))) \\ c_2 &= b_2 \uparrow (b_3 + (b_4 \times a_2)) \\ c_3 &= (b_3 + (b_4 \times a_3) \\ c_4 &= b_4 \times a_4\end{aligned}$$

$$\iota_\oplus = (+\infty, 0, 0, 1)$$

Now we can apply Theorem 5.10 and obtain the following auxiliary functions.

$$\phi\ (\odot) = ((+\infty, 0, 0, 1), (\odot))$$

$$\psi_n\ l\ ((a_{1n}, a_{2n}, a_{3n}, a_{4n}), \odot_n)\ r = a_{1n} \downarrow (a_{2n} \uparrow (a_{3n} + (a_{4n} \times (l \odot_n r))))$$

$$\begin{aligned}&\psi_l\ ((a_{1l}, a_{2l}, a_{3l}, a_{4l}), (\odot_l))\ ((a_{1n}, a_{2n}, a_{3n}, a_{4n}), (\odot_n))\ r \\ &\quad = ((a_{1n}, a_{2n}, a_{3n}, a_{4n}) \oplus (b_1, b_2, b_3, b_4) \oplus (a_{1l}, a_{2l}, a_{3l}, a_{4l}), \odot_l) \\ &\qquad \textbf{where}\ b_1 = \textbf{if}\ ((\odot_n) == (\downarrow))\ \textbf{then}\ r\ \textbf{else}\ +\infty \\ &\qquad\qquad b_2 = \textbf{if}\ ((\odot_n) == (\uparrow))\ \textbf{then}\ r\ \textbf{else}\ 0 \\ &\qquad\qquad b_3 = \textbf{if}\ ((\odot_n) == (+))\ \textbf{then}\ r\ \textbf{else}\ 0 \\ &\qquad\qquad b_4 = \textbf{if}\ ((\odot_n) == (\times))\ \textbf{then}\ r\ \textbf{else}\ 1\end{aligned}$$

$$\begin{aligned}&\psi_r\ l\ ((a_{1n}, a_{2n}, a_{3n}, a_{4n}), (\odot_n))\ ((a_{1r}, a_{2r}, a_{3r}, a_{4r}), (\odot_r)) \\ &\quad = ((a_{1n}, a_{2n}, a_{3n}, a_{4n}) \oplus (b_1, b_2, b_3, b_4) \oplus (a_{1r}, a_{2r}, a_{3r}, a_{4r}), \odot_r) \\ &\qquad \textbf{where}\ b_1 = \textbf{if}\ ((\odot_n) == (\downarrow))\ \textbf{then}\ r\ \textbf{else}\ +\infty \\ &\qquad\qquad b_2 = \textbf{if}\ ((\odot_n) == (\uparrow))\ \textbf{then}\ r\ \textbf{else}\ 0 \\ &\qquad\qquad b_3 = \textbf{if}\ ((\odot_n) == (+))\ \textbf{then}\ r\ \textbf{else}\ 0 \\ &\qquad\qquad b_4 = \textbf{if}\ ((\odot_n) == (\times))\ \textbf{then}\ r\ \textbf{else}\ 1\end{aligned}$$

With these auxiliary functions, we can implement the evaluation of the computational trees as $\mathsf{reduce}_{\mathrm{b}}\ \langle \phi, \psi_n, \psi_l, \psi_r \rangle_{\mathrm{u}}$.

5.2.4 Tupled-Ring Property

Many algorithms developed by the dynamic programming technique compute multiple values simultaneously along with data structures. There values often have complex dependency among themselves and therefore it is difficult to develop parallel programs for these algorithms. In this section, we give a condition for deriving auxiliary functions systematically from algorithms that compute multiple values simultaneously on trees. The idea is to utilize the associativity of matrix multiplication defined on a commutative semiring. The condition, tupled-ring property, is practical and captures a wide class of dynamic programming algorithms on trees.

First we define commutative semirings.

Definition 5.9 An algebra $\mathcal{A} = \{\mathcal{D}, \oplus, \otimes\}$ is said to be a commutative semiring, if the following three hold.

- $\mathcal{D}$ is a set of elements.
- $\oplus$ is an associative and commutative operator with unit $\iota_{\oplus}$.
- $\otimes$ is an associative and commutative operator with unit $\iota_{\otimes}$, and distributes over $\oplus$.

□

Three examples of commutative semirings are $\{\mathsf{Num}, +, \times\}$, $\{\mathsf{Num}, \uparrow, +\}$, and $\{\mathsf{Bool}, \vee, \wedge\}$. It is worth noting that many dynamic programming algorithms are developed on the latter two commutative semirings.

Next, we define a class of functions on these commutative semirings. In the following, let k be a finite value, $\mathcal{D}^k$ denotes a set of finitely tupled values $(v_1, v_2, \ldots, v_k)$ where $v_i \in \mathcal{D}$.

Definition 5.10 Let $\{\mathcal{D}, \oplus, \otimes\}$ be a commutative semiring. Function $g :: \mathcal{D}^k \to \mathcal{D}$ is said to be a *linear polynomial function*, if it is defined in the following form:

$$g\ (x_1, x_2, \ldots, x_k) = (a_1 \otimes x_1) \oplus (a_2 \otimes x_2) \oplus \cdots \oplus (a_k \otimes x_k) \oplus a_{k+1}$$

where $a_1, a_2, \ldots, a_k$ and a_{k+1} are constants. □

Consider a tuple of k linear polynomial functions g_1, g_2, …, g_k where each function g_i is defined as $g_i\ (x_1, x_2, \ldots, x_k) = (a_{i1} \otimes x_1) \oplus (a_{i2} \otimes x_2) \oplus \cdots \oplus (a_{ik} \otimes x_k) \oplus a_{i(k+1)}$. We can specify the tuple of functions as follows. The operator $\times_{\otimes,\oplus}$ denotes matrix multiplication on commutative semiring $\{\mathcal{D}, \oplus, \otimes\}$, where operators $\times$ and $+$ in the ordinal matrix multiplication are replaced with operators $\otimes$ and $\oplus$ respectively. Similarly, the

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1k} & a_{1(k+1)} \\ a_{21} & a_{22} & \cdots & a_{2k} & a_{2(k+1)} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{k1} & a_{k2} & \cdots & a_{kk} & a_{k(k+1)} \\ \iota_\oplus & \iota_\oplus & \cdots & \iota_\oplus & \iota_\otimes \end{pmatrix} \ominus \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_k \end{pmatrix} =$$

$$\textbf{let} \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_k \\ \iota_\otimes \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1k} & a_{1(k+1)} \\ a_{21} & a_{22} & \cdots & a_{2k} & a_{2(k+1)} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{k1} & a_{k2} & \cdots & a_{kk} & a_{k(k+1)} \\ \iota_\oplus & \iota_\oplus & \cdots & \iota_\oplus & \iota_\otimes \end{pmatrix} \times_{\otimes,\oplus} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_k \\ \iota_\otimes \end{pmatrix} \textbf{ in } \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_k \end{pmatrix}$$

Figure 5.3. The definition of the semi-associative operator $\ominus$ for the tupled-ring property on commutative semiring $\{\mathcal{D}, \oplus, \otimes\}$.

operator $+_\oplus$ denotes matrix (vector) addition on commutative semiring $\{\mathcal{D}, \oplus, \otimes\}$. We may denote a tuple as a column vector for readability.

$$\begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_k \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1k} \\ a_{21} & a_{22} & \cdots & a_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ a_{k1} & a_{k2} & \cdots & a_{kk} \end{pmatrix} \times_{\otimes,\oplus} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_k \end{pmatrix} +_\oplus \begin{pmatrix} a_{1(k+1)} \\ a_{2(k+1)} \\ \vdots \\ a_{k(k+1)} \end{pmatrix}$$

By inserting another element for input and output, we obtain the following simpler definition with a matrix multiplication.

$$\begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_k \\ \iota_\otimes \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1k} & a_{1(k+1)} \\ a_{21} & a_{22} & \cdots & a_{2k} & a_{2(k+1)} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{k1} & a_{k2} & \cdots & a_{kk} & a_{k(k+1)} \\ \iota_\oplus & \iota_\oplus & \cdots & \iota_\oplus & \iota_\otimes \end{pmatrix} \times_{\otimes,\oplus} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_k \\ \iota_\otimes \end{pmatrix}$$

Now this matrix multiplication is an associative operator useful for deriving auxiliary functions of parallel programs. Before discussing the derivation of auxiliary functions for concrete functions, we introduce an operator $\ominus$ that bridges between a tuple with k elements and a $(k+1) \times (k+1)$ matrix (Figure 5.3). This $\ominus$ operator is indeed a semi-associative operator with its associative complement being matrix multiplication $\times_{\otimes,\oplus}$.

Then we discuss the condition and derive auxiliary functions for the $\mathsf{reduce}_\mathrm{b}$ and uAcc_b skeletons. First we define another class of functions in an analogy with the linear polynomial functions.

Definition 5.11 (Bi-linear Polynomial Function) Let $\{\mathcal{D}, \oplus, \otimes\}$ be a commutative semiring, and $\mathcal{B}$ be a set of elements. Function $g :: (\mathcal{D}^k, \mathcal{B}, \mathcal{D}^k) \to \mathcal{D}$ is said to be a *bi-linear polynomial function*, if it can be defined in the following two forms:

$$g\ ((l_1, l_2, \ldots, l_k), b, (r_1, r_2, \ldots, r_k)) = (a_1^r \otimes l_1) \oplus (a_2^r \otimes l_2) \oplus \cdots \oplus (a_k^r \otimes l_k) \oplus a_{k+1}^r$$

and

$$g\ ((l_1, l_2, \ldots, l_k), b, (r_1, r_2, \ldots, r_k)) = (a_1^l \otimes r_1) \oplus (a_2^l \otimes r_2) \oplus \cdots \oplus (a_k^l \otimes r_k) \oplus a_{k+1}^l$$

where values $a_1^r, a_2^r, \ldots, a_k^r$ and a_{k+1}^r are computed only from $r_1, r_2, \ldots, r_k$ and b; values $a_1^l, a_2^l, \ldots, a_k^l$ and a_{k+1}^l are computed only from $l_1, l_2, \ldots, l_k$ and b. □

Note that the class of the bi-linear polynomial functions is broader than that of the linear functions with respect to all the arguments. For example, the following function g

$$g((l_1), b, (r_1)) = l_1 \otimes r_1$$

is a bi-linear polynomial function but not a linear polynomial function with respect to l_1 and r_1.

Let function k for the $\mathsf{reduce_b}$ and $\mathsf{uAcc_b}$ skeletons be defined with a tuple of bi-linear polynomial functions. Then, we can denote such a function in the following two forms using the operator $\ominus$ bridging to the matrix multiplication. In the following, we may denote vectors and matrices in the bold font, for example we denote $\boldsymbol{l}$ and $\boldsymbol{r}$ for $(l_1, l_2, \ldots, l_k)$ and $(r_1, r_2, \ldots, r_k)$ respectively. Let $g_l\ (b, \boldsymbol{r})$ be a $(k+1) \times (k+1)$ matrix that contains coefficients $\{a_{ij}^r\}$, and $g_r\ (b, \boldsymbol{l})$ be a $(k+1) \times (k+1)$ matrix that contains coefficients $\{a_{ij}^l\}$.

$$k\ b\ \boldsymbol{l}\ \boldsymbol{r} = g_l\ (b, \boldsymbol{r}) \ominus \boldsymbol{l}$$
$$k\ b\ \boldsymbol{l}\ \boldsymbol{r} = g_r\ (b, \boldsymbol{l}) \ominus \boldsymbol{r}$$

Under this condition, we can derive auxiliary functions required in the $\mathsf{reduce_b}$ and $\mathsf{uAcc_r}$ skeleton by simply applying Theorem 5.6.

Theorem 5.12 (Tupled-Ring Property for $\mathsf{reduce_b}$ and $\mathsf{uAcc_b}$) *Let $\{\mathcal{D}, \oplus, \otimes\}$ be a commutative semiring, and function g be defined as a tuple of bi-linear functions as follows.*

$$g\ (l_1, l_2, \ldots, l_k)\ b\ (r_1, r_2, \ldots, r_k) = (x_1, x_2, \ldots, x_k)$$
$$\quad \textbf{where}\ x_i = (a_{i1}^r \otimes l_1) \oplus (a_{i2}^r \otimes l_2) \oplus \cdots \oplus (a_{ik}^r \otimes l_k) \oplus a_{i(k+1)}^r$$

$$g\ (l_1, l_2, \ldots, l_k)\ b\ (r_1, r_2, \ldots, r_k) = (y_1, y_2, \ldots, y_k)$$
$$\quad \textbf{where}\ y_i = (a_{i1}^l \otimes r_1) \oplus (a_{i2}^l \otimes r_2) \oplus \cdots \oplus (a_{ik}^l \otimes r_k) \oplus a_{i(k+1)}^l$$

Here, coefficients a_{ij}^r are computed only with $r_1, r_2, \ldots, r_k$, and b, and coefficients a_{ij}^l are computed only with $l_1, l_2, \ldots, l_k$ and b. Under this condition, the auxiliary functions $g = \langle \phi, \psi_n, \psi_l, \psi_r \rangle_{\mathrm{u}}$ can be systematically derived.

Proof. Let function $g_l\ (b, \boldsymbol{r})$ return a $(k+1) \times (k+1)$ matrix that consists of coefficients a_{ij}^l, and function $g_r\ (b, \boldsymbol{l})$ return a $(k+1) \times (k+1)$ matrix that consists of coefficients a_{ij}^r. Then the auxiliary functions can be defined as follows.

$$\phi\ b = (\boldsymbol{I}, b)$$

$$\psi_n\ \boldsymbol{l}\ (\boldsymbol{M}_n, b_n)\ r = \boldsymbol{M}_n \ominus k\ b_n\ l\ r$$

$$\psi_l\ (\boldsymbol{M}_l, b_l)\ (\boldsymbol{M}_n, b_n)\ r = (\boldsymbol{M}_n \times_{\otimes,\oplus} g_l\ (b, \boldsymbol{r}) \times_{\otimes,\oplus} \boldsymbol{M}_l, b_l)$$

$$\psi_r\ l\ (\boldsymbol{M}_n, b_n)\ (\boldsymbol{M}_r, b_r) = (\boldsymbol{M}_n \times_{\otimes,\oplus} g_r\ (b, \boldsymbol{l}) \times_{\otimes,\oplus} \boldsymbol{M}_r, b_r)$$

Here matrix $\boldsymbol{I}$ is the identity matrix whose diagonal elements are $\iota_\otimes$ and the other elements are $\iota_\oplus$, and the operator $\ominus$ is defined in Figure 5.3. □

Next, we discuss the derivation of auxiliary functions for the $\mathsf{dAcc_b}$ skeleton. Here, we assume that the two parameter functions g_l and g_r are defined as a tuple of linear polynomial functions. Under this condition, we can transform the functions and obtain two equations in the assumption of the derivation theorem (Theorem 5.7) using the semi-associative operator.

Theorem 5.13 (Tupled-Ring Property for $\mathsf{dAcc_b}$) *Let $\{\mathcal{D}, \oplus, \otimes\}$ be a commutative semiring, and functions g_l and g_r be defined as a tuple of linear polynomial functions as follows.*

$$g_l\ (c_1, c_2, \ldots, c_k)\ b = (l_1, l_2, \ldots, l_k)$$
$$\quad \textbf{where}\ l_i = (a^l_{i1} \otimes c_1) \oplus (a^l_{i2} \otimes c_2) \oplus \cdots \oplus (a^l_{ik} \otimes c_k) \oplus a^l_{i(k+1)}$$

$$g_r\ (c_1, c_2, \ldots, c_k)\ b = (r_1, r_2, \ldots, r_k)$$
$$\quad \textbf{where}\ r_i = (a^r_{i1} \otimes c_1) \oplus (a^r_{i2} \otimes c_2) \oplus \cdots \oplus (a^r_{ik} \otimes c_k) \oplus a^r_{i(k+1)}$$

Here, coefficients a^l_{ij} and a^r_{ij} are computed only from b. Under this condition, the auxiliary functions $(g_l, g_r) = \langle \phi_l, \phi_r, \psi_u, \psi_d \rangle_{\mathrm{d}}$ can be systematically derived.

Proof. Let function $g'_l\ b$ return a $(k+1) \times (k+1)$ matrix that consists of coefficients a^l_{ij}, and function g'_r return a $(k+1) \times (k+1)$ matrix that consists of coefficients a^r_{ij}. Then the following two equations hold with the semi-associative operator $\ominus$ defined in Figure 5.3.

$$g_l\ \boldsymbol{c}\ b = g'_l\ b \ominus \boldsymbol{c} \qquad g_r\ \boldsymbol{c}\ b = g'_r\ b \ominus \boldsymbol{c}$$

Using these functions, we can define the auxiliary functions as follows.

$$\phi_l\ b = g'_l\ b$$
$$\phi_r\ b = g'_r\ b$$
$$\psi_u\ \boldsymbol{M}\ \boldsymbol{M}' = \boldsymbol{M}' \times_{\otimes,\oplus} \boldsymbol{M}$$
$$\psi_d\ \boldsymbol{c}\ \boldsymbol{M} = \boldsymbol{M} \ominus \boldsymbol{c}$$

Here $\times_{\otimes,\oplus}$ is matrix multiplication on algebra $\{\mathcal{D}, \oplus, \otimes\}$. □

In the following section, we illustrate how the tupled-ring property works using the example of the party planning problem. The tupled-ring property is also used in deriving skeletal parallel programs for the maximum marking problem in Chapter 8 and for the XPath queries in Chapter 9.

5.3 Deriving Parallel Program for Party Planning Problem

In this section, we illustrate how we can systematically derive skeletal programs based on the diffusion theorems and the algebraic properties. We use the party planning problem in Introduction as our running example.

We show the problem statement of the party planning problem again.

> The president of a company wants to have a company party. To make the party fun for all attendees, the president does not want both an employee and his or her direct supervisor to attend. The company has a hierarchical structure; that is, the supervisory relations form a tree rooted at the president. The personnel office has ranked each employee with a conviviality rating, which is a real number. Given the structure of the company and the ratings of the employees, the problem is to select the guests so that the sum of their conviviality ratings is maximized.

The derivation of skeletal parallel program consists of the following four procedures.

1. Develop a sequential program.
2. Transform the sequential program into a form of diffusion theorems.
3. Apply diffusion theorems with some optimizations.
4. Derive auxiliary functions for each use of skeleton.

5.3.1 Deriving Skeletal Parallel Program for Party Planning Problem on Binary Trees

First we illustrate the derivation of a skeletal parallel program for binary trees.

Develop Sequential Program

At the first step, we develop a sequential program as recursive functions on the structure of binary trees. Here, the sequential program should not necessarily be efficient. We will perform optimizations in later steps.

For our running example, a known sequential algorithm is given based on the dynamic programming technique. The following recursive function ppp_{b} (named from *party planning problem*) defined with auxiliary functions ppp'_{b} and mis_{b} (named from *maximum independent sum*) solves the party planning problem. The function mis_{b} takes a binary tree and computes a pair of values (ms, us):

- ms: the maximum sum of non adjacent nodes under the condition that the root node of the tree is selected, and
- us: the maximum sum of non adjacent nodes under the condition the root node is *not* selected.

Based on the pair of values and the parent mark (p_marked), the function ppp'_{b} computes whether a node should be marked or not.

$$
\begin{aligned}
&ppp_{\mathrm{b}} :: \mathsf{BTree\ Int} \to \mathsf{BTree\ Bool}\\
&ppp_{\mathrm{b}}\ t = ppp'_{\mathrm{b}}\ \mathsf{False}\ t\\
\\
&ppp'_{\mathrm{b}}\ p\_marked\ (\mathsf{BLeaf}\ a)\\
&\quad = \mathbf{if}\ p\_marked\ \mathbf{then}\ \mathsf{BLeaf\ False}\\
&\qquad\qquad \mathbf{else}\ \ \mathbf{let}\ (ms, us) = mis_{\mathrm{b}}\ (\mathsf{BLeaf}\ a)\\
&\qquad\qquad\qquad\ \mathbf{in}\ \mathsf{BLeaf}\ (ms > us)\\
&ppp'_{\mathrm{b}}\ p\_marked\ (\mathsf{BNode}\ l\ b\ r)\\
&\quad = \mathbf{if}\ p\_marked\ \mathbf{then}\ \mathsf{BNode}\ (ppp'_{\mathrm{b}}\ \mathsf{False}\ l)\ \mathsf{False}\ (ppp'_{\mathrm{b}}\ \mathsf{False}\ r)\\
&\qquad\qquad \mathbf{else}\ \ \mathbf{let}\ (ms, us) = mis_{\mathrm{b}}\ (\mathsf{BNode}\ l\ b\ r)\\
&\qquad\qquad\qquad\qquad\ marked = ms > us\\
&\qquad\qquad\qquad\ \mathbf{in}\ \mathsf{BNode}\ (ppp'_{\mathrm{b}}\ mark\ l)\ mark\ (ppp'_{\mathrm{b}}\ mark\ r)\\
\\
&mis_{\mathrm{b}}\ (\mathsf{BLeaf}\ a) \quad\ = (a, 0)\\
&mis_{\mathrm{b}}\ (\mathsf{BNode}\ l\ b\ r) = \mathbf{let}\ (ms_l, us_l) = mis_{\mathrm{b}}\ l\\
&\qquad\qquad\qquad\qquad\ (ms_r, us_r) = mis_{\mathrm{b}}\ r\\
&\qquad\qquad\qquad\ \mathbf{in}\ (b + us_l + us_r, (ms_l \uparrow us_l) + (ms_r \uparrow us_r))
\end{aligned}
$$

Transform Sequential Program

We then transform the sequential program into a recursive form of diffusion theorems. We can apply several techniques to the transformation of sequential programs. There have been several techniques studied in the community of program transformation (or program calculation), such as, fusion transformation [30,51,58,126], tupling transformation [30,58], and condition normalization [29, 43]. It is worth noting that we need not be aware the parallelism in the program in this step.

In our running example, result tree is constructed in the **if**-branches for both cases ($\mathsf{BLeaf}\ a$) and ($\mathsf{BNode}\ l\ b\ r$), but this is off from the recursive forms of diffusion theorems. Noting that the two branches generates the same tree structure except for the values of the node and the accumulative parameter, we can bring the construction of trees out of the $\mathbf{if} - branches$ by using the following program transformation.

$$\mathbf{if}\ p\ \mathbf{then}\ f\ a\ \mathbf{else}\ f\ b \Longrightarrow \mathbf{let}\ x = \mathbf{if}\ p\ \mathbf{then}\ a\ \mathbf{else}\ b\ \mathbf{in}\ f\ x$$

With this transformation, we obtain the following recursive definition for the auxiliary function ppp'_{b}.

$$
\begin{aligned}
&ppp'_{\mathrm{b}}\ p\_marked\ (\mathsf{BLeaf}\ a)\\
&\quad = \mathbf{let}\ (ms, us) = mis_{\mathrm{b}}\ (\mathsf{BLeaf}\ a)\\
&\qquad\quad marked = \mathbf{if}\ p\_marked\ \mathbf{then}\ \mathsf{False}\ \mathbf{else}\ ms > us\\
&\qquad \mathbf{in}\ \mathsf{BLeaf}\ marked\\
&ppp'_{\mathrm{b}}\ p\_marked\ (\mathsf{BNode}\ l\ b\ r)\\
&\quad = \mathbf{let}\ (ms, us) = mis_{\mathrm{b}}\ (\mathsf{BNode}\ l\ b\ r)\\
&\qquad\quad marked = \mathbf{if}\ p\_marked\ \mathbf{then}\ \mathsf{False}\ \mathbf{else}\ ms > us\\
&\qquad \mathbf{in}\ \mathsf{BNode}\ (ppp'_{\mathrm{b}}\ marked\ l)\ marked\ (ppp'_{\mathrm{b}}\ marked\ r)
\end{aligned}
$$

Apply Diffusion Theorem with Optimization

Now we apply the diffusion theorem to derive a skeletal program. We first derive the parameter functions by simply matching the program with the recursive definition, and then apply a diffusion theorem by substituting the parameter functions. Since it is often the case that the derived skeletal program has redundant calls of skeletons due to the generic definition of diffusion theorems, we optimize the derived program by removing them or replacing them with more specific ones.

To our running example, we apply Theorem 5.2 with substitution of the map_b skeleton instead of the last $\mathsf{reduce}_\mathrm{b}$ skeleton. To apply the theorem, we need to derive the definition of the parameter functions. The tree homomorphisms h' and h'' are the same tree homomorphism mis_b.

$$
\begin{aligned}
&h' = h'' = mis_\mathrm{b} = (\![mis_l, mis_n]\!)_\mathrm{b} \\
&mis_l\ a = (a, 0) \\
&mis_n\ (ms_l, us_l)\ b\ (ms_r, us_r) = (b + us_l + us_r, (ms_l \uparrow us_l) + (ms_r \uparrow us_r))
\end{aligned}
$$

The functions g_l and g_r have the same definition. Let $mark$ be a function defined as

$$mark\ c\ (ms, us) = \textbf{if}\ c\ \textbf{then}\ \mathsf{False}\ \textbf{else}\ (ms > us)\ ,$$

then the definition of the function g_l and g_r is given as follows.

$$
\begin{aligned}
&g_l = g_r = g \\
&g\ c\ (\_, (ms, us)) = mark\ c\ (ms, us)
\end{aligned}
$$

The parameter functions for generating the result tree by the map_b skeleton are also defined in the same way.

$$
\begin{aligned}
&k_l\ (\_, c, (ms, us)) = mark\ c\ (ms, us) \\
&k_n\ (\_, c, (ms, us)) = mark\ c\ (ms, us)
\end{aligned}
$$

Substituting the parameter functions above, we obtain the following skeletal program. In the derivation we merged t'' into t', since the two tree homomorphisms h' and h'' are the same.

$$
\begin{aligned}
ppp'_\mathrm{b}\ p\_marked\ t = \textbf{let}\ t' &= \mathsf{uAcc}_\mathrm{b}\ mis_n\ (\mathsf{map}_\mathrm{b}\ mis_l\ id\ t) \\
ct &= \mathsf{dAcc}_\mathrm{b}\ (g, g)\ p\_marked\ (\mathsf{zipwith}_\mathrm{b}\ \_\ (,)\ t\ t') \\
zt &= \mathsf{zipwith}_\mathrm{b}\ tup\ tup\ t\ (\mathsf{zipwith}_\mathrm{b}\ (,)\ (,)\ ct\ t') \\
\textbf{in}\ &\mathsf{map}_\mathrm{b}\ k_l\ k_n\ zt \\
\textbf{where}\ &tup\ a\ (b, c) = (a, b, c)
\end{aligned}
$$

We further optimize the derived skeletal program by removing redundant skeletons. Since the first element of second argument of the function g for the dAcc_b skeleton is not used, we remove the $\mathsf{zipwith}_\mathrm{b}$ skeleton before the dAcc_b skeleton. Since the first element of functions k_l and k_n for the map_b skeleton is not used either, we remove the last $\mathsf{zipwith}_\mathrm{b}$ skeleton. Here, according to the changes we need to modify the parameter functions for the map_b skeleton a bit.

With these two optimizations we obtain the following skeletal program.

$$
\begin{aligned}
ppp'_{\mathrm{b}}\ p\_marked\ t = \textbf{let}\ & t' = \mathsf{uAcc}_{\mathrm{b}}\ mis_n\ (\mathsf{map}_{\mathrm{b}}\ mis_l\ id\ t) \\
& ct = \mathsf{dAcc}_{\mathrm{b}}\ (mark, mark)\ p\_marked\ t' \\
& zt = \mathsf{zipwith}_{\mathrm{b}}\ (,)\ (,)\ ct\ t' \\
\textbf{in}\ & \mathsf{map}_{\mathrm{b}}\ mark'\ mark'\ zt \\
\textbf{where}\ & mark'\ (c, (ms, us)) = mark\ c\ (ms, us)
\end{aligned}
$$

The last $\mathsf{map}_{\mathrm{b}}$ skeleton after the $\mathsf{zipwith}_{\mathrm{b}}$ skeleton can be fused into a single $\mathsf{zipwith}_{\mathrm{b}}$ skeleton. We substitute the initial value of the accumulative parameter False. With these optimization and transformation, we obtain the following simpler skeletal program for the main function ppp_{b}.

$$
\begin{aligned}
ppp_{\mathrm{b}}\ t = \textbf{let}\ & t' = \mathsf{uAcc}_{\mathrm{b}}\ mis_n\ (\mathsf{map}_{\mathrm{b}}\ mis_l\ id\ t) \\
& ct = \mathsf{dAcc}_{\mathrm{b}}\ (mark, mark)\ \mathsf{False}\ t' \\
\textbf{in}\ & \mathsf{zipwith}_{\mathrm{b}}\ mark\ mark\ ct\ t'
\end{aligned}
$$

Derive Auxiliary Functions for Skeletons

In the skeletal program, the $\mathsf{reduce}_{\mathrm{b}}$, $\mathsf{uAcc}_{\mathrm{b}}$, and $\mathsf{dAcc}_{\mathrm{b}}$ skeletons require auxiliary functions for their efficient parallel implementation, and thus finally we derive auxiliary functions for these skeletons. If the parameter functions used for skeletons satisfy one of the three algebraic properties, then we can derive the auxiliary functions systematically. For rose-tree skeletons $\mathsf{reduce}_{\mathrm{r}}$ and $\mathsf{uAcc}_{\mathrm{r}}$, we adopt the generation-and-test approach.

In the case of our running example, the skeletal program has one $\mathsf{uAcc}_{\mathrm{b}}$ skeleton and one $\mathsf{dAcc}_{\mathrm{b}}$ skeleton. The function mis_n used for the $\mathsf{uAcc}_{\mathrm{b}}$ skeleton returns a pair of values and is defined on a commutative ring $\{\mathsf{Int}, \uparrow, +\}$. Therefore we derive auxiliary functions based on the tupled-ring property (Theorem 5.12). First we transform the function mis_n into the bi-linear polynomial forms.

$$
\begin{aligned}
& mis_n\ (ms_l, us_l)\ b\ (ms_r, us_r) = (x_1, x_2) \\
& \quad \textbf{where}\ x_1 = (b + us_r) + us_l \\
& \qquad\ \ x_2 = ((ms_r \uparrow us_r) + ms_l) \uparrow ((ms_r \uparrow us_r) + us_l)
\end{aligned}
$$

$$
\begin{aligned}
& mis_n\ (ms_l, us_l)\ b\ (ms_r, us_r) = (y_1, y_2) \\
& \quad \textbf{where}\ y_1 = (b + us_l) + us_r \\
& \qquad\ \ y_2 = ((ms_l \uparrow us_l) + ms_r) \uparrow ((ms_l \uparrow us_l) + us_r)
\end{aligned}
$$

After we insert $((-\infty + ms_r) \uparrow\ )$ to the head of the first element and $(\ \uparrow (-\infty))$ to both tails, the function mis_n is indeed a bi-linear polynomial function.

$$
\begin{aligned}
& mis_n\ (ms_l, us_l)\ b\ (ms_r, us_r) = (x_1, x_2) \\
& \quad \textbf{where}\ x_1 = (-\infty + ms_l) \uparrow ((b + us_r) + us_l) \uparrow (-\infty) \\
& \qquad\ \ x_2 = ((ms_r \uparrow us_r) + ms_l) \uparrow ((ms_r \uparrow us_r) + us_l) \uparrow (-\infty)
\end{aligned}
$$

$$
\begin{aligned}
& mis_n\ (ms_l, us_l)\ b\ (ms_r, us_r) = (y_1, y_2) \\
& \quad \textbf{where}\ y_1 = (-\infty + ms_r) \uparrow ((b + us_l) + us_r) \uparrow (-\infty) \\
& \qquad\ \ y_2 = ((ms_l \uparrow us_l) + ms_r) \uparrow ((ms_l \uparrow us_l) + us_r) \uparrow (-\infty)
\end{aligned}
$$

Now we obtain the functions g_l and g_r in Theorem 5.12 by extracting the coefficients.

$$g_l\ \left(b, \begin{pmatrix} ms_l \\ us_l \end{pmatrix}\right) = \begin{pmatrix} -\infty & b + us_l & -\infty \\ ms_l \uparrow us_l & ms_l \uparrow us_l & -\infty \\ -\infty & -\infty & 0 \end{pmatrix}$$

$$g_r\ \left(b, \begin{pmatrix} ms_r \\ us_r \end{pmatrix}\right) = \begin{pmatrix} -\infty & b + us_r & -\infty \\ ms_r \uparrow us_r & ms_r \uparrow us_r & -\infty \\ -\infty & -\infty & 0 \end{pmatrix}$$

The operator $\ominus$ takes 3×3 matrix and a pair, and returns a pair as follows.

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \ominus \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} (a_{11} + x_1) \uparrow (a_{12} + x_2) \uparrow a_{13} \\ (a_{21} + x_1) \uparrow (a_{22} + x_2) \uparrow a_{23} \end{pmatrix}$$

Finally, we apply Theorem 5.12 and obtain the auxiliary functions successfully. The derived auxiliary functions ϕ^u, ψ^u_n, ψ^u_l, and ψ^u_r are as follows.

$$\phi^u\ b = \left(\begin{pmatrix} 0 & -\infty & -\infty \\ -\infty & 0 & -\infty \\ -\infty & -\infty & 0 \end{pmatrix}, b\right)$$

$$\begin{aligned} \psi^u_n\ \begin{pmatrix} l_1 \\ l_2 \end{pmatrix}\ &\left(\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}, b\right)\ \begin{pmatrix} r_1 \\ r_2 \end{pmatrix} \\ &= \mathbf{let}\ \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} b + l_2 + r_2 \\ (l_1 \uparrow l_2) + (r_1 \uparrow r_2) \end{pmatrix} \\ &\quad \mathbf{in}\ \begin{pmatrix} (a_{11} + x_1) \uparrow (a_{12} + x_2) \uparrow a_{13} \\ (a_{21} + x_1) \uparrow (a_{22} + x_2) \uparrow a_{23} \end{pmatrix} \end{aligned}$$

$$\begin{aligned} &\psi^u_l\ (\boldsymbol{A}^l, b^l)\ (\boldsymbol{A}^n, b^n)\ \begin{pmatrix} r_1 \\ r_2 \end{pmatrix} \\ &\quad = \left(\boldsymbol{A}^n \times_{+,\uparrow} \begin{pmatrix} -\infty & b^n + r_2 & -\infty \\ r_1 \uparrow r_2 & r_1 \uparrow r_2 & -\infty \\ -\infty & -\infty & 0 \end{pmatrix} \times_{+,\uparrow} \boldsymbol{A}^l, b^l\right) \end{aligned}$$

$$\begin{aligned} &\psi^u_r\ \begin{pmatrix} l_1 \\ l_2 \end{pmatrix}\ (\boldsymbol{A}^n, b^n)\ (\boldsymbol{A}^r, b^r) \\ &\quad = \left(\boldsymbol{A}^n \times_{+,\uparrow} \begin{pmatrix} -\infty & b^n + l_2 & -\infty \\ l_1 \uparrow l_2 & l_1 \uparrow l_2 & -\infty \\ -\infty & -\infty & 0 \end{pmatrix} \times_{+,\uparrow} \boldsymbol{A}^r, b^r\right) \end{aligned}$$

We can simplify the auxiliary functions by finding the constant through the computation of the $\mathsf{uAcc_b}$ skeleton. For this case, only the upper-left 2×2 values may change. Therefore, we can remove the third column and the third row from the matrices. The definition of the optimized auxiliary functions are given in Figure 5.4. We will study how we can find these unnecessary values in Section 7.3.

Next, we derive auxiliary functions for the function g used in the $\mathsf{dAcc_b}$ skeleton. The range of the function g is finite, i.e., Bool consisting of two values True and False. Thus, we apply the theorem of finiteness property to derive auxiliary functions.

We first rewrite the function *mark* using the **case**-statement.

$$mark\ c\ (ms, us) = \mathbf{case}\ c\ \mathbf{of}\ \mathsf{True} \to \mathsf{False}; \mathsf{False} \to (ms > us)$$

Using this definition, we can derive the auxiliary functions ϕ^d, ψ_u^d and ψ_d^d by substituting the function *mark* above. The auxiliary function ϕ^d is derived by extracting the results of two cases.

$$\phi^d\ (b_1, b_2) = (\mathsf{False}, (b_1 > b_2))$$

The auxiliary function ψ_u^d is defined as follows.

$$\begin{aligned}\psi_u^d\ (b_1^1, b_2^1)\ (b_1^2, b_2^2) = (b_1', b_2') \\ \mathbf{where}\ b_1' = \mathbf{case}\ b_1^1\ \mathbf{of}\ \mathsf{True} \to b_1^2; \mathsf{False} \to b_2^2 \\ b_2' = \mathbf{case}\ b_2^1\ \mathbf{of}\ \mathsf{True} \to b_1^2; \mathsf{False} \to b_2^2\end{aligned}$$

The auxiliary function ψ_d^d is the *mark* function.

$$\psi_d^d\ c\ (b_1, b_2) = \mathbf{if}\ c\ \mathbf{then}\ b_1\ \mathbf{else}\ b_2$$

Substituting the auxiliary functions for the skeletons, we can derive a skeletal parallel program. Figure 5.4 shows the derived skeletal parallel program for the party planning problem on binary trees.

5.3.2 Deriving Skeletal Parallel Program for Party Planning Problem on Rose Trees

We then derive a skeletal parallel program for the party planning problem on rose trees. The outline of the derivation is almost the same except that the characteristic functions for the extended distributivity is derived by the generalization-and-test technique.

Develop Sequential Program

The sequential program that solves the party planning problem for rose trees is almost the same as that for binary trees. Let mis_{r} be a function that takes a rose tree and computes the following two values (m, u).

- m: The maximum sum of non adjacent nodes under the condition that the root node is selected, and
- u: The maximum sum of non adjacent nodes under the condition that the root node is *not* selected.

With this function mis_{r}, we can define a sequential program ppp_{r} as follows. The accumulative parameter for the auxiliary function ppp'_{r} represents whether the parent node is selected or not.

$$\begin{aligned}&ppp_{\mathrm{r}} :: \mathsf{RTree\ Int} \to \mathsf{RTree\ Bool} \\ &ppp_{\mathrm{r}}\ t = ppp'_{\mathrm{r}}\ \mathsf{False}\ t\end{aligned}$$

$$
\begin{array}{l}
ppp_{\mathrm{b}}\ t = \textbf{let}\ t' = \mathsf{uAcc}_{\mathrm{b}}\ \langle \phi^u, \psi_n^u, \psi_l^u, \psi_r^u \rangle_{\mathrm{u}}\ (\mathsf{map}_{\mathrm{b}}\ (\lambda a.(a,0))\ id\ t) \\
\qquad\qquad ct = \mathsf{dAcc}_{\mathrm{b}}\ \langle \phi^d, \phi^d, \psi_u^d, \psi_d^d \rangle_{\mathrm{d}}\ \mathsf{False}\ t' \\
\qquad\quad \textbf{in}\ \mathsf{zipwith}_{\mathrm{b}}\ mark\ mark\ ct\ t'
\end{array}
$$

$$
mark\ c\ (ms, us) = \textbf{if}\ c\ \textbf{then}\ \mathsf{False}\ \textbf{else}\ ms > us
$$

$$
\phi^u\ b = \left(\begin{pmatrix} 0 & -\infty \\ -\infty & 0 \end{pmatrix}, b \right)
$$

$$
\begin{array}{l}
\psi_n^u \begin{pmatrix} l_1 \\ l_2 \end{pmatrix} \left(\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}, b \right) \begin{pmatrix} r_1 \\ r_2 \end{pmatrix} \\
\quad = \textbf{let} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} b + l_2 + r_2 \\ (l_1 \uparrow l_2) + (r_1 \uparrow r_2) \end{pmatrix} \\
\qquad \textbf{in} \begin{pmatrix} (a_{11} + x_1) \uparrow (a_{12} + x_2) \\ (a_{21} + x_1) \uparrow (a_{22} + x_2) \end{pmatrix}
\end{array}
$$

$$
\begin{array}{l}
\psi_l^u \left(\begin{pmatrix} a_{11}^l & a_{12}^l \\ a_{21}^l & a_{22}^l \end{pmatrix}, b^l \right) \left(\begin{pmatrix} a_{11}^n & a_{12}^n \\ a_{21}^n & a_{22}^n \end{pmatrix}, b_n \right) \begin{pmatrix} r_1 \\ r_2 \end{pmatrix} \\
\quad = \left(\begin{pmatrix} a_{11}^n & a_{12}^n \\ a_{21}^n & a_{22}^n \end{pmatrix} \times_{+,\uparrow} \begin{pmatrix} -\infty & b^n + r_2 \\ r_1 \uparrow r_2 & r_1 \uparrow r_2 \end{pmatrix} \times_{+,\uparrow} \begin{pmatrix} a_{11}^l & a_{12}^l \\ a_{21}^l & a_{22}^l \end{pmatrix}, b^l \right)
\end{array}
$$

$$
\begin{array}{l}
\psi_r^u \begin{pmatrix} l_1 \\ l_2 \end{pmatrix} \left(\begin{pmatrix} a_{11}^n & a_{12}^n \\ a_{21}^n & a_{22}^n \end{pmatrix}, b_n \right) \left(\begin{pmatrix} a_{11}^r & a_{12}^r \\ a_{21}^r & a_{22}^r \end{pmatrix}, b^r \right) \\
\quad = \left(\begin{pmatrix} a_{11}^n & a_{12}^n \\ a_{21}^n & a_{22}^n \end{pmatrix} \times_{+,\uparrow} \begin{pmatrix} -\infty & b^n + l_2 \\ l_1 \uparrow l_2 & l_1 \uparrow l_2 \end{pmatrix} \times_{+,\uparrow} \begin{pmatrix} a_{11}^r & a_{12}^r \\ a_{21}^r & a_{22}^r \end{pmatrix}, b^r \right)
\end{array}
$$

$$
\phi^d\ (ms, us) = (\mathsf{False}, (ms > us))
$$

$$
\psi_d^d\ c\ (b_1, b_2) = \textbf{case}\ c\ \textbf{of}\ \mathsf{True} \to b_1; \mathsf{False} \to b_2;
$$

$$
\begin{array}{l}
\psi_u^d\ (b_1^1, b_2^1)\ (b_1^2, b_2^2) = (b_1', b_2') \\
\quad \textbf{where}\ b_1' = \textbf{case}\ b_1^1\ \textbf{of}\ \mathsf{True} \to b_1^2; \mathsf{False} \to b_2^2 \\
\qquad\quad b_2' = \textbf{case}\ b_2^1\ \textbf{of}\ \mathsf{True} \to b_1^2; \mathsf{False} \to b_2^2
\end{array}
$$

Figure 5.4. The derived parallel skeletal program for the party planning problem on binary trees.

$$
\begin{aligned}
&ppp'_{\mathrm{r}}\ p\_marked\ (\mathsf{RNode}\ a\ ts)\\
&\quad = \mathbf{if}\ p\_marked\ \mathbf{then}\ \mathsf{RNode}\ \mathsf{False}\ [ppp'_{\mathrm{r}}\ \mathsf{False}\ t_i \mid i \in [1..\#ts]]\\
&\qquad\qquad \mathbf{else\ let}\ (m,u) = mis_{\mathrm{r}}\ (\mathsf{RNode}\ a\ ts)\\
&\qquad\qquad\qquad marked = m > u\\
&\qquad\qquad\quad \mathbf{in}\ \mathsf{RNode}\ marked\ [ppp'_{\mathrm{r}}\ marked\ t_i \mid i \in [1..\#ts]]
\end{aligned}
$$

The function mis_{r} is defined as follows by using the list-comprehension notation.

$$
\begin{aligned}
mis_{\mathrm{r}}\ (\mathsf{RNode}\ a\ ts) = \mathbf{let}\ & u' = \textstyle\sum_{+}[u_i \mid (\_, u_i) = mis_{\mathrm{r}}\ t_i, i \in [1..\#ts]]\\
& w' = \textstyle\sum_{+}[m_i \uparrow u_i \mid (m_i, u_i) = mis_{\mathrm{r}}\ t_i, i \in [1..\#ts]]\\
\mathbf{in}\ & (a + u', w' \uparrow 0)
\end{aligned}
$$

In the following of this section, we derive a skeletal parallel program from this sequential program.

Transform Sequential Program

We first transform the sequential program into a form of diffusion theorems.

In the function ppp'_{r}, a result tree is constructed in the **if**-branches. By applying the transformation rule in the previous section, we can bring the tree construction out of the **if**-branches as shown in the following program.

$$
\begin{aligned}
&ppp'_{\mathrm{r}}\ p\_marked\ (\mathsf{RNode}\ a\ ts)\\
&\quad = \mathbf{let}\ (m,u) = mis_{\mathrm{r}}\ (\mathsf{RNode}\ a\ ts)\\
&\qquad\quad marked = \mathbf{if}\ p\_marked\ \mathbf{then}\ \mathsf{False}\ \mathbf{else}\ m > u\\
&\qquad \mathbf{in}\ \mathsf{RNode}\ marked\ [ppp'_{\mathrm{r}}\ marked\ t_i \mid i \in [1..\#ts]]
\end{aligned}
$$

In the function mis_{r}, the list of children is traversed twice for computing values u' and w'. We merge these two traversals into one by introducing the following function p and operator $\otimes$.

$$
\begin{aligned}
&p\ (m,u) = (u, m \uparrow u)\\
&(u,w) \otimes (u',w') = (u + u', w + w')
\end{aligned}
$$

The operator $\otimes$ is associative and commutative due to the associativity and commutativity of $+$. Using them, we can obtain the following definition of the function mis_{r}.

$$
\begin{aligned}
&mis_{\mathrm{r}}\ (\mathsf{RNode}\ a\ ts)\\
&\quad = \mathbf{let}\ (u',w') = \textstyle\sum_{\otimes}[g\ (mis_{\mathrm{r}}\ t_i) \mid i \in [1..\#ts]]\\
&\quad\ \mathbf{in}\ (a + u', w' \uparrow 0)
\end{aligned}
$$

Unfortunately, this definition of mis_{r} is not in the form of parallelizable homomorphism, because the function g is applied between the function mis_{r} and reduction by $\otimes$. To transform the function into a parallelizable homomorphism, we fuse the functions p and mis_{r}. Let mis'_{r} be the fused function, $mis'_{\mathrm{r}} = p \circ mis_{\mathrm{r}}$. To fuse the functions, we need to find a function p' that computes the necessary value in the ppp'_{r} function.

$$
p'\ (p\ (m,u)) = m > u
$$

After unfolding the function p, we can successfully find a definition of the function p' as follows.

$$p'\ (u, w) = u \neq w$$

Therefore, we can fuse the functions p and mis_{r} and obtain the following sequential program.

$$\begin{aligned}
&ppp'_{\mathrm{r}}\ p_marked\ (\mathsf{RNode}\ a\ ts)\\
&\quad = \mathbf{let\ if}\ p_marked\ \mathbf{then}\ \mathsf{False}\ \mathbf{else}\ p'\ (mis'_{\mathrm{r}}\ (\mathsf{RNode}\ a\ ts))\\
&\quad\ \ \mathbf{in}\ \mathsf{RNode}\ marked\ [ppp'_{\mathrm{r}}\ marked\ t_i \mid i \in [1..\#ts]]\\
&\quad\ \ \mathbf{where}\ p'\ (u, w) = u \neq w
\end{aligned}$$

$$\begin{aligned}
&mis'_{\mathrm{r}}\ (\mathsf{RNode}\ a\ ts)\\
&\quad = \mathbf{let}\ (u', w') = \textstyle\sum_{\otimes}[mis'_{\mathrm{r}}\ t_i \mid i \in [1..\#ts]]\\
&\quad\ \ \mathbf{in}\ (w' \uparrow 0, (a + u') \uparrow w' \uparrow 0)\\
&\quad\ \ \mathbf{where}\ (u, w) \otimes (u', w') = (u + u', w + w')
\end{aligned}$$

The function mis'_{r} may be a parallelizable homomorphism, if the exist function k and operator $\oplus$ such that

- $k\ a \oplus (u', w') = (w' \uparrow 0, (a + u') \uparrow w' \uparrow 0)$ holds, and
- operator $\otimes$ is extended distributive over operator $\oplus$.

Apply Diffusion Theorems with Optimization

We then apply diffusion theorems and derive a skeletal program. By comparing the sequential definition with diffusion theorems, we can find that the sequential definition is almost the same as the form of Corollary 5.2. The difference is that the result value of a node is given not from the original value but the result of bottom-up computation. Therefore, we apply Corollary 5.2 with minor modifications and obtain the following skeleton program.

$$\begin{aligned}
&ppp_{\mathrm{r}}\ t = \mathbf{let}\ t' = \mathsf{uAcc}_{\mathrm{r}}\ (\oplus)\ (\otimes)\ (\mathsf{map}_{\mathrm{r}}\ k\ t)\\
&\qquad\qquad\ \ t'' = \mathsf{dAcc}_{\mathrm{r}}\ g\ (\mathsf{zipwith}_{\mathrm{r}}\ (,)\ t\ t')\\
&\qquad\ \ \mathbf{in}\ \mathsf{zipwith}_{\mathrm{r}}\ h\ t'\ t''\\
&\qquad\ \ \mathbf{where}\ g\ p_marked\ (_, (u, w)) = \mathbf{if}\ p_marked\ \mathbf{then}\ \mathsf{False}\ \mathbf{else}\ u \neq w\\
&\qquad\qquad\qquad h\ (u, w)\ p_marked = \mathbf{if}\ p_marked\ \mathbf{then}\ \mathsf{False}\ \mathbf{else}\ u \neq w
\end{aligned}$$

In the derived skeletal program, the first value of the second argument of g is not used. Therefore, we can simplify the definition by removing the first $\mathsf{zipwith}_{\mathrm{r}}$ skeleton and modifying the function g.

$$\begin{aligned}
&ppp_{\mathrm{r}}\ t = \mathbf{let}\ t' = \mathsf{uAcc}_{\mathrm{r}}\ (\oplus)\ (\otimes)\ (\mathsf{map}_{\mathrm{r}}\ k\ t)\\
&\qquad\ \ \mathbf{in}\ \mathsf{zipwith}_{\mathrm{r}}\ h\ t'\ (\mathsf{dAcc}_{\mathrm{r}}\ g\ t')\\
&\qquad\ \ \mathbf{where}\ g\ p_marked\ (u, w) = \mathbf{if}\ p_marked\ \mathbf{then}\ \mathsf{False}\ \mathbf{else}\ u \neq w\\
&\qquad\qquad\qquad h\ (u, w)\ p_marked = \mathbf{if}\ p_marked\ \mathbf{then}\ \mathsf{False}\ \mathbf{else}\ u \neq w
\end{aligned}$$

Derive Auxiliary Functions for Skeletons

Finally, we derive auxiliary functions and characteristic functions for rose-tree skeletons.

For the $\mathsf{dAcc_r}$ skeleton, we need to find auxiliary functions ϕ, ψ_u, and ψ_d such that the following two equations hold.

$$g\ c\ n = \psi_d\ c\ (\phi\ n)$$
$$\psi_d\ (\psi_d\ c\ n)\ m = \psi_d\ c\ (\psi_u\ n\ m)$$

This condition is a subset of the condition for the $\mathsf{dAcc_b}$ skeleton and thus we can use the finiteness property to derive the auxiliary functions for the $\mathsf{dAcc_r}$ skeleton.

We consider a tabled function of the following form

$$\lambda c.\mathbf{case}\ c\ \mathbf{of}\ \mathsf{True} \to c_1'; \mathsf{False} \to c_2'\ ,$$

where the two values c_1' and c_2' are boolean values. Then we can derive auxiliary functions as follows.

$$\phi\ (u, w) = (\mathsf{False}, u \neq w)$$

$$\begin{aligned}\psi_u\ (a, b)\ (a', b') = (a'', b'')\\ \mathbf{where}\ a'' = \mathbf{case}\ a\ \mathbf{of}\ \mathsf{True} \to a'; \mathsf{False} \to b'\\ b'' = \mathbf{case}\ b\ \mathbf{of}\ \mathsf{True} \to a'; \mathsf{False} \to b'\end{aligned}$$

$$\psi_d\ c\ (a, b) = \mathbf{if}\ c\ \mathbf{then}\ a\ \mathbf{else}\ b$$

We then find characteristic functions of extended distributivity of the operator $\otimes$ over operator $\oplus$. The condition on the function k and $\oplus$ is to satisfy the following equation.

$$k\ a \oplus (u, w) = (w \uparrow 0, (a + u) \uparrow w \uparrow 0)$$

Let us consider the most simplest definition of k, i.e., $k\ a = a$. Then, the operator $\oplus$ can be defined as

$$a \oplus (u, w) = (w \uparrow 0, (a + u) \uparrow w \uparrow 0)\ .$$

Under this definition, we check whether the operators have closure property or not. Since the operator $\otimes$ is commutative, we calculate the both sides of equation in Lemma 4.1.

$$\begin{aligned}&(\lambda(x_u, x_w).a \oplus ((u, w) \otimes (x_u, x_w))) \circ (\lambda(x_u, x_w).a' \oplus ((u', w') \otimes (x_u, x_w)))\\ &= \lambda(x_u, x_w).a \oplus ((u, w) \otimes (a' \oplus ((u', w') \otimes (x_u, x_w))))\\ &= \lambda(x_u, x_w).a \oplus ((u, w) \otimes (a' \oplus (u' + x_u, w' + x_w)))\\ &= \lambda(x_u, x_w).a \oplus ((u, w) \otimes ((w' + x_w) \uparrow 0, (a + u' + x_u) \uparrow (w' + x_w) \uparrow 0))\\ &= \lambda(x_u, x_w).a \oplus (((u + w' + x_w) \uparrow u, (w + a + u' + x_u) \uparrow (w + w' + x_w) \uparrow w))\\ &= \lambda(x_u, x_w).((w + a + u' + x_u) \uparrow (w + w' + x_w) \uparrow w \uparrow 0,\\ &\qquad (a + u + w' + x_w) \uparrow (a + u) \uparrow (w + a + u' + x_u) \uparrow (w + w' + x_w) \uparrow w \uparrow 0)\\ &= \lambda(x_u, x_w).(((w + a + u') + x_u) \uparrow ((w + w') + x_w) \uparrow (w \uparrow 0),\\ &\qquad ((w + a + u') + x_u) \uparrow (((a + u + w') \uparrow (w + w')) + x_w) \uparrow ((a + u) \uparrow w \uparrow 0))\end{aligned}$$

$$
\begin{aligned}
&\lambda(x_u, x_w).A \oplus ((U, W) \otimes (x_u, x_w)) \\
&= \lambda(x_u, x_w).A' \oplus ((U', W') \otimes (x_u, x_w)) \\
&= \lambda(x_u, x_w).A' \oplus (U' + x_u, W' + x_w) \\
&= \lambda(x_u, x_w).((W' + x_w) \uparrow 0, (A + U' + x_u) \uparrow (W' + x_w) \uparrow 0)
\end{aligned}
$$

As we can see by comparing the two sides, the two sides are different in the following two points.

- For the first value of the pair, the term x_u appears on the left side but it does not on the right side.
- The constant values on the right side corresponds to terms including variables.

To absorb these difference, we give another definition for function k and operator $\oplus$. For reasons of readability, we denote the tuples as matrices.

$$
k\ a = \begin{pmatrix} -\infty & 0 & 0 \\ a & 0 & -\infty \end{pmatrix}
$$

$$
\begin{pmatrix} a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 \end{pmatrix} \oplus \begin{pmatrix} u \\ w \end{pmatrix} = \begin{pmatrix} (a_1 + u) \uparrow (a_2 + w) \uparrow a_3 \\ (a_4 + u) \uparrow (a_5 + w) \uparrow a_6 \end{pmatrix}
$$

Now we can prove the extended distributivity of $\otimes$ over $\oplus$ by the calculations below.

$$
\begin{aligned}
&\left(\lambda \begin{pmatrix} x_u \\ x_w \end{pmatrix}. \begin{pmatrix} a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 \end{pmatrix} \oplus \left(\begin{pmatrix} u \\ w \end{pmatrix} \otimes \begin{pmatrix} x_u \\ x_w \end{pmatrix} \right)\right) \\
&\qquad \circ \left(\lambda \begin{pmatrix} x_u \\ x_w \end{pmatrix}. \begin{pmatrix} a'_1 & a'_2 & a'_3 \\ a'_4 & a'_5 & a'_6 \end{pmatrix} \oplus \left(\begin{pmatrix} u' \\ w' \end{pmatrix} \otimes \begin{pmatrix} x_u \\ x_w \end{pmatrix} \right)\right) \\
&= \begin{pmatrix}
\begin{array}{l}
(((a_1 + u + a'_1 + u') \uparrow (a_2 + w + a'_4 + u')) + x_u) \\
\quad \uparrow (((a_1 + u + a'_2 + w') \uparrow (a_2 + w + a'_5 + w')) + x_w) \\
\quad\quad \uparrow ((a_1 + u + a'_3) \uparrow (a_2 + w + a'_6) \uparrow a_3)
\end{array} \\
\begin{array}{l}
(((a_4 + u + a'_1 + u') \uparrow (a_5 + w + a'_4 + u')) + x_u) \\
\quad \uparrow (((a_4 + u + a'_2 + w') \uparrow (a_5 + w + a'_5 + w')) + x_w) \\
\quad\quad \uparrow ((a_4 + u + a'_3) \uparrow (a_5 + w + a'_6) \uparrow a_6)
\end{array}
\end{pmatrix}
\end{aligned}
$$

$$
\begin{aligned}
&\lambda \begin{pmatrix} x_u \\ x_w \end{pmatrix}. \begin{pmatrix} A_1 & A_2 & A_3 \\ A_4 & A_5 & A_6 \end{pmatrix} \oplus \left(\begin{pmatrix} U \\ W \end{pmatrix} \otimes \begin{pmatrix} x_u \\ x_w \end{pmatrix} \right) \\
&\qquad = \begin{pmatrix} (A_1 + U + x_u) \uparrow (A_2 + W + x_w) \uparrow A_3 \\ (A_4 + U + x_u) \uparrow (A_5 + W + x_w) \uparrow A_6 \end{pmatrix}
\end{aligned}
$$

Based on these calculations, we can derive characteristic functions of the extended distributivity easily by substituting zeros for U and W.

Therefore, we have succeeded in deriving a skeletal parallel program for the party planning problem on rose trees. Figure 5.5 summarize the derived program.

$$ppp_{\mathrm{r}}\ t = \textbf{let}\ t' = \mathsf{uAcc}_{\mathrm{r}}\ (\oplus)\ (\otimes)\ (\mathsf{map}_{\mathrm{r}}\ k\ t)$$
$$\textbf{in}\ \mathsf{zipwith}_{\mathrm{r}}\ h\ t'\ (\mathsf{dAcc}_{\mathrm{r}}\ g\ t')$$

$$g\ p\_marked\ (u, w) = \textbf{if}\ p\_marked\ \textbf{then}\ \mathsf{False}\ \textbf{else}\ u \neq w$$

$$h\ (u, w)\ p\_marked = \textbf{if}\ p\_marked\ \textbf{then}\ \mathsf{False}\ \textbf{else}\ u \neq w$$

$$k\ a = (-\infty, 0, 0, a, 0, -\infty)$$

$$(a_1, a_2, a_3, a_4, a_5, a_6) \oplus (u, w) = ((a_1 + u) \uparrow (a_2 + w) \uparrow a_3, (a_4 + u) \uparrow (a_5 + w) \uparrow a_6)$$

$$(u, w) \otimes (u', w') = (u + u', w + w')$$

$$p_1\ ((a_1, a_2, a_3, a_4, a_5, a_6), (u_1, w_1), (u_2, w_2), (a'_1, a'_2, a'_3, a'_4, a'_5, a'_6), (u'_1, w'_1), (u'_2, w'_2))$$
$$= \begin{pmatrix} ((a_1 + u_1 + u_2 + a'_1 + u'_1 + u'_2) \uparrow (a_2 + w_1 + w_2 + a'_4 + u'_1 + u'_2)) \\ ((a_1 + u_1 + u_2 + a'_2 + w'_1 + w'_2) \uparrow (a_2 + w_1 + w_2 + a'_5 + w'_1 + w'_2)) \\ (a_1 + u_1 + u_2 + a'_3) \uparrow (a_2 + w_1 + w_2 + a'_6) \uparrow a_3 \\ ((a_4 + u_1 + u_2 + a'_1 + u'_1 + u'_2) \uparrow (a_5 + w_1 + w_2 + a'_4 + u'_1 + u'_2)) \\ ((a_4 + u_1 + u_2 + a'_2 + w'_1 + w'_2) \uparrow (a_5 + w_1 + w_2 + a'_5 + w'_1 + w'_2)) \\ (a_4 + u_1 + u_2 + a'_3) \uparrow (a_5 + w_1 + w_2 + a'_6) \uparrow a_6 \end{pmatrix}$$

$$p_2\ ((a_1, a_2, a_3, a_4, a_5, a_6), (u_1, w_1), (u_2, w_2), (a'_1, a'_2, a'_3, a'_4, a'_5, a'_6), (u'_1, w'_1), (u'_2, w'_2)) = (0, 0)$$
$$p_3\ ((a_1, a_2, a_3, a_4, a_5, a_6), (u_1, w_1), (u_2, w_2), (a'_1, a'_2, a'_3, a'_4, a'_5, a'_6), (u'_1, w'_1), (u'_2, w'_2)) = (0, 0)$$

$$\phi\ (u, w) = (\mathsf{False}, u \neq w)$$

$$\psi_u\ (a, b)\ (a', b') = (a'', b'')$$
$$\textbf{where}\ a'' = \textbf{case}\ a\ \textbf{of}\ \mathsf{True} \to a'; \mathsf{False} \to b'$$
$$b'' = \textbf{case}\ b\ \textbf{of}\ \mathsf{True} \to a'; \mathsf{False} \to b'$$

$$\psi_d\ c\ (a, b) = \textbf{if}\ c\ \textbf{then}\ a\ \textbf{else}\ b$$

Figure 5.5. The derived parallel program for the party planning problem on rose trees. The functions p_1, p_2 and p_3 are characteristic functions of the extended distributivity of $\otimes$ over $\oplus$. The functions ϕ, ψ_u, and ψ_d are auxiliary functions for the parallel implementation of the $\mathsf{dAcc}_{\mathrm{r}}$ skeleton.

5.4 Short Summary

In this chapter, we have developed theorems for deriving skeletal parallel programs from sequential programs. Sequential programs are often developed recursively along the input data structures and the diffusion theorems show many recursive functions can be decomposed into combinations of tree skeletons. After the decomposition into skeletal programs, we apply theorems on three algebraic properties, finiteness property, extended-ring property, and tupled-ring property to the parameter functions to derive auxiliary functions. If the parameter functions satisfy one of the properties, then we can derive auxiliary functions systematically.

To illustrate our methodology for deriving skeletal parallel programs, we have derived skeletal parallel programs for the party planning problems on binary trees and on rose trees. Though the derived parallel programs are not trivial, we can derive them systematically from sequential recursive programs.

We have formalized algebraic properties for the derivation of auxiliary functions for binary-tree skeletons. Whether we can extend the theorems to the cases of rose-trees remains as an open problem.

Chapter 6

Implementation of Binary-Tree Skeletons

Though there have been many studies on tree contraction algorithms, they have not been used so often in the development of parallel applications so far. There are mainly two problems. Firstly, many tree contraction algorithms have been developed on shared-memory parallel models such as PRAMs [2, 8, 35, 98], and only a few algorithms have been developed on the distributed-memory parallel models [94, 95]. Distributing data is important for efficient parallel programs, but distributing tree structures is a non trivial problem. Secondly, for tree contraction algorithms only a few implementations are provided as libraries. As far as we are aware, an implementation of tree contraction algorithms is developed on multithreaded parallel language cilk [111, 123], and a parallel graph library CGM graph/CGM lib [38] also has an implementation of tree contraction algorithms. However, these two implementations are based on shared-memory algorithms and therefore for the distributed-memory environments we need another implementation of tree contraction algorithms.

We have implemented the parallel binary-tree skeletons so that the implementation suits especially for the distributed-memory parallel computers. Compared with the implementations so far mainly for shared-memory parallel computers, our implementation has the following three features.

- *Less overheads of parallelism.* Locality is one of the most important properties in developing efficient parallel programs especially for distributed-memory computers. We borrow the idea of m-bridges [45, 112] in the basic graph-theory to divide binary trees with high locality. Furthermore, we call the auxiliary functions as few as possible, to minimize the overhead of the parallel implementation.

- *High sequential performance.* The performance of the sequential computation parts is as important as that of the communication parts for efficient parallel programs. We

Part of this chapter was presented in [83], and the full discussion of this chapter is given as a technical report [84].

represent a local segment as a serialized array and implemented local computations of tree skeletons with loops rather than recursive functions. This implementation choice provides quite high performance in the sequential computation parts.

- *Cost model.* One advantage of skeletal parallel programming is the predictability of the performance of skeletal programs, where the cost of skeletal programs is given based on cost models of skeletons. Not only we have implemented the tree skeletons efficiently, but also we have formalized a cost model of our parallel implementation. The cost model also helps us to divide binary trees with good load balance.

In this chapter, we give an implementation of the binary-tree skeletons for distributed-memory parallel computers. In Section 6.1, we explain how to represent distributed binary-trees with high locality after reviewing basic graph-theoretic results. In Section 6.2, we develop an efficient implementation and a cost model of the binary-tree skeletons. We discuss the division of binary trees in more details based on the cost model in Section 6.3. Finally we report experimental results in Section 6.4, and summarize this chapter in Section 6.5.

6.1 Division of Binary Trees with High Locality

To develop efficient parallel programs on distributed-memory parallel computers, we need to divide data structures into smaller parts to distribute them to the processors. Here, the division of data structures should have the following two properties for efficiency of parallel programs.

- *Locality.* The data distributed to each processor should be adjacent. If two elements, which are adjacent in the original data, are distributed to different processors, then we often need communications between the processors.

- *Load balance.* The number of nodes distributed to each processor should be nearly equal since the cost of local computation is often proportional to the number of nodes.

It is easy to divide a list with these two properties, that is, for a given list of N elements we simply divide the list into P sublists with N/P elements for each sublist. It is, however, difficult to divide a tree satisfying both of the two properties. The non linear and ill-balanced structure of binary trees makes it difficult to divide the tree into connected components with good load balance.

In this section, we introduce a division of binary trees based on the basic graph theory, and show how to represent the distributed tree structures for efficient implementation of tree skeletons.

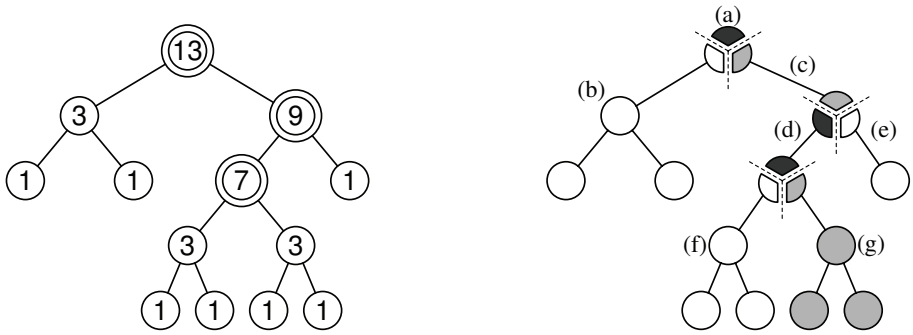


Figure 6.1. An example of m-critical nodes and m-bridges. Left: In this binary tree, there are three 4-critical nodes denoted by the doubly-lined circles. The number in each node denotes the number of nodes in the subtree. Right: For the same tree there are seven 4-bridges, (a)–(g), each of which is a set of connected nodes.

6.1.1 Graph-Theoretic Results for Division of Binary Trees

We start the discussion by introducing some graph-theoretic results [45,112]. Let $size_{\mathrm{b}}(v)$ denote the number of nodes in the subtree rooted at node v.

Definition 6.1 (m-Critical Node [45, 112]) Let m be an integer such that $1 < m \leq N$ where N is the number of nodes in a binary tree. A node v is called *m-critical*, if

- v is an internal node, and
- for each child v' of v inequality $\lceil size_{\mathrm{b}}(v)/m \rceil > \lceil size_{\mathrm{b}}(v')/m \rceil$ holds. □

The m-critical nodes divide a tree into sets of adjacent nodes (m-bridges) as shown in Figure 6.1. Note that the global structure given by m-bridges also forms a binary tree.

Definition 6.2 (m-Bridge [45, 112]) Let m be an integer such that $1 < m \leq N$ where N is the number of nodes in a binary tree. An *m-bridge* is a set of adjacent nodes divided by m-critical nodes, that is, a largest set of adjacent nodes in which m-critical nodes are only at the root or bottom. □

The m-critical nodes and the m-brides have several important properties in dividing binary trees.

The following two lemmas show properties of the m-critical nodes and the m-bridges in terms of the global shape of them.

Lemma 6.1 ([45, 112]) *If v_1 and v_2 are m-critical nodes then their least common ancestor is also an m-critical node.* □

Lemma 6.2 ([45, 112]) *If B is an m-bridge of a tree then B has at most one m-critical node among the leaves of it.* □

The root node in each m-bridge, except for the root m-bridge that includes the global root node, is an m-critical node. If we remove the root m-critical node if it exists, from Lemma 6.2 and the definition of the m-bridge, the m-bridge has at most one m-critical node. From Lemma 6.1, an m-critical node is an segment appearing in dividing a binary tree and the leaf m-critical node corresponds to the terminal node.

The following three lemmas are related to the number of nodes in an m-bridge and the number of m-bridges in a tree. Note that the first two lemmas hold on general trees while the last lemma only holds on binary trees.

Lemma 6.3 ([45, 112]) *The number of nodes in an m-bridge is at most $m + 1$.* □

Lemma 6.4 ([45, 112]) *Let N be the number of nodes in a tree then the number of m-critical nodes in the tree is at most $2N/m - 1$.* □

Lemma 6.5 *Let N be the number of nodes in a binary tree then the number of m-critical nodes in the binary tree is at least $(N/m - 1)/2$.*

Proof. Let n_k be the number of nodes in binary trees that have k m-critical nodes. We prove this lemma by proving the following inequality by induction on k.

$$n_k \leq (2k + 1)m \tag{6.1}$$

1. Base case:

By definition of m-critical nodes, for the root node v we have $\lceil size(v)/m \rceil = 1$. Therefore, we obtain $0 < size(v) \leq m$, which satisfies the inequality (6.1) for the case $k = 0$.

2. Inductive step:

Assume that for all i less that k inequality $n_i \leq (2i+1)m$ holds. Let v be the m-critical node nearest to the root node. Since the least common ancestor of two m-critical nodes is also m-critical node as Lemma 6.1 says, we can find such an m-critical node for any binary tree. Now we consider the following three parts of a tree: the left subtree of the node v, which has k_1 terminal nodes, the right subtree of the node v, which has k_2 terminal nodes, and the other parts, which has no terminal node. By definition $1 + k_1 + k_2 = k$ holds.

Let x_1, x_2, and x_3 be the numbers of nodes of the first, second, and third parts, respectively. Then, by hypothesis we obtain $x_1 \leq (2k_1 + 1)m$ and $x_2 \leq (2k_2 + 1)m$. The number of nodes in the third part is at most m, i.e., $x_3 \leq m$, where the equality holds if for the node v and the root r of the binary tree, equations $size(v) = am + 1$ and $size(r) = (a + 1)m$ holds for some value a.

With these inequalities, we can prove the inequality (6.1) by the following calculation.

$$\begin{aligned} n_k &= x_1 + x_2 + x_3 \\ &\leq (2k_1 + 1)m + (2k_2 + 1)m + m \\ &= (2(k_1 + k_2 + 1) + 1)m \\ &= (2k + 1)m \end{aligned}$$

It follows from the transformation of inequality (6.1) that the lemma holds. □

Let N be the number of nodes and P be the number of processors. In the previous studies [45, 87, 112], we divided a tree into m-bridges using the parameter m given by $m = 2N/P$. Under this division we obtain at most $(2P - 1)$ m-bridges and thus each processor deals with at most two m-bridges. Of course this division enjoys high locality, but it has poor load balance since the maximum number of nodes passed to a processor may be $2N/P$, which is twice of the number of nodes for the best lead-balancing case.

In Section 6.3, we adjust the value m for better division of binary trees based on the cost model of the skeletons. The idea is to divide a binary tree into more m-bridges using smaller m so that we obtain enough load balance while keeping the overheads caused by loss of locality rather small.

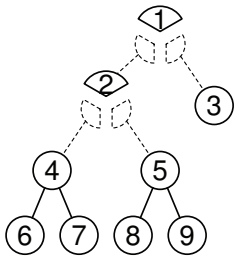
6.1.2 Data Structure for Distributed Segments

The performance of the sequential computation parts is as important as that of the communication parts for efficient parallel programs. This means that the data structure of local segments is important.

Generally speaking, tree structures are often implemented using pointers or references. There are, however, two problems in this implementation for large-scale tree applications. First, a lot of memory is required. Considering trees of integers or trees of real numbers, for example, the pointers use as many memory as the value for each node. Furthermore, if we allocate nodes one by one, more memory are consumed for the information for freeing the nodes. Second, locality is often lost. Recent computers have a cache hierarchy to bridge the gap between the CPU speed and the memory speed, and cache misses greatly decrease the performance especially in data-intensive applications. If we allocate nodes from here and there then the probability of cache misses increases.

To resolve these problems, we represent a binary tree as an array serialized in the order of the prefix traversal. We represent a tree divided by the m-bridges using one array gt for the global structure and one array of arrays *segs* for the local segments. Note that the arrays in *segs* are distributed among processors and only one processor has the array for each local segment. Figure 6.2 illustrates the array representation of the distributed tree. Since adjoining elements are aligned one next to another in this representation, we can reduce cache misses.

We introduce some notations for the discussion of implementation algorithms in the next section. Some values may be attached to the global structure, and we write $gt[i]$ to access to the value attached on ith element of global structure. If ith segment in *segs* is distributed on pth processor, we denote $pr(i) = p$. For a given serialized array for a segment *seg*, we denote $seg[i]$ for the ith value in the serialized array, and use functions isLeaf($seg[i]$), isNode($seg[i]$) and isTerminal($seg[i]$) to check whether the ith node is a leaf, an internal node, and a terminal node, respectively.



$$gt = [\square^{\mathsf{N}}, \square^{\mathsf{N}}, \square^{\mathsf{L}}, \square^{\mathsf{L}}, \square^{\mathsf{L}}]$$

$$seg = [\,[\,1^{\mathsf{C}}\,],\ [\,2^{\mathsf{C}}\,],\ [\,4^{\mathsf{N}}, 6^{\mathsf{L}}, 7^{\mathsf{L}}\,],\ [\,5^{\mathsf{N}}, 8^{\mathsf{L}}, 9^{\mathsf{L}}\,],\ [\,3^{\mathsf{L}}\,]\,]$$

Figure 6.2. Array representation of divided binary trees. Each local segment of *segs* is distributed to one of processors and is not shared. Labels L, N and T denote a leaf, a normal internal node, and a terminal node, respectively. Each m-critical node is included in the parent segment.

6.2 Implementation and Cost Model of Tree Skeletons

In this section, we show the implementation and the cost model of the tree skeletons on distributed-memory parallel computers. We implement the local computations in tree skeletons using loops and stacks on the serialized arrays to reduce the cache misses. This is the most significant technique, with which the parallel programs achieve high performance in the sequential computation parts of the algorithm.

We introduce several parameters for discussion of the cost model (Table 6.1). The computational time of function f executed with p processors is denoted by $t_p(f)$. In particular, $t_1(f)$ denotes the cost of sequential computation of f. Parameter N denotes the number of nodes, and P denotes the number of processors. Parameter m is used for m-critical nodes and m-bridges, and M denotes the number of segments after the division. For the ith segment, in addition to the parameter of the number of nodes L_i, we introduce parameter D_i indicating the depth of the critical node. Parameter c_α denotes the communication time for a value of type α.

The cost model for tree accumulations can be uniformly given as the sum of the maximum cost of local computations and the cost of global computations as follows.

$$\max_p \sum_{pr(i)=p} (L_i \times t_l + D_i \times t_d) + M \times t_m$$

Table 6.1. Parameters for the cost model.

$t_p(f)$	computational time of function f using p processors
N	the number of nodes in the input tree
P	the number of processors
m	the parameter for m-critical nodes and m-bridges
M	the number of segments given by division of trees
L_i	the number of nodes in the ith segment
D_i	the depth of the terminal node in the ith segment
c_α	the time needed for communicating one data of type α

In the expression of the cost model, $\sum_{pr(i)=p}$ denotes the summation of cost for m-bridges associated to processor p, and t_l, t_d, t_m are given with the cost of functions and the cost of communications. The term $(L_i \times t_l)$ indicates the computational time required in sequential computation and the term $(D_i \times t_d)$ indicates the overhead caused by parallelism. The last term $(M \times t_m)$ indicates the overhead in terms of global structure.

6.2.1 Map and Zipwith Skeleton

Since there are no dependencies among nodes in the computation of the map_b skeleton, we can implement the map_b skeleton by applying the following function MAP_LOCAL to each local segment. The MAP_LOCAL function applies function k_l to each leaf and function k_n to each internal node and the terminal node in a local segment seg.

```
MAP_LOCAL(k_l, k_n, seg)
    for i ← 0 to seg.size − 1
        if (isLeaf(seg[i]))       seg'[i] ← k_l(seg[i]);
        if (isNode(seg[i]))       seg'[i] ← k_n(seg[i]);
        if (isTerminal(seg[i]))   seg'[i] ← k_n(seg[i]);
    return seg';
```

Using this function, we can implement the map_b skeleton as follows.

```
map_b(k_l, k_n, (gt, segs))
    for i ← 0 to gt.size − 1
        if (pr(i) == p) segs'[i] = MAP_LOCAL(k_l, k_n, segs[i])
    return (gt, segs')
```

In a local segment with L_i nodes, the number of leaves is at most $L_i/2 + 1$ and the number of internal nodes including the terminal node is at most $L_i/2 + 1$. Therefore, ignoring small constants we can specify the computational cost of the MAP_LOCAL function as follows.

$$t_1(\text{MAP\_LOCAL}) = \frac{L_i}{2} \times t_1(k_l) + \frac{L_i}{2} \times t_1(k_n)$$

Therefore, the cost model for the map_b skeleton is as follows.

$$t_P(\mathsf{map}_\mathrm{b}\ k_l\ k_n) = \max_p \sum_{pr(i)=p} L_i \times \frac{t_1(k_l) + t_1(k_n)}{2}$$

Since the $\mathsf{zipwith}_\mathrm{b}$ skeleton performs the similar computation as the map_b skeleton, we can give the implementation algorithm and the cost model for the $\mathsf{zipwith}_\mathrm{b}$ skeleton in the same manner.

6.2.2 Reduce Skeleton

We then show an implementation and the cost model of the $\mathsf{reduce_b}$ skeleton called with function k and auxiliary functions $k = \langle \phi, \psi_n, \psi_l, \psi_r \rangle_{\mathrm{u}}$. Let the type of $\mathsf{reduce_b}$ skeleton be $\mathsf{reduce_b} :: (\beta \to \alpha \to \alpha \to \alpha) \to \mathsf{BTree}\ \alpha\ \beta \to \alpha$ and the type of the intermediate value be γ (i.e., the function ϕ has type $\phi :: \beta \to \gamma$).

The implementation of the $\mathsf{reduce_b}$ skeleton consists of the following three steps:

1. local reduction for each segment,
2. gathering local results to the root processor, and
3. global reduction on the root processor.

Step 1. Local Reduction

The bottom-up computation of the $\mathsf{reduce_b}$ skeleton can be computed by reversed traversal on the array using a stack for the intermediate results. Firstly we apply REDUCE_LOCAL function to each local segment to reduce it to a value. In the the REDUCE_LOCAL function,

- we apply functions ϕ and either ψ_l or ψ_r to the terminal node and its ancestors, and
- we apply function k to the other internal nodes.

Here, applying the function k is cheaper than applying function ϕ and ψ_n, even though $k\ l\ n\ r = \psi_n\ l\ (\phi\ n)\ r$ holds with respect to the results of functions. To specify where the terminal node or its ancestor is in the stack, we use a variable d that indicates the position. Note that in the computation of the REDUCE_LOCAL function, at most one element in the stack has the value of the terminal node or its ancestors.

```
REDUCE_LOCAL(k, φ, ψ_l, ψ_r, seg)
    stack ← ∅; d ← −∞;
    for i ← seg.size − 1 to 0
        if (isLeaf(seg[i]))
            stack ← seg[i]; d ← d + 1;
        if (isNode(seg[i]))
            lv ← stack; rv ← stack;
            if (d == 0)        stack ← ψ_l(lv, φ(seg[i]), rv);
            else if (d == 1)   stack ← ψ_r(lv, φ(seg[i]), rv); d ← 0;
            else               stack ← k(lv, seg[i], rv); d ← d − 1;
        if (isTerminal(seg[i]))
            stack ← φ(seg[i]); d ← 0;
    top ← stack; return top;
```

In this step, functions ϕ and either ψ_l or ψ_r is applied to the terminal node and its ancestors (D_i nodes) and function k is applied to the other internal nodes ($(M_i/2 - D_i)$ nodes). Thus, the cost of REDUCE_LOCAL is given as follows.

$$
\begin{aligned}
t_1(\text{REDUCE\_LOCAL}) &= D_i \times (t_1(\phi) + \max(t_1(\psi_l), t_1(\psi_r))) + \left(\frac{L_i}{2} - D_i\right) \times t_1(k) \\
&= \frac{L_i}{2} \times t_1(k) + D_i \times (t_1(\phi) + \max(t_1(\psi_l), t_1(\psi_r)) - t_1(k))
\end{aligned}
$$

Step 2. Gathering Local Results to Root Processor

In the second step, we gather all the local results to the root processor. The communication cost is given by the number of leaf segments of type α and the number of internal segments of type γ.

$$t_P(\text{Step 2}) = \frac{M}{2} \times c_\alpha + \frac{M}{2} \times c_\gamma$$

Let the gathered values be put in array gt after this step.

Step 3. Global Reduction on Root Processor

Finally, we compute the result of the $\mathsf{reduce}_\mathrm{b}$ skeleton by applying the REDUCE_GLOBAL function below to the array of local results. This computation is performed on the root processor. We compute the result by applying ψ_n for each internal node in a bottom-up manner, which is implemented by a reversed traversal with a stack on the array for the global structure.

```
REDUCE_GLOBAL(ψ_n, gt)
    stack ← ∅;
    for i ← gt.size − 1 to 0
        if (isLeaf(gt[i]))
            stack ← gt[i];
        if (isNode(gt[i]))
            lv ← stack; rv ← stack; stack ← ψ_n(lv, gt[i], rv)
    top ← stack; return top;
```

In this step the function ψ_n is applied to each internal segment and thus the cost of REDUCE_GLOBAL is given as follows.

$$t_1(\text{REDUCE\_GLOBAL}) = \frac{M}{2} \times t_1(\psi_n)$$

In summary, we obtain the following program for the $\mathsf{reduce}_\mathrm{b}$ skeleton.

```
reduce_b(k, ⟨φ, ψ_n, ψ_l, ψ_r⟩_u, (gt, segs))
    for i ← 0 to gt.size − 1
        if (pr(i) == p) gt[i] = REDUCE_LOCAL(k, φ, ψ_l, ψ_r, segs[i])
    gather_to_root(gt)
    if (p == 0) REDUCE_GLOBAL(ψ_n, gt)
```

The cost model of the $\mathsf{reduce}_\mathrm{b}$ skeleton is given as follows.

$$\begin{aligned}
&t_P(\mathsf{reduce}_\mathrm{b}\ k) \\
&\quad = \max_p \sum_{pr(i)=p} t_1(\text{REDUCE_LOCAL}) + t_P(\text{Step 2}) + t_1(\text{REDUCE_GLOBAL}) \\
&\quad = \max_p \sum_{pr(i)=p} \left(L_i \times \frac{t_1(k)}{2} + D_i \times (-t_1(k) + t_1(\phi) + \max(t_1(\psi_l), t_1(\psi_r))) \right) \\
&\qquad\quad + M \times \frac{c_\alpha + c_\gamma + t_1(\psi_n)}{2}
\end{aligned}$$

6.2.3 Upwards Accumulate Skeleton

Next, we develop an implementation of the $\mathsf{uAcc_b}$ skeleton called with function k and auxiliary functions $k = \langle \phi, \psi_n, \psi_l, \psi_r \rangle_{\mathrm{u}}$. Similar to the $\mathsf{reduce_b}$ skeleton, let the type of the $\mathsf{uAcc_b}$ skeleton be $\mathsf{uAcc_b} :: (\beta \to \alpha \to \alpha \to \alpha) \to \mathsf{BTree}\ \alpha\ \beta \to \mathsf{BTree}\ \alpha\ \alpha$, and the type of intermediate value be γ.

The implementation of the $\mathsf{uAcc_b}$ skeleton on distributed trees consists of the following five steps:

1. local upwards accumulation and reduction for each segment,
2. gathering results of local reductions to the root processor,
3. global upwards accumulation on the root processor,
4. distributing result of global upwards accumulation, and
5. local update for each internal segment.

Step 1. Local Upwards Accumulation

In the first step, we apply the following function UACC_LOCAL to each segment and compute local upwards accumulation and reduction. This function puts the intermediate results to array seg' if a node has no terminal node as descendants. This result value is indeed the result of the uAcc skeleton. This function puts nothing to array seg' if a node is either the terminal node or an ancestor of the terminal node. Returned values are the result of local reduction and the array seg'.

```
UACC_LOCAL(k, φ, ψ_l, ψ_r, seg)
    stack ← ∅; d ← −∞;
    for i ← seg.size − 1 to 0
        if (isLeaf(seg[i]))
            seg'[i] ← seg[i]; stack ← seg'[i]; d ← d + 1;
        if (isNode(seg[i]))
            lv ← stack; rv ← stack;
            if (d == 0)        stack ← ψ_l(lv, φ(seg[i]), rv); d ← 0;
            else if (d == 1)   stack ← ψ_r(lv, φ(seg[i]), rv); d ← 0;
            else               seg'[i] ← k(lv, seg[i], rv); stack ← seg'[i]; d ← d − 1;
        if (isTerminal(seg[i]))
            stack ← φ(seg[i]); d ← 0;
    top ← stack; return(top, seg');
```

In the computation of the UACC_LOCAL function, ϕ and either of ψ_l or ψ_r are applied to each node of the terminal node and its ancestors (D_i nodes), and k is applied to the other internal nodes ($(L_i/2 - D_i)$ nodes). We therefore obtain the cost of the UACC_LOCAL function as follows. Note that this cost is the same as that of REDUCE_LOCAL function.

$$
\begin{aligned}
t_1(\text{UACC\_LOCAL}) &= D_i \times (t_1(\phi) + \max(t_1(\psi_l), t_1(\psi_r))) + \left(\frac{L_i}{2} - D_i\right) \times t_1(k) \\
&= \left(\frac{L_i}{2}\right) \times t_1(k) + D_i \times (t_1(\phi) + \max(t_1(\psi_l), t_1(\psi_r)) - t_1(k))
\end{aligned}
$$

Step 2. Gathering Results of Local Reductions to Root Processor

In the second step, we gather the results of the local reductions on the global structure gt of the root processor. From each leaf segment a value of type α is transferred, and from each internal segment a value of type γ is transferred. Since the number of leaf segments and the number of internal segments are almost $M/2$ respectively, the communication cost of the second step is given as follows.

$$t_P(\text{Step 2}) = \frac{M}{2} \times c_\alpha + \frac{M}{2} \times c_\gamma$$

Step 3. Global Upward Accumulation on Root Processor

In the third step, we compute the upwards accumulation for the global structure gt on the root processor. Function uAcc_global performs sequential upwards accumulation using function ψ_n.

```
uAcc_global(ψ_n, gt)
    stack ← ∅;
    for i ← gt.size − 1 to 0
        if (isLeaf(gt[i]))
            gt'[i] ← gt[i];
        if (isNode(gt[i]))
            lv ← stack; rv ← stack; gt'[i] ← ψ_n(lv, gt[i], rv);
        stack ← gt'[i];
    return(gt');
```

In this function, we apply function ψ_n to each internal segment of gt, and thus the cost of the third step is given as

$$t_1(\text{uAcc\_global}) = \frac{M}{2} \times t_1(\psi_n) \ .$$

Step 4. Distributing Global Result

In the fourth step, we send the result of global upwards accumulation to processors, where two values are sent to each internal segment and no values are sent to each leaf segment. Since all the values have type α after the global upwards accumulation, and the communication cost of the fourth step is given as follows.

$$t_P(\text{Step 4}) = M \times c_\alpha$$

Step 5. Local Updates for Each Internal Segment

In the last step, we apply function uAcc_update to each internal segment. At the beginning of the function, the two values pushed in the previous step are pushed to the stack. These two values correspond to the results of children of the terminal node. Note

that in the last step we only compute the missing values in the segment seg', which is given in the local upwards accumulation (Step 1).

```
uAcc_update(k, seg, seg', (lc, rc))
    stack ← ∅; stack ← rc; stack ← lc;
    d ← −∞;
    for i ← seg.size − 1 to 0
        if (isLeaf(seg[i]))
            stack ← seg'[i]; d ← d + 1;
        if (isNode(seg[i]))
            lv ← stack; rv ← stack;
            if (d == 0)        seg'[i] ← k(lv, seg[i], rv); stack ← seg'[i];
            else if (d == 1) seg'[i] ← k(lv, seg[i], rv); stack ← seg'[i]; d ← 0;
            else               stack ← seg'[i]; d ← d − 1;
        if (isTerminal(seg[i]))
            lv ← stack; rv ← stack;
            seg'[i] ← k(lv, seg[i], rv); stack ← seg'[i]; d ← 0;
    return(seg');
```

In this step, function k is applied to the nodes on the path from the terminal node to the root node for each internal segment. Noting that the depth of the terminal nodes is D_i, we can give the cost of uAcc_update as follows.

$$t_1(\text{uAcc\_update}) = D_i \times t_1(k)$$

Using the functions defined so far, we can implement the $\mathsf{uAcc_b}$ skeleton as follows.

```
uAcc_b(k, ⟨φ, ψ_n, ψ_l, ψ_r⟩_u, (gt, segs))
    for i ← 0 to gt.size − 1
        if (pr(i) == p) (gt[i], segs'[i]) = uAcc_local(k, φ, ψ_l, ψ_r, segs[i])
    gather_to_root(gt)
    if (p == 0) gt' = uAcc_global(ψ_n, gt)
    distribute_from_root(gt')
    for i ← 0 to gt.size − 1
        if (pr(i) == p ∧ isNode(gt'[i]))
            segs'[i] = uAcc_update(k, segs[i], segs'[i], gt'[i])
    return (gt', segs')
```

The cost model of the $\mathsf{uAcc_b}$ skeleton is given as follows.

$$\begin{aligned}
&t_P(\mathsf{uAcc_b}) \\
&\quad = \max_p \sum_{pr(i)=p} t_1(\text{uAcc_local}) + t_P(\text{Step 2}) + t_1(\text{uAcc_global}) \\
&\qquad\quad + t_P(\text{Step 4}) + \max_p \sum_{pr(i)=p} t_1(\text{uAcc_update}) \\
&\quad = \max_p \sum_{pr(i)=p} \left(L_i \times \frac{t_1(k)}{2} + D_i \times (t_1(\phi) + \max(t_1(\psi_l), t_1(\psi_r))) \right) \\
&\qquad\quad + M \times (3c_\alpha + c_\gamma + t_1(\psi_n))/2
\end{aligned}$$

6.2.4 Downwards Accumulate Skeleton

Finally, we develop an implementation and the cost model for the $\mathsf{dAcc_b}$ skeleton called with a pair of functions (g_l, g_r) and auxiliary functions $(g_l, g_r) = \langle \phi_l, \phi_r, \psi_u, \psi_d \rangle_{\mathrm{d}}$. Let the type of the skeleton be $\mathsf{dAcc_b} :: (\gamma \to \beta \to \gamma, \gamma \to \beta \to \gamma) \to \gamma \to \mathsf{BTree}\ \alpha\ \beta \to \mathsf{BTree}\ \gamma\ \gamma$ and the type of the intermediate value be δ (i.e., the function ϕ_l has type $\phi_l :: \beta \to \delta$, for example.).

The implementation of the $\mathsf{dAcc_b}$ skeleton also consists of the following five steps:

1. computing two intermediate values for each internal segment,
2. gathering local results to the root processor,
3. global downwards accumulation on the root processor,
4. distributing the result of global downwards accumulation, and
5. local downwards accumulation for each segment.

Step 1. Computing Local Intermediate Values

In the first step, we compute for each internal segment two local intermediate values, which are used in updating the accumulative parameter from the root node to the both children of the terminal node. To minimize the computation cost, we first find the terminal node and then compute two values only on the path from the root node to the terminal node. We implement this computation by the following function DACC_PATH, in which the computation is done by a reversed traversal on the array with an integer d instead of a stack. Two variables toL and toR are the intermediate values.

```
DACC_PATH(φ_l, φ_r, ψ_u, seg)
  d ← −∞;
  for i ← seg.size − 1 to 0
    if (isLeaf(seg[i]))
      d ← d + 1;
    if (isNode(seg[i]))
      if (d == 0)
        toL = ψ_u(φ_l(seg[i]), toL); toR = ψ_u(φ_l(seg[i]), toR);
      else if (d == 1)
        toL = ψ_u(φ_r(seg[i]), toL); toR = ψ_u(φ_r(seg[i]), toR);
        d ← 0;
      else
        d ← d − 1;
    if (isTerminal(seg[i]))
      toL ← φ_l(seg[i]); toR ← φ_r(seg[i]);
      d ← 0;
  return (toL, toR);
```

In this step we apply ψ_u twice and either ϕ_l or ϕ_r for each of the ancestors of the terminal nodes (D_i nodes). Therefore, omitting some small constants, the cost of the DACC_PATH function is given as follows.

$$t_1(\text{DACC\_PATH}) = D_i \times (\max(t_1(\phi_l), t_1(\phi_r)) + 2t_1(\psi_u))$$

Step 2. Gathering Local Results to Root Processor

In the second step, we gather the local results of the internal segments to the root processor. Since the two intermediate values have type δ and the number of internal segments is $M/2$, the communication cost in the second step is given as follows.

$$t_P(\text{Step 2}) = M \times c_\delta$$

The pair of local results from each internal segment is put to the array of the global tree structure gt.

Step 3. Global Downwards Accumulation

In the third step, we compute global downwards accumulation on the root processor. We implement this global downwards accumulation with a forward traversal using a stack as shown in the following function DACC_GLOBAL. Firstly, the initial value of accumulative parameter is pushed to the stack, and then the accumulative parameter in the stack is updated with the pair of local results given in the previous step. The result of global accumulation is the accumulative parameter passed to the root node for each segment.

```
DACC_GLOBAL(ψ_d, c, gt)
    stack ← ∅; stack ← c;
    for i ← 0 to gt.size − 1
        if (isLeaf(gt[i]))
            gt'[i] ← stack;
        if (isNode(gt[i]))
            gt'[i] ← stack; (toL, toR) ← gt[i];
            stack ← ψ_d(gt'[i], toR); stack ← ψ_d(gt'[i], toL);
    return gt';
```

The DACC_GLOBAL function applies function ψ_d twice for each internal segment in the global structure. Therefore, the computational cost of the DACC_GLOBAL function is given as follows.

$$t_1(\text{DACC\_GLOBAL}) = M \times t_1(\psi_d)$$

Step 4. Distributing Global Result

In the fourth step, we distribute the result of global downwards accumulation to the corresponding processor. Since each result of global downwards accumulation has type γ, the communication cost of the fourth step is given as follows.

$$t_P(\text{step 4}) = M \times c_\gamma$$

Step 5. Local Downwards Accumulation

Finally, we compute local downwards accumulation for each segment. The initial value c' of the accumulative parameter is given in the previous step. Note that the definition of the following dAcc_local function is just the same as the sequential version of the downwards accumulation on the serialized array if we assume the terminal node as a leaf.

```
dAcc_local(g_l, g_r, c', seg)
    stack ← ∅; stack ← c';
    for i ← 0 to seg.size − 1
        if (isLeaf(seg[i]))
            seg'[i] ← stack;
        if (isNode(seg[i]))
            seg'[i] ← stack; stack ← g_r(seg'[i], seg[i]); stack ← g_l(seg'[i], seg[i]);
        if (isTerminal(seg[i]))
            seg'[i] ← stack;
    return seg';
```

The local downwards accumulation applies functions g_l and g_r for each internal node. Since the number of the internal nodes is almost $L_i/2$, the computational cost of the dAcc_local function is given as follows.

$$t_1(\text{dAcc\_local}) = \frac{L_i}{2} \times (t_1(g_l) + t_1(g_r))$$

Summarizing the discussion so far, we obtain the following implementation of the $\mathsf{dAcc_b}$ skeleton.

```
dAcc_b((g_l, g_r), ⟨φ_l, φ_r, ψ_u, ψ_d⟩_d, c, (gt, segs))
    for i ← 0 to gt.size − 1
        if (pr(i) == p ∧ isNode(gt[i]))
            gt[i] = dAcc_path(φ_l, φ_r, ψ_u, segs[i])
    gather_to_root(gt)
    if (p == 0) gt' = dAcc_global(ψ_d, c, gt)
    distribute_from_root(gt')
    for i ← 0 to gt.size − 1
        if (pr(i) == p) segs'[i] = dAcc_local(g_l, g_r, gt'[i], segs[i])
    return (gt', segs')
```

The cost model of the $\mathsf{dAcc_b}$ skeleton is given as follows.

$$\begin{aligned}
&t_P(\mathsf{dAcc_b}) \\
&\quad = \max_p \sum_{pr(i)=p} t_1(\text{dAcc_path}) + t_P(\text{Step 2}) + t_1(\text{dAcc_global}) \\
&\qquad + t_P(\text{Step 4}) + \max_p \sum_{pr(i)=p} t_1(\text{dAcc_local}) \\
&\quad = \max_p \sum_{pr(i)=p} \left(L_i \times \frac{t_1(g_l) + t_1(g_r)}{2} + D_i \times (\max(t_1(\phi_l), t_1(\phi_r)) + 2t_1(\psi_u)) \right) \\
&\qquad + M \times (c_\delta + t_1(\psi_d) + c_\gamma)
\end{aligned}$$

6.3 Optimal Division of Binary Trees Based on Cost Model

As we stated at the beginning of Section 6.1, locality and load balance are two important properties in developing efficient parallel programs in particular on distributed-memory parallel computers. When we divide and distribute a binary tree using the m-bridges, we enjoy good locality with large m while we enjoy good load balance with small m. Therefore, we need to find an appropriate value for m to achieve both good locality and good load balance.

First, we show the relations among parameters of the cost model. From Lemma 6.3 and the representation of local segments in Figure 6.2, we have

$$L_i \leq m \ . \tag{6.2}$$

Since the height of a tree is at least a half of the number of nodes, we obtain

$$D_i \leq L_i/2 \leq m/2 \ . \tag{6.3}$$

From Lemmas 6.4 and 6.5, the number of local segments M is bound with the number N of nodes and the parameter m as follows.

$$\frac{1}{2}\left(\frac{N}{m} - 1\right) \leq M \leq \frac{2N}{m} - 1 \tag{6.4}$$

By inequality (6.3), the general form of the cost model can be transformed into the following simpler form.

$$\begin{aligned}
&\max_p \sum_{pr(i)=p} (L_i \times t_l + D_i \times t_d) + M \times t_m \\
&\quad \leq \max_p \sum_{pr(i)=p} \left(L_i \times t_l + \frac{L_i}{2} \times t_d\right) + M \times t_m \\
&\quad = (\max_p \sum_{pr(i)=p} L_i) \times \left(t_l + \frac{t_d}{2}\right) + M \times t_m
\end{aligned}$$

Next, we want to bound the maximum number of nodes distributed to a processor, $\max_p \sum_{pr(i)=p} L_i$,. We distribute the local segment to processors so as to obtain good load balance. One easy way to implement the load balancing is greedy distribution of the local segments from the largest one. By this greedy distribution, the difference between the maximum number of nodes $\max_p \sum_{pr(i)=p} L_i$ and the minimum number of nodes $\min_p \sum_{pr(i)=p} L_i$ is less than or equal to the maximum number of nodes in a segment. Since the maximum number of nodes in a local segment is m as stated in inequality (6.2) and the total number of nodes in the original binary tree is N, we can bound the maximum number of nodes distributed to a processor as follows:

$$\max_p \sum_{pr(i)=p} L_i \leq \frac{N}{P} + m$$

where P denotes the number of processors. By substituting this inequality to the cost model, we can bound the cost of the worst case.

$$\max_p \sum_{pr(i)=p} (L_i \times t_l + D_i \times t_d) + M \times t_m \leq \left(\frac{N}{P} + m\right) \times \left(t_l + \frac{t_d}{2}\right) + M \times t_m \quad (6.5)$$

Now we want to minimize the worst-case cost given in the right-hand side of inequality (6.5). By substituting the parameter M (inequality (6.4)), the worst-case cost is bound with respect to m. We can bound the worst-case cost for smaller m as

$$\left(\frac{N}{P} + m\right) \times \left(t_l + \frac{t_d}{2}\right) + M \times t_m \leq \left(\frac{N}{P} + m\right) \times \left(t_l + \frac{t_d}{2}\right) + \frac{1}{2}\left(\frac{N}{m} - 1\right) \times t_m \ ,$$

and we can bound the worst-case cost for larger m as

$$\left(\frac{N}{P} + m\right) \times \left(t_l + \frac{t_d}{2}\right) + M \times t_m \leq \left(\frac{N}{P} + m\right) \times \left(t_l + \frac{t_d}{2}\right) + \frac{2N}{m} - 1 \times t_m \ .$$

From these bounds, we can minimize the worst-case cost for some value m in the following range.

$$\sqrt{\frac{t_m}{2t_l + t_d}}\sqrt{N} \leq m \leq 2\sqrt{\frac{t_m}{2t_l + t_d}}\sqrt{N}$$

This new range of the parameter m is much smaller than that used in the previous studies [45, 87, 112]. In Section 6.4, we will show several experiment results that support this new range.

6.4 Experiment Results

To confirm the efficiency of the implementation algorithm for binary-tree skeletons, we implemented binary tree skeletons in C++ and MPI and made several experiments. We used our PC-cluster of uniform PCs with Pentium 4 2.8-GHz CPU and 2-GByte memory connected with Gigabit Ethernet. The compiler and MPI library used are gcc 4.1.1 and MPICH 1.2.7, respectively.

We used the skeletal parallel program that solves the party planning problem. For the skeletal program on our PC-cluster, the parameters of the cost model are given as $t_l = 0.18$ μs, $t_d = 0.25$ μs, and $t_m = 100$ μs, which are measured with a small input. The input trees are (1) a balanced tree, (2) a randomly generated tree and (3) a fully ill-balanced tree, each with 16,777,215 ($= 2^{24} - 1$) nodes.

Figures 6.3 and 6.4 shows the general performance of the tree skeletons. Each execution time excludes the initial data distribution and final gathering. The speedups are plotted against the efficient sequential implementation of the program, which is implemented on the array representing binary trees based on the same algorithm. As seen in these plots, the implementation shows good scalability even against the efficient sequential program. By the m-bridges, the balanced tree is divided into leaf segments of the same size and internal

segments consisting only of one node. Therefore, the overhead caused by parallelism is very small for the balanced binary tree, and the implementation achieves almost linear speedups against the sequential program. For the random tree, the average depth of the terminal nodes is so small that the implementation achieves good performance close to that for the balance tree. The fully ill-balanced tree, however, is divided into leaf segments consisting of one node and internal segments with their terminal node at the depth $D_i = L_i/2$. From the cost model and its parameters, the skeletal parallel program has overheads caused by the factor of depth of the terminal nodes, $D_i \times t_d \approx 0.7 \times L_i \times t_l$. In fact, the experimental results show that the skeletal parallel program runs about two times slower for the fully ill-balanced tree than for the other two inputs.

To analyze the cost model and the range of the parameter m more in detail, we made more experiments for the randomly generated tree by changing the value of the parameter m. By substituting the parameters measured by a small input, we can expect that the parameter m in the range $50{,}000 < m < 100{,}000$ supports good performance. Figure 6.5 plots the execution times to the number of processors for three values of the parameter m. As we can see from this figure, the implementation achieves good performance for a wide range of m. Figure 6.6 plots the execution times to the parameter m. This figure shows that the performance gets worse if the parameter m is too small ($m < 4{,}000$) or too large ($m > 150{,}000$). For the parameter m in the range above the skeletal program achieves near the best performance, and we can conclude that the cost model and the estimation of the parameter m is useful for efficient implementations.

6.5 Short Summary

In this chapter, we have developed an efficient implementation of binary-tree skeletons for the distributed-memory parallel computers and its cost model. In our implementation, binary trees are divided into segments by m-bridges where the parameter m is estimated based on the cost model. The implementation mainly consists of loops with stacks traversing on the array representation of segments and this greatly makes the sequential computation parts efficient.

We used the parameters of the cost model measured manually for the program in this chapter. Some profiling systems that help estimating the parameters would be useful for the practical use of the implementation.

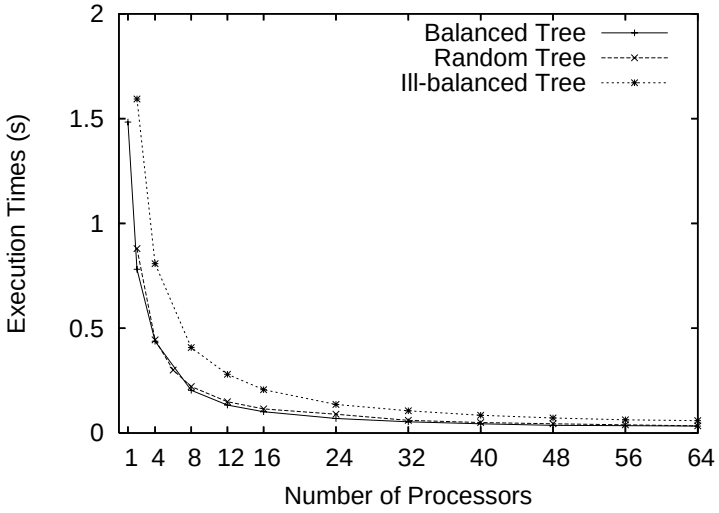


Figure 6.3. Execution times plotted to the number of processors. The parameter m for the division of trees is $m = 2 \times 10^4$.

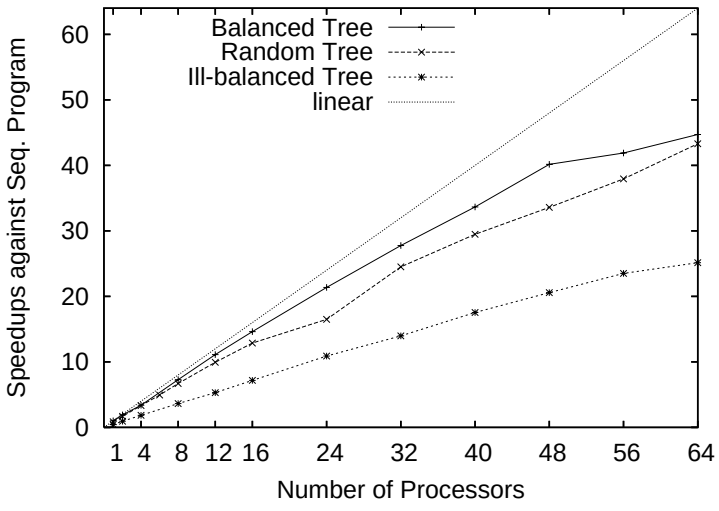


Figure 6.4. Speedups against sequential program plotted to the number of processors. The parameter m for the division of trees is $m = 2 \times 10^4$.

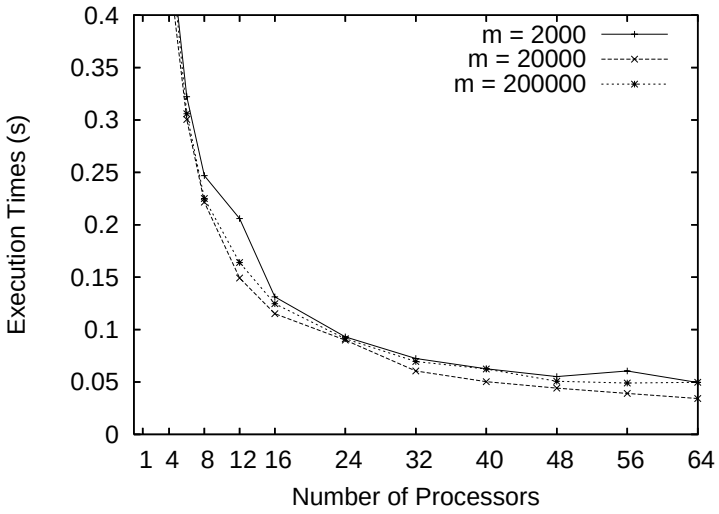


Figure 6.5. Execution times plotted to the number of processors. The input trees are from the same randomly generated tree divided with different parameter m.

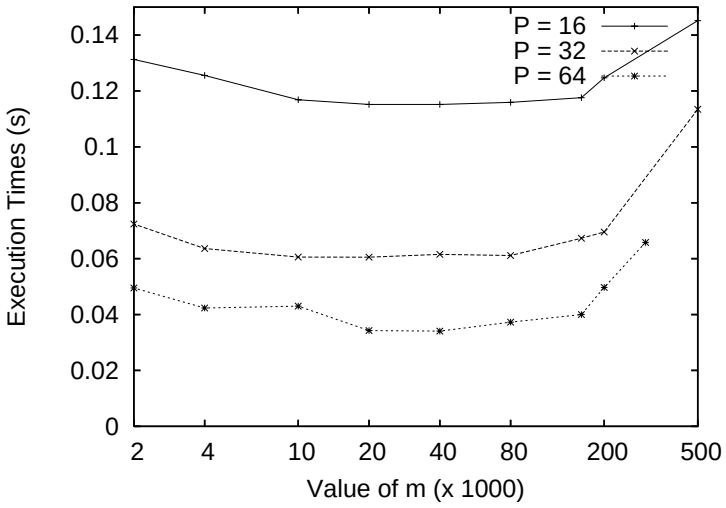


Figure 6.6. Execution times plotted to the parameter m for the cases with 16 processors, 32 processors, and 64 processors. The input trees are generated from the same randomly generated tree.

Chapter 7

SkeTo: Parallel Skeleton Library

We have implemented a parallel skeleton library named SkeTo (SkeTo is an abbreviation for "**Ske**leton Library in **To**kyo", and means "supporter" in Japanese.) to bring the theories studied so far to practice. Programmers can write skeletal parallel programs in a sequential programming style on the SkeTo library. The SkeTo library is now available online (`http://www.ipl.t.u-tokyo.ac.jp/sketo/`).

Figure 7.1 depicts the framework of the SkeTo library. The SkeTo library consists of two parts: the parallel skeleton library and the optimization mechanism. The parallel skeleton library is implemented in C++ and MPI; the optimization mechanism is implemented in OpenC++ [28].

Three major features of the SkeTo library are as follows.

Supporting Three Data Structures Based on Constructive Algorithmics

So far many parallel skeleton libraries have been implemented by many research groups [4, 7, 11, 25, 26, 75]. Many of them supports parallel manipulation of lists (one-dimensional arrays) and some of them supports parallel manipulation of matrices (two-dimensional arrays) or higher-dimensional arrays.

The SkeTo library provides parallel skeletons for lists, matrices, and trees. These parallel skeletons are formalized based on the solid theory of constructive algorithmics. For list skeletons, there have been several studies [65, 117, 118], and for matrix skeletons the foundation was given by Emoto et al. [42]. For tree skeletons, we have studies in Chapters 2–4. The uniform interface of the these parallel skeletons based on the constructive algorithmics helps users to use them.

Interface and Implementation with Standard C++ and MPI

The theories of parallel skeletons are formalized based on the ideas in functional programming such as polymorphism, higher-order functions. At the early stages of parallel

Overall of this chapter is based on [92] among which Section 7.1.3 is based on [89], Section 7.2 is based on [93, 132], and Section 7.3 is based on [91].

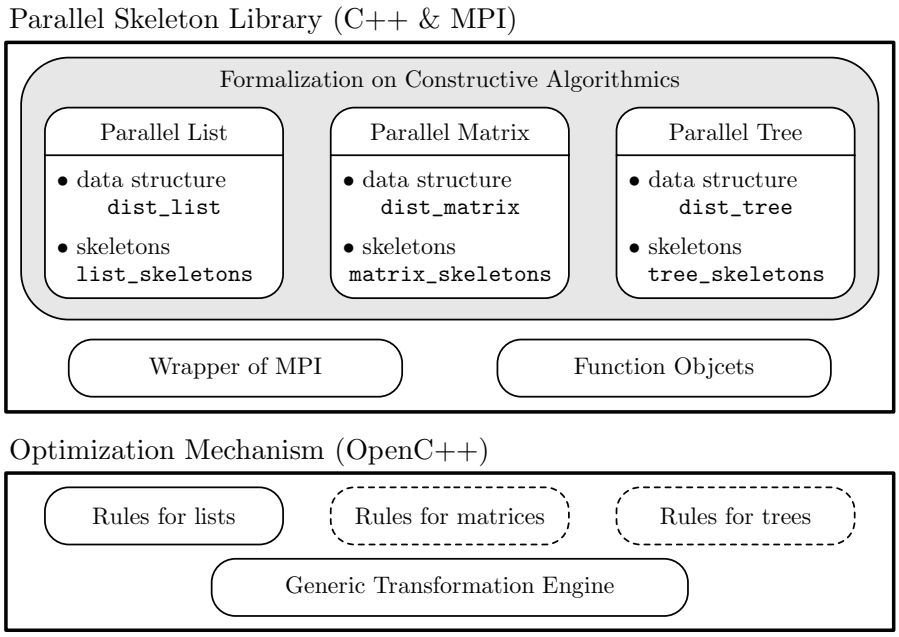


Figure 7.1. Overview of the SkeTo Library

skeleton libraries, there were several implementation of parallel skeletons in imperative languages with some extensions to support functional features [25, 114].

We have implemented the parallel skeletons with standard C++ and MPI without any language extensions borrowing the ideas from *Muesli* [75]. Since the C++ compiler and MPI library are now widely available, users can use the SkeTo library on many parallel computers. The SkeTo library also provides several wrapper functions of MPI library to encourage users to write their parallel programs as if they wrote sequential ones.

Optimization Mechanism

Skeletal parallel programs may show poorer performance than hand-coded parallel programs. This is mainly due to overhead caused by the limited algorithms manipulatable by parallel skeletons and overhead caused by the intermediate data structures between parallel skeletons. The second overhead can be removed by optimizing skeletal parallel programs while the first overhead is inevitable. Here, fusion transformations work well.

We have implemented the optimization mechanism in a meta-language OpenC++. This optimization mechanism automatically removes the intermediate data structures by fusing two successive skeletons.

The organization of this chapter is as follows. In Section 7.1, we discuss several implementation issues of the skeleton library. In Section 7.2, we show the optimization mechanism. In Section 7.3, we propose a code generator for auxiliary functions that supports developing skeletal parallel programs. In Section 7.4, we show experiment results on the SkeTo library for several examples. In Section 7.5, we summarize this chapter.

7.1 Coding Techniques for Efficiency and Programmability

With the SkeTo library users can develop parallel programs in C++ without considering data allocation and processor communication. Figure 7.2 shows a skeletal program using the SkeTo library corresponding to the following program that computes the height of tree.

$$
\begin{aligned}
\mathit{height}_{\mathrm{b}}\ \mathit{tree1} = {} & \textbf{let}\ \mathit{tree2} = \mathsf{dAcc}_{\mathrm{b}}\ (\lambda c\ b.c + 1, \lambda c\ b.c + 1)\ 1\ \mathit{tree1} \\
& \textbf{in}\ \mathsf{reduce}_{\mathrm{b}}\ \mathit{max3}\ \mathit{tree2} \\
& \textbf{where}\ \mathit{max3}\ l\ b\ r = l \uparrow b \uparrow r \\
& \qquad (\lambda c\ b.c + 1, \lambda c.c\ b + 1) = \langle \lambda b.1, \lambda b.1, +, + \rangle_{\mathrm{d}} \\
& \qquad \mathit{max3} = \langle \mathit{id}, \mathit{max3}, \mathit{max3}, \mathit{max3} \rangle_{\mathrm{u}}
\end{aligned}
$$

In this program, parameter functions are first defined, then the main part of the program follows. In the main part, the input tree is read from a file, and then the $\mathsf{dAcc}_{\mathrm{b}}$ and $\mathsf{reduce}_{\mathrm{b}}$ skeletons are called to compute the height of tree, and finally the result is output on the standard output of the root process.

In this section, we show implementation issues of the parallel tree skeletons.

7.1.1 Function Objects

In the SkeTo library, C++ template mechanism and function objects are used to make parallel skeletons generic and efficient. Advantages of these features of C++ for the implementation of parallel skeletons were studied by Striegnitz and Kuchen [76,122]. There are two ways in developing higher-order functions in C++: by pointers to functions and by function objects. In the implementation of the SkeTo library, we utilize function objects for reasons of efficiency and programmability.

In programs developed with skeletons, parameter functions are called so many times that the overhead of calling parameter functions greatly slows the programs down. Using function objects the C++ compilers perform inline-expansion of the function objects, and therefore there is no overhead for calling parameter functions.

In terms of programmability, based on the template mechanism we can generate new function objects by partially binding some arguments and/or composing functions. Striegnitz and Kuchen [76, 122] developed a skeleton library in which functions are curried and allowed to bind some arguments. These techniques are also implemented in the boost library as `boost::bind` and `boost::compose` [1, 71]. Generating new function objects is helpful if we implement user-defined parallel skeletons on the existing parallel skeletons. For example, we implement the rose-tree skeletons, by composing binary-tree skeletons where function objects for the binary-tree skeletons are automatically generated from the functions objects passed to the rose-tree skeletons.

In the SkeTo library, users should define function objects by inferring function objects provided by the SkeTo library. For example, the base class for the binary functions is provided as follows.

```
#include <iostream>
#include <tree_skeletons.h>
using namespace std;

//------------------------------------------------------------------------
// function objects

SKETO_DEF_BINOP( f_dAcc_g, int, int, int,
                 return x + 1; );

SKETO_DEF_UNOP( f_const_one, int, int,
                return 1; );

SKETO_DEF_BINOP( f_plus, int, int, int,
                 return x + y; );

SKETO_DEF_TEROP( f_max3, int, int, int, int,
                 return max( max( x, y ), z ); );

SKETO_DEF_UNOP( f_id, int, int,
                return x; );

//------------------------------------------------------------------------
// main function

int SketoMain( int argc, char *argv[] ) {
  const char* filename_in = argv[ 1 ];
  dist_tree< int, int > *tree1
    = dist_tree< int, int >::read_from_file( filename_in );

  dist_tree< int, int > *tree2
    = tree_skeletons::dAcc( f_dAcc_g, f_dAcc_g,
                            f_const_one, f_const_one, f_plus, f_plus,
                            1, tree1 );

  int result
    = tree_skeletons::reduce( f_max3,
                              f_id, f_max3, f_max3, f_max3,
                              tree2 );

  skeleton::cout << "height of the input tree: " << result << std::endl;

  skeleton::safe_delete( tree2 );
  skeleton::safe_delete( tree1 );
  return 0;
}
```

Figure 7.2. Sample program for computing height of binary tree.

```
template< typename Arg, typename Arg2, typename Res >
struct binary_function {
  typedef Arg first_argument_type;
  typedef Arg second_argument_type;
  typedef Res result_type;
};
```

The type information (e.g., `first_argument_type`) is used in determining the type of parallel skeletons. With this information, we need not to specify types when we call skeletons. For example, the function object `f_dAcc_g` that represents $(\lambda c\ b.c + 1)$ can be defined as follows. Here, the type of the function object is named as `f_dAcc_g_t`.

```
struct f_dAcc_g_t
     : public skeleton::binary_function< int, int, int > {
  int operator(int c, int b) const {return c + 1;}
} f_dAcc_g;
```

To shorten this long definition of function object, we can use the `SKETO_DEF_BINOP` macro.

```
SKETO_DEF_BINOP(f_dAcc_g, int, int, int
                return x + 1; );
```

This new definition of function is easy to develop and read.

Due to the limitation of C macros, the arguments of the function are named as `x` and `y` in this order, and types specified with templates are not allowed.

7.1.2 Implementation of Binary-Tree Skeletons

The implementation algorithm for binary-tree skeletons is in Chapter 6. In this section, we show the interface of skeletons and discuss some coding techniques in the implementation of skeletons in C++.

The binary-tree skeleton library mainly consists of two classes `dist_tree` for binary trees distributed over processors and `tree_skeletons` for providing binary-tree skeletons.

The tree skeletons are separated from class `dist_tree` in two reasons. One reason is the look of the program. We have used Haskell for the discussion of developing skeletal programs. The calls of tree skeletons in the SkeTo library look like programs in Haskell. We discuss this topic in the implementation of the $\mathtt{map}_\mathrm{b}$ skeleton. Another reason is the limitation of template mechanism in C++. If we define two classes with different template parameter, the two classes are considered to be different ones and the private attributes of one class cannot be accessed from the other.

Data Structure for Binary Trees

The distributed binary trees that are manipulated by binary-tree skeletons are defined by the `dist_tree` class. The following code segment shows the definition of `dist_tree`. We have formalized binary trees to have two different types for leaves and for internal nodes. In the implementation in C++, we specify these two types for two template parameters `A` and `B`.

```
template< typename A, typename B >
class dist_tree
{
  friend class local_tree< A, B >;
  friend class tree_skeletons;

  int                  global_size;    // the number of segments
  node_type           *types;          // an array of node-types
  int                 *index_to_proc;  // an array for processor IDs
  local_tree< A, B > **segments;       // an array of pointers to segments
  ...
```

The local segments are distributed over processors. Therefore, among the `segments`, the processor in charge of a segment has an actual pointer to the local segment. The `node_type` is an enumeration type that is used to distinguish a leaf, an internal node, and a terminal node. After the attributes, several functions for input/output are defined.

Local segments are defined by the `local_tree` class that consists of an array for flags of node types and an array of values. The nodes are put in the array in the order of the prefix traversal as shown in Section 6.1.2.

```
template < typename A, typename B >
class local_tree
{
  friend class dist_tree< A, B >;
  friend class tree_skeletons;

  int size;             // the number of nodes
  node_type    *types;  // an array of node-types
  node< A, B > *vals;   // an array of values
  ...
```

There are two ways for implementing the class `node` for nodes. One is by the structure that have two values for a leaf and an internal node disjointly. The other, and used in our implementation, is by the union as shown in the following code.

```
template< typename A, typename B >
struct node
{
  union {
    A val_l; // for the case of a leaf
    B val_n; // for the case of an internal node
  };
};
```

The advantage of using union is to decrease the amount of memory needed for a node, but a disadvantage is that the constructors cannot be used for the types `A` or `B`.

For the cases when the two types of a leaf value and an internal node value are the same, we define the following specialization of the `node` class.

```
template< typename A >
struct node< A, A >
{
  union {
    A val_l; // for the case of a leaf
    A val_n; // for the case of an internal node

    A val;   // for the unified access to the value
  };
};
```

By the new value `val`, we can access to the value uniformly. This uniform access to the value benefits in implementing the uAcc_b and dAcc_b skeletons.

In fact, if we deal with the values for leaves and values for internal nodes separately, we can reduce the amount of memory used and the constructors are allowed. However, this makes the implementation of tree skeletons so complicated that we now do not adopt it.

Implementation of the map_b *Skeleton*

The binary-tree skeletons are implemented in the `tree_skeletons` class. In the following of this section, we show the implementation of the map_b and $\mathsf{reduce}_\mathrm{b}$ skeletons.

The map_b skeleton has the following interface.

```
template< typename K1, typename K2 >
static dist_tree< typename K1::result_type, typename K2::result_type > *
map( const K1 &k1,
     const K2 &k2,
     const dist_tree< typename K1::argument_type,
                      typename K2::argument_type > *t );
```

Only two template parameters for the parameter functions `K1` and `K2` are used in this definition. These two template parameters are guessed by C++ compilers and users need not to specify them.

With this coding technique and the separation of skeletons from the `dist_tree` class, we can call the tree skeletons as the specification in Haskell. For example, the program using the map_b skeleton, $t2 = \mathsf{map}_\mathrm{b}\ k_l\ k_n\ t1$, can be implemented as follows by using our library.

```
dist_tree* t2 = tree_skeletons::map(kl, kn, t1);
```

The type of values are extracted from the definitions of function objects. The local computation is implemented as shown in Section 6.2 using loops on the local segments. The following shows the implementation of the map_b skeleton where the function `map_adapter` is called from `map` with specification of template parameters.

```
template< typename A, typename B, typename C, typename D,
          typename K1, typename K2 >
dist_tree< C, D >* tree_skeletons::map_adapter( const K1 &k1,
                                                const K2 &k2,
                                                const dist_tree< A, B > *t )
{
  dist_tree< C, D >* ret_tree = copy_shape< C, D, A, B >( t );
  for ( int ltree = 0; ltree < t->global_size; ltree++ ) {
    if ( !t->segments[ ltree ] ) continue;
    const local_tree< A, B >* src = t->segments[ ltree ];
    local_tree< C, D >* dist = ret_tree->segments[ ltree ];

    for ( int i = 0; i < src->size; i++ ) {
      switch ( src->types[ i ] ) {
      case LEAF:
        dist->vals[ i ].val_l = k1( src->vals[ i ].val_l ); break;
      case NODE: case TERMINAL:
        dist->vals[ i ].val_n = k2( src->vals[ i ].val_n ); break;
      default: assert( false ); } } }
  return ret_tree;
}
```

Implementation of the $\mathsf{reduce}_\mathrm{b}$ *Skeleton*

The $\mathsf{reduce}_\mathrm{b}$ skeleton takes a parameter function f and for parallel implementation we need four auxiliary functions. We implemented the following function `reduce` so that it takes function objects corresponding to k and four auxiliary functions explicitly. Users should be responsible to whether the auxiliary functions satisfy the condition.

```
template< typename K,
          typename PHI, typename PSIN, typename PSIL, typename PSIR >
static typename K::result_type
reduce( const K &k,
        const PHI &phi,
        const PSIN &psiN,
        const PSIL &psiL,
        const PSIR &psiR,
        const dist_tree< typename K::result_type,
                         typename K::second_argument_type >* t );
```

In the implementation of local reductions, we use a stack to store intermediate values. Here, the values pushed to the stack may have different types in the implementation algorithm in Section 6.2. Since only one value that corresponds to a subtree with a terminal node is in the stack at a time, we implement the value of type `D` (δ in the implementation algorithm) out of the stack. The following segment of code represents the local reductions.

```
D value_d; int depth_d = -1;
std::stack< C > stack_c;

for ( int i = local_tree->size - 1; i >= 0; i-- ) {
  switch ( local_tree->types[ i ] ) {
  case LEAF:
    stack_c.push( local_tree->vals[ i ].val_l );
    if ( depth_d != -1 ) depth_d++;
    break;
  case NODE:
    switch ( depth_d ) {
    case 0: {
      C right = stack_c.top( ); stack_c.pop( );
      value_d = psiL( value_d, phi( local_tree->vals[ i ].val_n), right );
    } break;
    case 1: {
      C left = stack_c.top( ); stack_c.pop( );
      value_d = psiR( left, phi( local_tree->vals[ i ].val_n), value_d );
      depth_d = 0;
    } break;

   ...
```

Another implementation technique is used in the gathering step. To minimize the effect of communication latency, we first pack the data gathered from a process and then communicate it by the asynchronous communication of the MPI (`MPI_Isend` and `MPI_Irecv`).

As seen in Chapter 5, the $\mathsf{reduce}_\mathrm{b}$ and uAcc_b skeletons are often called after the map_b skeleton. In the local computation of these skeletons, loops traverse the arrays that include the values of leaves. Therefore, it is smart to provide functions for the fusions of the map_b and $\mathsf{reduce}_\mathrm{b}$ skeletons or the map_b and uAcc_b skeletons. We implemented these fused skeletons as `map_reduce` and `map_uAcc`.

7.1.3 Implementation of Rose-Tree Skeletons

In this section, we show the implementation of rose-tree skeletons based on their design in Chapter 4. The template mechanism in C++ enables us to develop rose-tree skeletons without modifying the base binary-tree skeletons.

We first show the implementation of the data structure for rose trees, and then show the implementation of the $\mathsf{map}_{\mathrm{r}}$ and $\mathsf{lAcc}_{\mathrm{r}}$ skeletons for examples.

Data Structure for Rose Trees

In our implementation of the rose-tree skeletons, we deal with rose trees in the form of their binary-tree representation in Section 4.3.1. We can implement the class for the binary-tree representation just by using the `dist_tree` class. The following is a segment of code for rose-tree structures where `A` is a template type representing the type of node in a rose tree. Since the leaves are dummy nodes, we assign the same type as the internal nodes.

```
template< typename A >
class dist_rose_tree
{
  dist_tree< A, A >* btree;
  ...
```

In our implementation, we provide several functions for input/output.

Implementation of the $\mathsf{map}_{\mathrm{r}}$ *Skeleton*

Similar to the implementation of binary-tree skeletons, we implement the rose-tree skeletons in another class `rose_tree_skeletons`. We show the implementation of the $\mathsf{map}_{\mathrm{r}}$ skeleton, which is the simplest skeleton of our seven rose-tree skeletons.

Interface of the $\mathsf{map}_{\mathrm{r}}$ skeleton is given in the following code. The first argument is a function object of type `K`, whose argument and return value are of types `K::result_type` and `K::argument_type`, respectively.

```
class rose_tree_skeletons
{
public:
  template< typename K >
  static dist_rose_tree< typename K::result_type > *
  map( const K &k,
       const dist_rose_tree< typename K::argument_type > *t );
```

The implementation of rose-tree skeletons consists of the two parts: the definitions of function objects passed to the binary-tree skeletons, and the wrapper functions which implement the rose-tree skeletons by calling the binary-tree skeletons.

For the $\mathsf{map}_{\mathrm{r}}$ skeleton, we need to define the don't-care function _, which does nothing but keeps consistency of the types. We can generate such a function object `UnaryUndef`, which accepts a value of type `A` and returns a dummy value of type `B` as follows.

```
template< typename A, typename B >
struct UnaryUndef : public skeleton::unary_function< A, B > {
  B operator()( const A& ) const { B dummy; return dummy; }
};
```

By using this function object, we can implement the map_r skeleton as follows. To make the program simple, we insert a function `map_adapter` which is instantiated from the `map` function. The implementation of the skeleton is directly given by calling the `map` skeleton for binary trees.

```
template< typename A, typename B, typename K >
dist_rose_tree< B > *
rose_tree_skeletons::map_adapter( const K &k,
                                  const dist_rose_tree< A > *t )
{
  dist_tree< B, B > *bt
    = tree_skeletons::map( UnaryUndef< A, B >( ), k, t->btree );
  return new dist_rose_tree< B >( bt );
}
```

Implementation of the lAcc_r *skeleton*

We show the implementation of another more complicated skeleton lAcc_r.

In the implementation of the lAcc_r skeleton, we have two values as a tuple (p, a) in the auxiliary functions of uAcc_b. First, we define the structure of tuples as the following `lAcc_in_t` (named from lAcc_r internal type).

```
template < typename A >
struct lAcc_in_t {
  bool p;
  A a;
};
```

As in the case of the map_r skeleton, we implement the function objects for binary-tree skeletons. In the case of the lAcc_r skeleton, we need a constant function for the map_b skeleton, and k, ϕ, ψ_n, ψ_l, and ψ_r for the uAcc_b skeleton. For example, the function object for ϕ_l is constructed from the function object `oplus` for operator $\oplus$ defined as the following structure `func_lAcc_psiL_t`.

```
template < typename A, typename OPLUS >
struct func_lAcc_psiL_t : public skeleton::ternary_function<
                    lAcc_in_t< A >, A, lAcc_in_t< A >, lAcc_in_t< A > > {
  const OPLUS &oplus;
  func_lAcc_psiL_t( OPLUS oplus_ ) : oplus( oplus_ ) {};
  lAcc_in_t< A > operator()( const lAcc_in_t< A >& /* l */,
                             const lAcc_in_t< A >& n,
                             const A& r ) const {
    lAcc_in_t< A > retval;
    if (n.p) { retval.p = false; retval.a = n.a + r; }
    else     { retval.p = false; retval.a = n.a; }
    return retval;
  }
};
```

The other functions can also be defined in the same way.

After defining the function objects, we can straightforwardly implement the lAcc_r skeleton as follows. The `map_uAcc` skeleton is the composition of the map_b skeleton followed by the uAcc_b skeleton.

```
template< typename A, typename OPLUS >
dist_rose_tree< A > *
rose_tree_skeletons::lAcc_adapter( const OPLUS& oplus,
                                   const A& unit_oplus,
                                   const dist_rose_tree< A > *t )
{
  dist_tree< A, A > *bt1 = tree_skeletons::map_uAcc(
      UnaryConst< A, A >( unit_oplus ),
      func_lAcc_k_t< A, OPLUS >( oplus ),
      func_lAcc_phi_t< A >( ),
      func_lAcc_psiN_t< A, OPLUS >( oplus ),
      func_lAcc_psiL_t< A, OPLUS >( oplus ),
      func_lAcc_psiR_t< A, OPLUS >( oplus ),
      t->btree );
  dist_tree< A, A > *bt2
    = tree_skeletons::getchr( unit_oplus, bt1 );
  if ( bt1 ) delete bt1;
  return new dist_rose_tree< A >( bt2 );
}
```

As seen so far, using the function objects and the template mechanism in C++, we can implement the lAcc_r skeleton without much effort. The implemented skeletons however may be worse in efficiency than hand-optimized C++ and MPI programs due to the intermediate data structures passed between the skeletons. The main aim of the implementation in this section is to verify the scalability of the parallel skeletons, and improvement of the efficiency of the parallel skeletons is our future work.

7.2 Optimization Mechanism

We implemented a prototype system of optimization mechanism in OpenC++, which transforms the skeletal parallel programs written in C++. In this section, we first summarize fusion transformation for list skeletons [61], and then show the implementation of the optimization system and experiment results.

7.2.1 List Skeletons and Their Fusion Transformation

The prototype implementation of the optimization mechanism only manipulates list skeletons. Here, we show the definition of list skeletons briefly. Important list skeletons are map, reduce, and scan.

Map applies a function to every element in a list. Informally, we have

$$\mathsf{map}\ k\ [x_1, x_2, \ldots, x_n] = [k\ x_1, k\ x_2, \ldots, k\ x_n].$$

Reduce collapses a list into a single value by repeatedly applying a certain associative binary operator. Informally, for associative binary operator $\oplus$ and initial value e, we have

$$\mathsf{reduce}\ (\oplus)\ e\ [x_1, x_2, \ldots, x_n] = x_1 \oplus x_2 \oplus \cdots \oplus x_n \oplus e.$$

Scan accumulates all intermediate results in the computation of reduce. Informally, for associative binary operator $\oplus$ and initial value e, we have

$$\mathsf{scan}\ (\oplus)\ e\ [x_1, x_2, \ldots, x_n] = [e, e \oplus x_1, e \oplus x_1 \oplus x_2, \ldots, e \oplus x_1 \oplus x_2 \oplus \cdots \oplus x_n].$$

Note that this scan skeleton returns a list that is longer by one than the input list.

To fuse the composition of skeletons into one to eliminate unnecessary intermediate data structures passed between skeletons, one may develop rules to do algebraic transformations on skeletal parallel programs like the authors in [5, 53]. Unfortunately, this would require a huge set of rules to take all possible combinations of skeletal functions into account. In this paper, we borrow the idea of shortcut deforestation [51], which optimizes sequential programs, and simplifies the entire set into just a single rule. The idea is to structure each skeleton with an interface that characterizes how it consumes and produces the parallel data structure.

To manipulate skeletal parallel programs, we structured skeletons with the following three functions: acc, cataJ and buildJ. Let g, p, q be functions, and $\oplus$ and $\otimes$ be associative operators. The skeleton acc, for which we write $[\![g, (p, \oplus), (q, \otimes)]\!]$, is defined by

$$
\begin{aligned}
&[\![g, (p, \oplus), (q, \otimes)]\!]\ [\,]\ e &&= g\ e \\
&[\![g, (p, \oplus), (q, \otimes)]\!]\ (a : x)\ e &&= p\ (a, e)\ \oplus\ [\![g, (p, \oplus), (q, \otimes)]\!]\ x\ (e \otimes q\ a)\ .
\end{aligned}
$$

We define cataJ as a special case of acc, where the accumulative parameter, e, is not used. The skeleton cataJ, for which we write $(\![\oplus, p, e]\!)$, is defined by

$$
\begin{aligned}
&(\![\oplus, p, e]\!)\ [\,] &&= e \\
&(\![\oplus, p, e]\!)\ (a : x) &&= p\ a \oplus (\![\oplus, p, e]\!)\ x\ .
\end{aligned}
$$

Indeed, the cataJ skeleton is a specialization of accumulate as follows.

$$(\![\oplus, p, e]\!) = [\![id, (p \circ fst, \oplus), (\_, \_)]\!]$$

The last function, buildJ, is to standardize the production of join-lists with implicit parallelism.

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

Here, $[.]$ stands for the function $\lambda a.[a]$, used for construction a singleton list from an element.

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

$$
\begin{aligned}
&\mathsf{map}\ f = \mathsf{buildJ}\ (\lambda c\ s\ e.\ (\![c, s \circ f, e]\!)) \\
&\mathsf{reduce}\ (\oplus)\ e = (\![\oplus, id, e]\!) \\
&\mathsf{scan}\ (\oplus)\ e\ x = \mathsf{buildJ}\ (\lambda c\ s\ e.\ [\![s, (\lambda(a, e').\ s\ e', c), (id, \oplus)]\!])\ x\ e
\end{aligned}
$$

Following the thought in [51], we may define our shortcut fusion for join lists as follows. This rule is called *CataJ-BuildJ* rule.

$$(\![c, s, e]\!) \circ \mathsf{buildJ}\ gen = gen\ c\ s\ e$$

An example of applying this rule shows that the reduce skeleton after the map skeleton can be fused into a single cataJ skeleton.

$$
\begin{aligned}
(\mathsf{reduce}\ (\oplus)\ e) \circ (\mathsf{map}\ k) = &\quad \{\text{structured form for the map and reduce skeletons}\} \\
&(\![\oplus, id, e]\!) \circ (\mathsf{buildJ}\ (\lambda\ c\ s\ e.\ (\![c, s \circ f, e]\!))) \\
= &\quad \{\text{CataJ-BuildJ rule}\} \\
&(\lambda\ c\ s\ e.\ (\![c, s \circ f, e]\!))\ (\oplus)\ id\ e \\
= &\quad \{\text{lambda application}\} \\
&(\![\oplus, f, e]\!)
\end{aligned}
$$

A variant of this rule is given as follows for the case that the both functions are enclosed by buildJ. We call the following *BuildJ(CataJ-BuildJ)* rule.

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

Finally, we generalize this rule to the following most generic fusion rule for accumulate.

Definition 7.1 (BuildJ(Acc-BuildJ) Rule [61])

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

□

7.2.2 Implementation of Optimization Mechanism

Figure 7.3 overviews our optimization system, which consists of three components:

- the user-interface database,
- the generic transformation engine, and
- the implementation database.

Taking a C++ program with skeletons, our transformation system first converts the skeletons into *structured form* (an intermediate form) by applying rules given as meta-programs in the user-interface database. The generic transformation engine manipulates and fuses the structured form with the shortcut fusion rules. Finally, our system links the optimized program with efficiently implemented skeletons in our library by the rules in the implementation database.

The optimization system is implemented in OpenC++, a meta language for manipulating C++ programs.

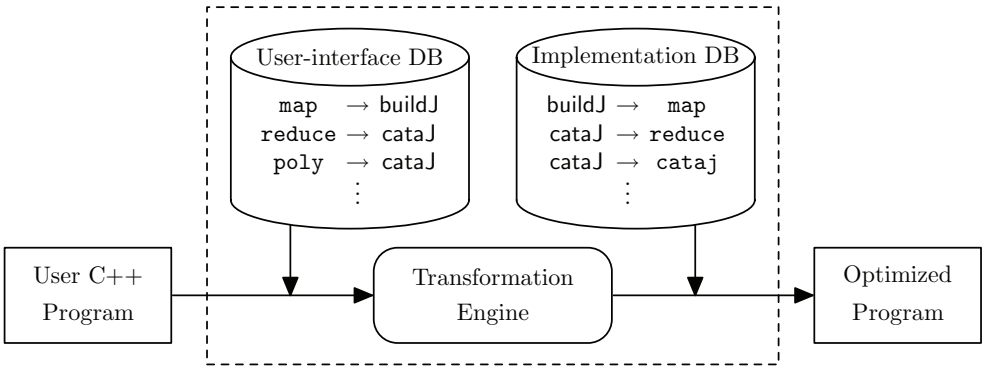


Figure 7.3. Overview of the optimization mechanism.

Transformation to Structured Forms

In OpenC++, the program text is accessible at the meta level in the form of a parse tree, represented as a nested list of logical tokens. A part of the C++ program to compute the variance

```
sum = as->reduce(add, 0.0);
ave = sum / size;
subs = as->map1(sub, ave);
sqs = subs->map(sq);
sq_sum = sqs->reduce(add, 0.0);
```

is converted into the following parse tree.

```
[[sum = [[as -> reduce] ( [add , 0.0] )]] ;]
[[ave = [sum / size]] ;]
[[subs = [[as -> map1] ( [sub , ave] )]] ;]
[[sqs = [[subs -> map] ( [sq] )]] ;]
[[sq_sum = [[sqs -> reduce] ( [add , 0.0] )]] ;]
```

We define the rules to convert user skeletons to structured form and vice versa in OpenC++. For example, a meta program that implements the conversion of a map skeleton into buildJ form

$$\mathsf{map}\ f\ as = \mathsf{buildJ}\ (\lambda c\ s\ e.(\!|c, f \circ s, e|\!))$$

can be implemented as follows.

```
Ptree* map_to_buildJ( Ptree *sentence )
{
  Ptree *dst, *src, *function;
  if (Ptree::Match( sentence, "[[%? = [[%? -> map] ( %? )]] ;]",
                    &dst, &src, &function) ) {
    return make_buildJ( dst, src,
                        Ptree::List( var_c ),
                        Ptree::List( var_s, function ),
                        Ptree::List( var_e ));
  }
  ...
```

The user-interface database is a correction of these functions from list skeletons to their structured forms. The reflection mechanism in OpenC++ enables pattern matching and function composition to be easily implemented. Thus, we can easily convert skeletons to their structured forms.

Using conversion with our user-interface library, the last three lines in the parse tree above are converted into the following structured forms. st

```
[‘buildJ‘ subs as [[var_c] [var_s [sub ave]] [var_e]] ;]
[‘buildJ‘ sqs subs [[var_c] [var_s [sq]] [var_e]] ;]
[‘cataJ‘ sq_sum sqs [[add] [func_id] [0.0]] ;]
```

Here, terms `var_c`, `var_s`, and `var_e` are used to represent three arguments of the buildJ, $+\!\!+$, $[\cdot]$, and $[\ ]$.

Generic Transformation Engine

Our system implements the fusion rule in [65] and it repeatedly applies the rule on structured forms. We restricted the elements in structured forms so that they were represented as a composition of functions. This simplified the application of the fusion rule so that just the occurrences of a bound variable to the corresponding argument had to be replaced. Note that such a restriction is insignificant since reflection can take care of it.

Our generic transformation engine applies the *CataJ-BuildJ* rule [65] twice on the structured forms above in our running example,

```
[‘buildJ‘ subs as [[var_c] [var_s [sub ave]] [var_e]] ;]
[‘buildJ‘ sqs subs [[var_c] [var_s [sq]] [var_e]] ;]
[‘cataJ‘ sq_sum sqs [[add] [func_id] [0.0]] ;]
```

and optimizes it into a single cataJ form as follows.

```
[‘cataJ‘ sq_sum as [[add] [func_id [sq] [sub ave]] [0.0]] ;]
```

Transformation from Structured Forms

Finally, the optimization system puts the structured form back into one of the parallel skeletons. Due to the generality of the structured forms, we need to select a suitable skeleton implementation based on the parameters of the structured forms.

Two examples are shown as follows: the first one is from a combination of the map and reduce skeletons, and the second one is from a single reduce skeleton.

```
[‘cataJ‘ sq_sum as [[add] [func_id [sq] [sub ave]] [0.0]] ;]

[‘cataJ‘ sum as [[add] [func_id] [0.0]] ;]
```

For the first structured form, we map it onto the cataj skeleton implemented in the skeleton library. Though the cataj skeleton is semantically equal to the composition of map and reduce skeletons, the cataj skeleton provides more efficient implementation by fusion of them. For the second structured form, since the second argument is the identity function `func_id`, we map the structured form onto the reduce_{b} skeleton.

7.2.3 Experimental Results for Optimization

To see how efficient the generated optimized program is, we compared it with the following two programs for the variance problem [92]:

(1) the original program using a map skeleton that produces new data (duplicative map),

(2) another program using a map skeleton that overwrites the input data (overwriting map).

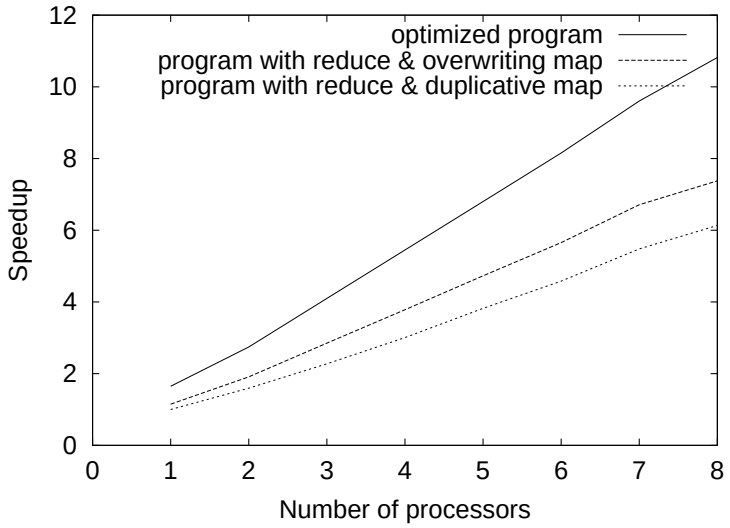


Figure 7.4. Experimental results for the optimization mechanism. Speedups are plotted against the program with reduce and duplicative map executed with one processor.

In the second program, the individual skeletons are optimized and they do not generate unnecessary data. We implemented these programs and did our experiments on a cluster of four Pentium 4 Xeon 2.0-GHz dual-processor PCs with 1-GB of memory, connected through a Gigabit Ethernet. The OS was FreeBSD 4.8 and we used gcc 2.95 for the compiler.

Figure 7.4 plots the results of speedups to the original program with one processor for an array of 1,000,000 elements. The computation time for the original program with one processor is 1.67 s and the computation times for (1), (2), and the optimized one with eight processors are 0.243 s, 0.197 s, and 0.138 s, respectively. As a natural consequence of using the skeletons, all programs demonstrated outstanding scalability. Comparison with (2) proves the success of our framework: The effect of fusion far exceeds individual refinements on each skeleton.

7.3 Code Generator

We have studied three algebraic properties in Section 5.2, with which we can derive auxiliary functions for binary-tree skeletons systematically. Among them, the tupled-ring property gives a clear condition for parallelizing tree manipulations with multiple parameters, but developing auxiliary functions is somehow tedious due to the large number of parameters introduced in the matrices.

To encourage programmers to develop parallel programs based on the tupled-ring property, we have developed a system that automatically translates users' recursive specifications into parallel C++ codes. Figure 7.5 depicts the outline of our code generator. In this

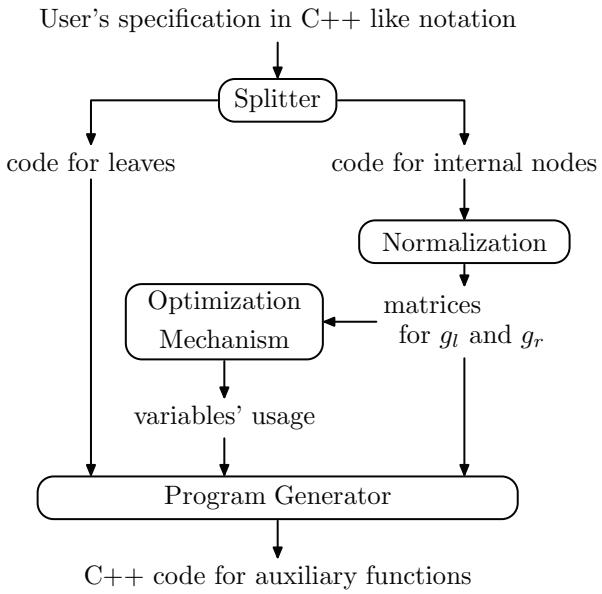


Figure 7.5. Outline of our code generator.

system, not only the parallel code is generated but also the generated code is optimized by removing the computation on constants.

In this section, we describe the outline of our code generator, and then demonstrate the parallelization steps with an example of simple XPath query.

Running Example: A Simple XPath Query

In this section, we demonstrate our code generator with a simple XPath query [13]. As our running example, we consider the following XPath query.

```
//x[./y/following-sibling::z]
```

This XPath query searches for a node labeled `x` that has children labeled `y` and `z` in this order from left (Note that the node `x` may have other children). The detailed derivation of skeletal parallel programs for XPath queries will be studied in Chapter 9. For simplicity, let the input tree is given in the binary-tree representation in Figure 4.9, and here we want to determine whether there is a node matching to the XPath query or not.

An automaton corresponding to the XPath query on the binary-tree representation is given in Figure 7.6. Based on this automaton, we can develop a sequential program that computes this simple query. The query is computed by $(\mathit{fst} \circ \mathit{xpq})$ where the function xpq is a tree homomorphism given as follows. In the function xpq, three values of a tuple corresponds to the states S_0, S_1, and S_2 in the automaton. Note that the program traverses the automaton in the reversed direction.

$$
\begin{aligned}
&\mathit{xpq}\ (\mathsf{BLeaf}\ a) = (\mathsf{False}, \mathsf{False}, \mathsf{False}) \\
&\mathit{xpq}\ (\mathsf{BNode}\ l\ b\ r) \\
&\quad = \mathbf{let}\ (l_0, l_1, l_2) = \mathit{xpq}\ l;\ (r_0, r_1, r_2) = \mathit{xpq}\ r \\
&\quad\quad \mathbf{in}\ (l_0 \vee r_0 \vee (b == \text{“}\mathtt{x}\text{”} \wedge l_1), r_1 \vee (b == \text{“}\mathtt{y}\text{”} \wedge r_2), r_2 \vee b == \text{“}\mathtt{z}\text{”})
\end{aligned}
$$

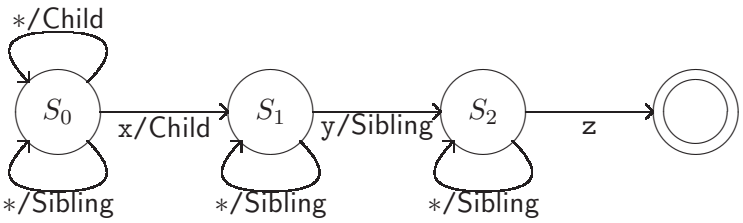


Figure 7.6. A nondeterministic automaton for the sample XPath query. The initial state is S_0. Transition labeled x/Child occurs when a node is labeled as x and we traverse to the child. Transition labeled x/Sibling occurs when a node is labeled as x and we traverse to the sibling.

7.3.1 Input of Code Generator

The code generator takes recursive functions written in C++ like notation with some annotations. A input code for the running example is given in Figure 7.7. Users are allowed to write their programs using operators, functions, and `if`-statements. We introduced a notation for tuples such as

```
tuple<bool> (l0, l1, l2) = xpq(n.l);
```

to enable users to write tree algorithms computing multiple values concisely. Since it is hard to find automatically two operators that form a commutative semiring, we ask users to specify the properties among operators (i.e., commutative semirings) as annotations. Annotation consists of the type of the values and two operators with their units.

Our code generator first splits the specification into two parts corresponding to two cases for leaves and for internal nodes by finding `if`-statement with the predicate of `is_leaf()`.

For the example XPath query, we can write a recursive function shown in Figure 7.7. With the notation of tuples, we can write the program in almost the same way as the specification above. Note that the first line

```
// semiring(bool, ||, &&, false, true);
```

```
// semiring(bool, ||, &&, false, true);
tuple<bool> xpq(node< char > n) {
  if (n.is_leaf()) {
    return (false, false, false);
  } else {
    tuple<bool> (l0, l1, l2) = xpq(n.l);
    tuple<bool> (r0, r1, r2) = xpq(n.r);
    bool v0 = l0 || r0 || ((n.v == 'x') && l1);
    bool v1 = r1 || ((n.v == 'y') && r2);
    bool v2 = r2 || (n.v == 'z');
    return (v0, v1, v2);
  }
}
```

Figure 7.7. A sample code for input specification.

is an annotation to let the system know the commutative semiring $\{\mathsf{Bool}, \vee, \wedge\}$.

The system first splits the definitions for leaves and internal nodes, and parses the definition for internal nodes. In this analysis, the system generates an abstract syntax tree corresponding to the following segment of program with annotation.

```
//semiring (bool, ||, &&, false, true)
//recursion (l0, l1, l2) (r0, r1, r2)
//node      v
//results   (v0, v1, v2)
v0 = l0 || r0 || ((n.v == 'x') && l1)
v1 = r1 || ((n.v == 'y') && r2)
v2 = r2 || (n.v == 'z')
```

7.3.2 Normalization

The system then transforms the abstract syntax tree into the two bi-linear polynomial forms

$$k\ \boldsymbol{l}\ b\ \boldsymbol{r} = g_l\ (b, \boldsymbol{r}) \ominus \boldsymbol{l}\ , \text{ and}$$
$$k\ \boldsymbol{l}\ b\ \boldsymbol{r} = g_r\ (b, \boldsymbol{l}) \ominus \boldsymbol{r}\ ,$$

to generate the matrices specified as functions g_l and g_r. This normalization is performed in the following three steps.

1. We expand the expressions by using the distributive law $x \otimes (b \oplus c) = (x \otimes b) \oplus (x \otimes c)$. Note that any operation distributes over the `if`-statement, that is, for any operator $\oplus$, we have (**if** p **then** a **else** b) $\oplus\ x =$ **if** p **then** $a \oplus x$ **else** $b \oplus x$.

2. We flatten the expression with the associative law, and sort the arguments with the commutative law.

3. For each argument we put sub-expressions together by using the distributive law in the reversed direction. Here if there is no occurrence of an argument x_i, then we insert $(\iota_\oplus \otimes x_i)$ that is equal to $\iota_\oplus$.

After normalization, the system checks that each expression is linear polynomial, and then generates the matrices for two functions g_l and g_r.

Let us see back to our running example. For example, the normalization of the following equation for `v0` against parameters `r0`, `r1` and `r2` is done as follows.

```
v0 = l0 || r0 || ((n.v == 'x') && l1)
```

At the first step, the system applies the distributive law to expand all the parts related to the parameters. The system does nothing for the equation, since the operator `&&` is inside of `||`. The system then sorts the subexpressions with respect to the arguments `r0`, `r1` and `r2`.

```
v0 = l0 || r0 || ((n.v == 'x') && l1)
   = r0 || l0 || ((n.v == 'x') && l1)
```

Finally, the system applies the distributive law in the reversed direction. For the arguments `r1` and `r2`, there are no occurrences of the arguments and thus the system inserts a special value `Z` (named from zero-element, $\iota_\oplus$) with the arguments. For the argument `r0`, there is no coefficient and thus the system inserts a special value `I` (named from identity-unit, $\iota_\otimes$). Therefore, the system transforms the equation as follows.

```
v0 = r0 || l0 || ((n.v == 'x') && l1)
   =    (I && r0)
     || (Z && r1)
     || (Z && r2)
     || (l0 || ((n.v == 'x') && l1))
```

After normalizing all the expressions with respect to the parameters `r0`, `r1` and `r2`, we obtain the following 4×4 matrix for the function g_r.

```
I       Z       Z              (l0 || ((n.v == 'x') && l1))
Z       I       (n.v == 'y')   Z
Z       Z       I              (n.v == 'z')
Z       Z       Z              I
```

In the same way, we obtain the following matrix for the function g_l.

```
I       (n.v == 'x')     Z      r0
Z       Z                Z      r1 || ((n.v == 'y') && r2)
Z       Z                Z      r2 || (n.v == 'z')
Z       Z                Z      I
```

7.3.3 Optimization by Removing Constants

In many cases some elements keep the same value throughout the computation of tree skeletons. If some values do not change throughout the computation of tree skeletons, we can save the memory space and computations for the values. In our code generator, we implemented the optimization by simulating the computation of tree skeletons with abstract values.

After deriving the matrices, the system proceeds into the optimization phase. In the optimization phase, the system abstracts the values to three values, 0, 1, and $*$:

- a 0 denotes an element keeping the value being the zero-element of the commutative semiring ($= \iota_\oplus$);
- an 1 denotes an element keeping the value being the identity-unit of the commutative semiring ($= \iota_\otimes$);
- an $*$ denotes an element that is neither 0 nor 1, listed above.

First, the system compares the corresponding values in the matrices for g_l and g_r, and generates an initial matrix for the analysis. In this initial matrix, the $*$ elements denote that the values on the positions are required to the tree contraction algorithms. The

$\oplus'$	0	1	$*$
0	0	1	$*$
1	1	$*$	$*$
$*$	$*$	$*$	$*$

$\otimes'$	0	1	$*$
0	0	0	0
1	0	1	$*$
$*$	0	$*$	$*$

$\odot$	0	1	$*$
0	0	$*$	$*$
1	$*$	1	$*$
$*$	$*$	$*$	$*$

Figure 7.8. Semantics of three operations on three values for updating abstract matrices.

system then simulates the computation of tree skeletons, by squaring the matrix using the operators given in Figure 7.8. We update the abstract matrix by the following recurrence equation

$$A^{n+1} = A^n \times_{\otimes',\oplus'} A^n +_{\odot} A^n$$

until the matrix reaches fixpoint, that is $A^{n+1} = A^n$. Note that the iteration terminates, since during the squaring the matrix the value changes only to $*$, and once an element has the value $*$ then the value never changes any more. In the result matrix, the value `V` indicates that the element should be computed through the tree contraction because the value may change; and the other values, 0s and 1s, denote that the elements do not change through the tree contractions and we can remove them by substituting the values to the variables. Thus, this optimization can reduce the computation time as well as the memory space during the computation of tree skeletons.

For our running example, the system perform the optimization as follows. First, by comparing two matrices for g_l and g_r, the system yields the following abstract matrix. The value of an element is 0 if corresponding elements in both matrices have value `Z`, 1 if corresponding elements in both matrices have value `I`, and $*$ otherwise.

$$A^0 = \begin{pmatrix} 1 & * & 0 & * \\ 0 & * & * & * \\ 0 & 0 & * & * \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Then the system updates the abstract matrix using the recurrence equation.

$$A^1 = \begin{pmatrix} 1 & * & 0 & * \\ 0 & * & * & * \\ 0 & 0 & * & * \\ 0 & 0 & 0 & 1 \end{pmatrix} \times_{\oplus',\otimes'} \begin{pmatrix} 1 & * & 0 & * \\ 0 & * & * & * \\ 0 & 0 & * & * \\ 0 & 0 & 0 & 1 \end{pmatrix} +_{\odot} \begin{pmatrix} 1 & * & 0 & * \\ 0 & * & * & * \\ 0 & 0 & * & * \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & * & * & * \\ 0 & * & * & * \\ 0 & 0 & * & * \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

In fact, the abstract matrix A^1 reaches fixpoint. We can verify the fact by computing the next abstract matrix A^2 as follows.

$$A^2 = \begin{pmatrix} 1 & * & * & * \\ 0 & * & * & * \\ 0 & 0 & * & * \\ 0 & 0 & 0 & 1 \end{pmatrix} \times_{\oplus',\otimes'} \begin{pmatrix} 1 & * & * & * \\ 0 & * & * & * \\ 0 & 0 & * & * \\ 0 & 0 & 0 & 1 \end{pmatrix} +_{\odot} \begin{pmatrix} 1 & * & * & * \\ 0 & * & * & * \\ 0 & 0 & * & * \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & * & * & * \\ 0 & * & * & * \\ 0 & 0 & * & * \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

In the abstract matrix that reaches fixpoint, there are eight $*$s. Therefore, we only need to compute these eight values not all the sixteen elements. In other words, we can reduce the number of variables in a matrix to a half.

7.3.4 Code Generation

The system finally generates parallel programs where C++ code calling a parallel skeleton is generated. For the case of the $\mathsf{reduce}_\mathrm{b}$ skeleton, generated components are as follows:

- data structure for tuples, matrices, and intermediate values,
- associative operator of matrix multiplication,
- parameter functions of the skeleton, k_l and k_n, and
- auxiliary functions for parallel implementation ϕ, ψ_n, ψ_l, and ψ_r.

In the generated code, a matrix has values corresponding to the $*$ elements in the optimization step. In addition to this optimization, a matrix has an additional flag for the case of identity matrix. Since in the computation of the $\mathsf{reduce}_\mathrm{b}$ and uAcc_b skeletons an identity matrix is assigned to each node, this flag for identity matrix is important in terms of efficiency. This flag for the identity matrix may reduce the number of $*$ elements on the diagonal line.

For our running example, the system generates C++ code as shown in Figure 7.9. The structures `xpq_ret_t`, `xpq_matrix_t`, and `xpq_inter_t` are for the resulting tuples, matrices with eight elements, and intermediate values, respectively, which are used in the computation of the $\mathsf{reduce}_\mathrm{b}$ skeleton. After the definition of the structures, the definition of function objects for the $\mathsf{reduce}_\mathrm{b}$ skeleton are given. Using these function objects, we can develop a parallel program for the sample XPath query.

7.4 Experiment Results

We have developed skeletal parallel programs listed below, and make several experiments to measure their scalability. The source code of all the parallel programs will be available at the website of the SkeTo library.

Programs Developed with Binary-Tree Skeletons

- **height**: This program computes the height of a given binary tree with the map_b and $\mathsf{reduce}_\mathrm{b}$ skeletons. The input is a randomly generated tree with 16,777,215 ($= 2^{24}-1$) nodes.
- **diameter**: This program computes the diameter of a given binary tree. This problem is a maximum weight-sum problem and the program is given in Section 8.6.2 with the map_b and $\mathsf{reduce}_\mathrm{b}$ skeletons. The input is a randomly generated tree with 16,777,215 ($= 2^{24} - 1$) nodes.

```
struct xpq_ret_t {
  int v0, v1, v2;
};

struct xpq_matrix_t {
  bool flag;
  bool a01, a02, a03, a11, a12, a13, a22, a23;
  xpq_matrix_t( ) {
    flag = true;
  }
};

struct xpq_inter_t {
  char b;
  xpq_matrix_t a;
};

xpq_matrix_t operator*( const xpq_matrix_t &lhs,
                        const xpq_matrix_t &rhs ) {
  if (lhs.flag) return rhs;
  if (rhs.flag) return lhs;

  xpq_matrix_t ret;
  ret.a01 = rhs.a01 || (lhs.a01 && rhs.a11);
  ret.a02 = rhs.a02 || (lhs.a01 && rhs.a12) || (lhs.a02 && rhs.a22 );

  ...  (snip) ...

  ret.a23 = (lhs.a22 && rhs.a23) || lhs.a23;
  return ret;
}

SKETO_DEF_UNOP( xpq_kl, char, xpq_ret_t,
  xpq_ret_t ret;
  ret.v0 = false; ret.v1 = false; ret.v2 = false;
  return ret;
);

SKETO_DEF_TEROP( xpq_kn, xpq_ret_t, char, xpq_ret_t, xpq_ret_t,
  xpq_ret_t ret;
  ret.v0 = x.v0 || z.v0 || ((y == 'x') && x.v1);
  ret.v1 = z.v1 || ((y == 'y') && z.v2);
  ret.v2 = z.v2 || (y == 'z');
  return ret;
);

SKETO_DEF_UNOP( xpq_phi, char, xpq_inter_t,
  xpq_inter_t ret;
  ret.b = x;
  return ret;
);

  ... (snip) ...

SKETO_DEF_TEROP( xpq_psiR, xpq_ret_t, xpq_inter_t, xpq_inter_t,
                 xpq_inter_t,
  xpq_matrix_t gr; gr.flag = false;
  gr.a01 = false; gr.a02 = false; gr.a03 = x.v0 || ((y.b == 'x') && x.v1);
  gr.a11 = true; gr.a12 = (y.b == 'y'); gr.a13 = false;
  gr.a22 = true; gr.a23 = (y.b == 'z');

  xpq_inter_t ret;
  ret.b = z.b; ret.a = y.a * gr * z.a;
  return ret;
);
```

Figure 7.9. Segments of generated code for the sample XPath query.

- **prefix** (prefix numbering): This program assigns a number for each node in the order of prefix traversal on a given binary tree. The program is given in Section 5.1.1 and uses the map_b, uAcc_b, and dAcc_b skeletons. The input is a randomly generated tree with 16,777,215 ($= 2^{24} - 1$) nodes.

- **3mcs** (maximum three connected-set sum): This problem is an instance of the maximum weight-sum problem in which the predicate validates that the nodes marked as True form at most three connected components. The program uses the map_b and $\mathsf{reduce}_\mathrm{b}$ skeletons where the intermediate value of auxiliary functions for the $\mathsf{reduce}_\mathrm{b}$ skeleton is the upper-right triangle of a 7×7 matrix. The input is a randomly generated tree with 16,777,215 ($= 2^{24} - 1$) nodes.

- **xpath-s** (small XPath query): This program perform a small XPath query

  ```
  /desc::*[desc::b/child::d]
  ```

  in parallel. The input is a binary tree representing a randomly generated rose tree with 100,000 nodes. This program uses the map_b and uAcc_b skeleton.

- **xpath-lr** (large XPath query for random tree): This program perform a larger XPath query

  ```
  /desc::*[desc::b/child::d]/desc::c[desc::u/child::w]/desc::f
  ```

  in parallel. The input is a binary tree representing a randomly generated rose tree with 100,000 nodes. This program uses the map_b, $\mathsf{zipwith}_\mathrm{b}$, uAcc_b and dAcc_b skeleton.

- **xpath-lf** (large XPath query for flat tree): This program perform the larger XPath query in parallel. The input is a binary tree representing a flat rose tree with its height at most ten. The number of nodes is 100,000.

Programs Developed with Rose-Tree Skeletons

- **prefix-Rb** (prefix numbering on complete binary-tree): The program given in Section 4.2.2 that assigns a number for each node in the order of prefix traversal. This program uses the map_r, uAcc_r, rAcc_r, $\mathsf{zipwith}_\mathrm{r}$, and dAcc_r skeletons. The input is a complete binary tree (dealt with as a rose tree) with 4,194,303 ($= 2^{22} - 1$) nodes.

- **prefix-Rr** (prefix numbering on random tree): The program of prefix numbering is executed on randomly generated rose tree. The number of nodes is 4,194,303 ($= 2^{22} - 1$) nodes.

Table 7.1. Execution times and speedups for programs height, diameter, prefix, and 3mcs, developed with binary-tree skeletons. The execution times are in seconds and the speedups are given in the parentheses.

P	height	diameter	prefix	3mcs
1	0.274 (1.00)	0.488 (1.00)	1.686 (1.00)	2.298 (1.00)
2	0.136 (2.02)	0.248 (1.97)	0.902 (1.87)	1.174 (1.96)
4	0.068 (3.98)	0.131 (2.91)	0.429 (3.93)	0.589 (3.90)
6	0.044 (6.23)	0.086 (3.72)	0.289 (5.84)	0.398 (5.77)
8	0.034 (7.94)	0.092 (5.69)	0.230 (7.34)	0.301 (7.63)
12	0.023 (11.68)	0.044 (5.28)	0.201 (8.40)	0.202 (11.35)
16	0.017 (16.50)	0.034 (11.15)	0.121 (13.90)	0.156 (14.77)
24	0.012 (22.64)	0.026 (14.33)	0.096 (17.57)	0.104 (22.06)
32	0.011 (24.06)	0.020 (18.99)	0.080 (21.01)	0.086 (26.78)

The parallel program is executed on a PC-cluster composed of uniform PCs with two Pentium 4 2.8-GHz CPUs (only one CPU is used in experiments) and 2-GByte memory connected with Gigabit Ethernet. The compiler and MPI library used are gcc 4.1.1 and MPICH 1.2.7, respectively.

Tables 7.1, 7.2 and 7.3 show the execution times without initial distribution and final gathering of data and the speedup against the case executed with one processor. We show the execution times without initial distribution and final gathering since the computational cost of the examples is rather small with respect to the the cost of distribution and gathering. We can omit distribution and gathering if tree skeletons are called successively. Note that the execution times listed are almost the same as those of sequential programs since the skeletons have almost no overhead for the execution on one processor. As seen from Figures 7.10, 7.11 and 7.12 that plots the experimental results, all the skeletal parallel programs achieved good scalability.

7.5 Short Summary

In this chapter, we have shown the implementation issues of our skeleton library named SkeTo. The SkeTo library mainly consists of two parts: the parallel skeleton library for parallel data structures lists, matrices, and trees; the optimization mechanism implemented by the meta-programming technique. We have also developed a code generator with optimization for the computation with tupled-ring property. We have confirmed the efficacy of parallel programs developed on the SkeTo library using several examples.

The SkeTo library is still growing up. For example, we are now working on the implementation of parallel skeletons for more flexible parallel data structures such as lists with variable length, and the implementation of fusion rules for matrices and trees. Another implementation specific to parallel computational models such as Cell Broadband Engine Architecture (Cell BE) [64] would be an interesting future research.

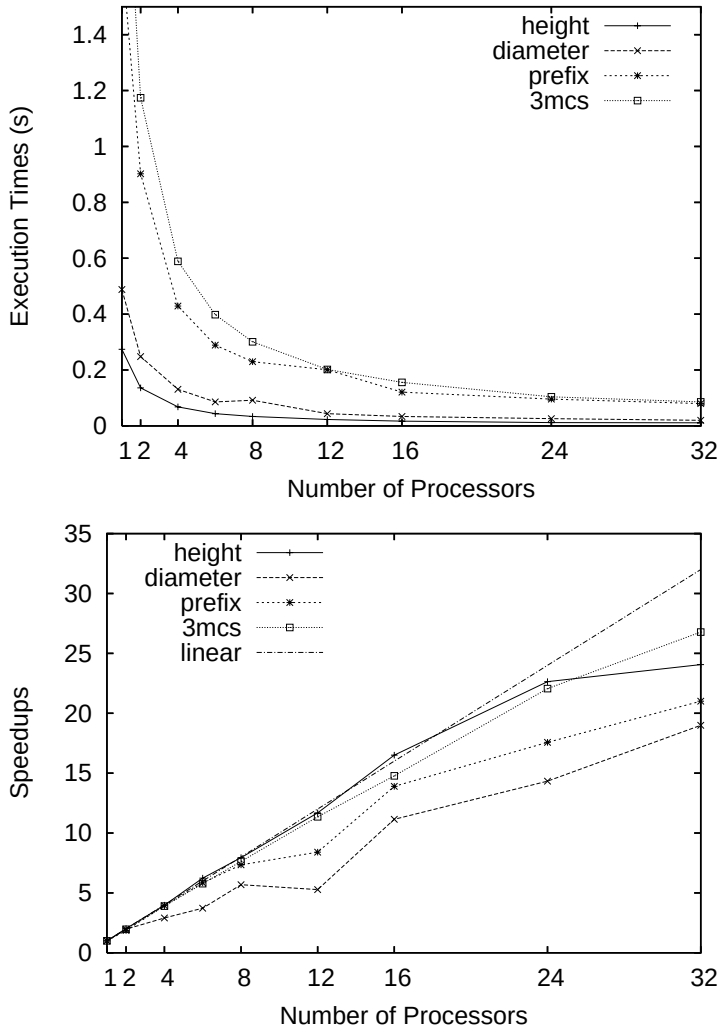


Figure 7.10. Experimental results for programs height, diameter, prefix, and 3mcs, developed with binary-tree skeletons.

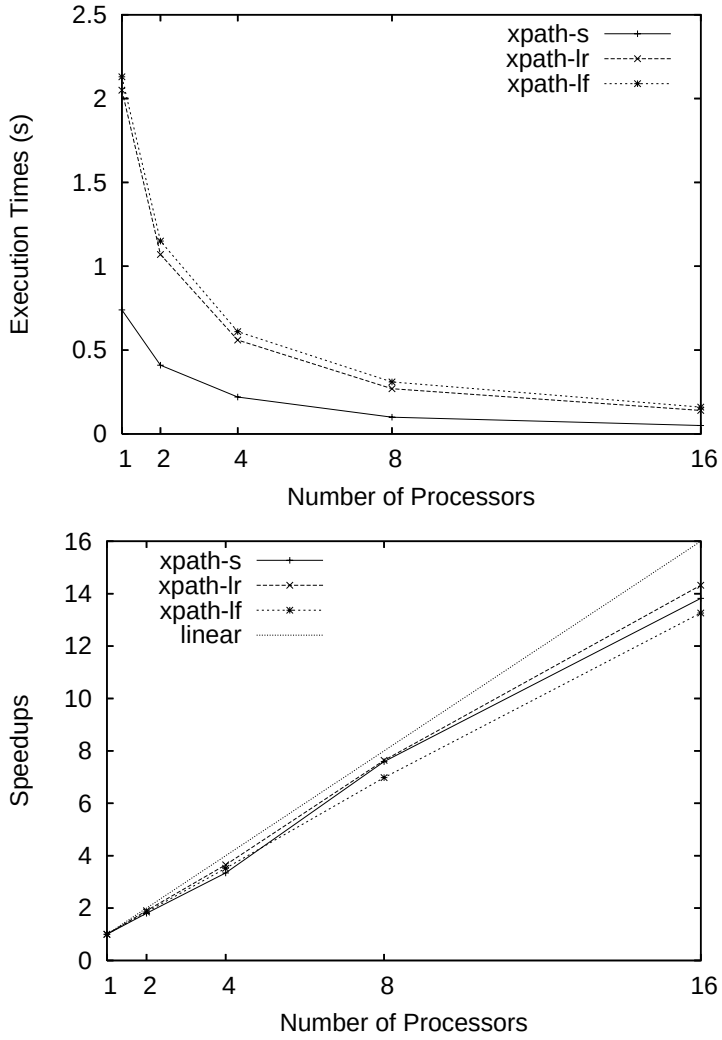


Figure 7.11. Experimental results for programs xpath-s, xpath-lr and xpath-sf, developed with binary-tree skeletons.

Table 7.2. Execution times and speedups for programs xpath-s, xpath-lr, and xpath-lf, developed with binary-tree skeletons. The execution times are in seconds and the speedups are given in the parentheses.

P	xpath-s	xpath-lr	xpath-lf
1	0.74 (1.00)	2.05 (1.00)	2.13 (1.00)
2	0.41 (1.80)	1.07 (1.90)	1.15 (1.85)
4	0.22 (3.34)	0.56 (3.65)	0.61 (3.52)
8	0.10 (7.59)	0.27 (7.64)	0.31 (6.98)
16	0.05 (13.82)	0.14 (14.32)	0.16 (13.26)

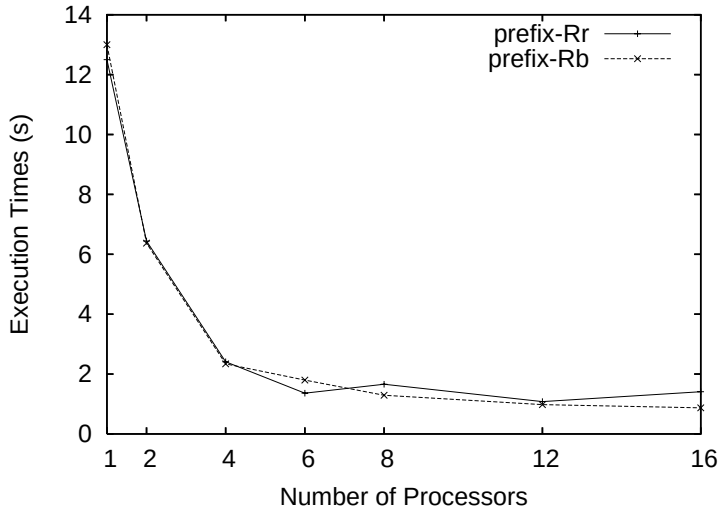


Figure 7.12. Experimental results for programs developed with rose-tree skeletons.

Table 7.3. Execution times and speedups for the programs developed with rose-tree skeletons. The execution times are in seconds and the speedups are given in the parentheses.

P	prefix-Rr	prefix-Rb
1	12.5 (1.00)	13.0 (1.00)
2	6.44 (1.94)	6.37 (2.04)
4	2.41 (5.19)	2.34 (5.55)
6	1.36 (9.19)	1.80 (7.22)
8	1.66 (7.53)	1.29 (10.1)
12	1.08 (11.6)	0.98 (13.3)
16	1.41 (8.86)	0.87 (14.9)

Chapter 8

Parallelizing Maximum Marking Problems

In the following two chapters, we study the expressiveness of parallel tree skeletons by showing the derivation of parallel programs for two classes of non trivial problems.

In this chapter, we study the maximum marking problems [17, 116] that cover a wide class of optimization problems. The generic specification of the problems is as follows.

> Given a data structure x and a predicate p, the task is to find a way to mark some elements in x, such that the marked data structure x^* satisfies a certain property p and the weight-sum of the marked elements gets maximum.

In this chapter, we assume each marked element to be a boolean value, True or False.

Parametrization of the data structure and the predicate gives this formalization great flexibility. For example, let x be a list and p be a predicate that allows at most one set of continuous Trues, we can specify the maximum segment sum problem over lists, which is a well-known problem in Bentley's programming pearls [12]. The party planing problems on binary trees, appeared as the running example in Introduction, is also a maximum marking problem where x is a binary tree and p invalidates markings in which there exist marked elements next to each other.

Many researchers have studied the derivation of parallel programs for instances of the maximum marking problems. Fisher and Ghuloum [43] developed a system that can derive a parallel program for the maximum segment sum problem on lists. Cole [34] showed derivation of a parallel program for the same problem based on list homomorphism. Several researchers have studied parallel algorithms for the maximum segment sum problem on lists (arrays) and the maximum subarray sum problem (maximum segment sum problem on two-dimensional arrays) for several parallel-computation models [6, 9, 59, 106, 109, 127]. For tree structures, He [57] developed a parallel algorithm for the maximum independent-set problem, which is a simpler version of the maximum marking problem where we ignore the value of the node.

Sasano et al. [116] developed a systematic method of deriving sequential programs from recursive specifications of the predicates. In this chapter, we extend it to the derivation of parallel programs. The finiteness property and the tupled-ring property in Section 5.2 play an important role in the derivation of parallel programs. We furthermore discuss the optimization of derived parallel programs.

This chapter is organized as follows. In Section 8.1, we define two classes of the maximum marking problems targeted in this chapter. Before studying the derivation of parallel programs, we review Sasano et al.'s method of deriving sequential programs in Section 8.2. In Section 8.3, we develop a derivation algorithm for the maximum weight-sum problems, subproblems of the maximum marking problems, and in Section 8.4, we develop a derivation algorithm for the maximum marking problems. In Section 8.5, we discuss the optimization of derived programs. The usefulness of the derivation method is demonstrated with two non trivial instances of the maximum marking problems in Section 8.6. Finally, Section 8.7 summarizes this chapter.

8.1 Specification of Maximum Marking Problems

We first specify the target classes of the maximum marking problems for which we derive parallel programs in this chapter. As we see in Section 8.6, even under the following conditions there are still many interesting problems.

1. The data structure x of the input is a binary tree in which nodes have a number of type Num. If the input has another type, we should apply a certain preprocess.

2. In terms of the marked data structure, each node is marked with a boolean value of type Bool.

3. The predicate p is given as composition of tree homomorphism $(\!|p_l, p_n|\!)_{\mathrm{b}}$ and a function *accept*. Here, the tree homomorphism may return a tuple of boolean values. In other words, the predicate should be written as follows where n is a finite number.

$$
\begin{aligned}
&p = \mathit{accept} \circ (\!|p_l, p_n|\!)_{\mathrm{b}} \\
&\quad \textbf{where } \mathit{accept} :: \mathsf{Bool}^n \to \mathsf{Bool} \\
&\qquad\qquad (\!|p_l, p_n|\!)_{\mathrm{b}} :: \mathsf{BTree\ Bool\ Bool} \ \to \ \mathsf{Bool}^n
\end{aligned}
$$

The second condition on the marked data structure and the third condition on the predicate are also assumed in the derivation algorithm developed by Sasano et al. [116]. Therefore, the additional condition for the derivation of parallel programs in this chapter is the binary-tree structure for the input.

Tree homomorphisms returning a tupled value are often given from mutual recursive functions over the tree structure. Such mutual recursive functions are called mutumorphisms [44, 67] and it is known that mutumorphisms can be turned into a single homomorphism by the tupling transformation [44, 62]. Based on the mutumorphisms and the

tupling transformation, we can specify the maximum marking problems more easily by using mutual recursive functions instead of a homomorphism returning a tupled values.

We give a formal specification of the maximum marking problems and their subproblems called the maximum weight-sum problems.

Definition 8.1 (Target Maximum Marking Problem) Let p be a given predicate in the form of $p = \mathit{accept} \circ (\!|p_l, p_n|\!)_\mathrm{b}$, where for a finite number n the function accept has type $\mathit{accept} :: \mathsf{Bool}^n \to \mathsf{Bool}$ and the tree homomorphism $(\!|p_l, p_n|\!)_\mathrm{b}$ has type $(\!|p_l, p_n|\!)_\mathrm{b} :: \mathsf{BTree\ Bool\ Bool} \to \mathsf{Bool}^n$. For a given binary tree of type $\mathsf{BTree\ Num\ Num}$, the maximum marking problem targeted in this paper is to mark the nodes by either True or False in such a way that the marked tree satisfies the predicate and the sum of values on nodes marked as True is as large as possible.

We denote a maximum marking problem defined with predicate p as $\mathit{mmp}\ p$. □

Definition 8.2 (Target Maximum Weight-Sum Problem) Let p be a given predicate in the form of $p = \mathit{accept} \circ (\!|p_l, p_n|\!)_\mathrm{b}$, where for a finite number n the function accept has type $\mathit{accept} :: \mathsf{Bool}^n \to \mathsf{Bool}$ and the tree homomorphism $(\!|p_l, p_n|\!)_\mathrm{b}$ has type $(\!|p_l, p_n|\!)_\mathrm{b} :: \mathsf{BTree\ Bool\ Bool} \to \mathsf{Bool}^n$. For a given binary tree of type $\mathsf{BTree\ Num\ Num}$, the maximum weight-sum problem targeted in this paper is to compute the maximum among the sums of values on nodes marked as True under the condition that the marked tree satisfies the predicate p.

We denote a maximum weight-sum problem defined with predicate p as $\mathit{mws}\ p$. □

We introduce two instances of the maximum marking problems: computing the height of binary trees (in Section 2.2) and the party planning problem (in Sections 1.2 and 5.3). We will use these two instances as our running examples in deriving skeletal parallel programs.

We can specify computing the height of binary trees as a maximum weight-sum problem. After assigning one to each node, we obtain the height of a tree by the length of the longest path from the root to a leaf. We define the following function path that validates whether or not marking of nodes forms a path. Auxiliary function nomark returns True if there is no node marked as True.

$$
\begin{aligned}
&\mathit{path}\ (\mathsf{BLeaf}\ a) &&= a \\
&\mathit{path}\ (\mathsf{BNode}\ l\ b\ r) &&= b \wedge ((\mathit{path}\ l \wedge \mathit{nomark}\ r) \vee (\mathit{nomark}\ l \wedge \mathit{path}\ r))
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{nomark}\ (\mathsf{BLeaf}\ a) &&= \neg a \\
&\mathit{nomark}\ (\mathsf{BNode}\ l\ b\ r) &&= \neg b \wedge \mathit{nomark}\ l \wedge \mathit{nomark}\ r
\end{aligned}
$$

Tupling these two functions, that is,

$$(\!|p_l, p_n|\!)_\mathrm{b}\ x = (\mathit{path} \vartriangle \mathit{nomark})\ x = (\mathit{path}\ x, \mathit{nomark}\ x)\ ,$$

we can specify computing the height of binary trees as a maximum weight-sum problem as follows.

$$\begin{aligned} height_{\mathrm{b}} &= mws\ (fst \circ (\!|p_l, p_n|\!)_{\mathrm{b}}) \circ map\ (\lambda a.1, \lambda b.1) \\ &\quad \textbf{where}\ p_l\ a = (a, \neg a) \\ &\qquad p_n\ (l_p, l_n)\ b\ (r_p, r_n) = (b \wedge ((l_p \wedge r_n) \vee (l_n \wedge r_p)), \neg b \wedge l_n \wedge r_n) \end{aligned}$$

The party planning problem is a maximum marking problem. We define function *independent* that returns True if no two nodes marked as True are adjacent, and auxiliary function $root_{\mathrm{b}}$.

$$\begin{aligned} &independent\ (\mathsf{BLeaf}\ a) = \mathsf{True} \\ &independent\ (\mathsf{BNode}\ l\ b\ r) = independent\ l \wedge independent\ r \\ &\qquad \wedge (\neg b \vee (\neg(root_{\mathrm{b}}\ l) \wedge \neg(root_{\mathrm{b}}\ r))) \end{aligned}$$

$$\begin{aligned} &root_{\mathrm{b}}\ (\mathsf{BLeaf}\ a) = a \\ &root_{\mathrm{b}}\ (\mathsf{BNode}\ l\ b\ r) = b \end{aligned}$$

Tupling these two functions,

$$(\!|p_l, p_n|\!)_{\mathrm{b}}\ x = (independent \vartriangle root_{\mathrm{b}})\ x = (independent\ x, root_{\mathrm{b}}\ x)\ ,$$

we can specify the party planning problem as a maximum marking problem as follows.

$$\begin{aligned} ppp_{\mathrm{b}} &= mmp\ (fst \circ (\!|p_l, p_n|\!)_{\mathrm{b}}) \\ &\quad \textbf{where}\ p_l\ a = (\mathsf{True}, a) \\ &\qquad p_n\ (l_i, l_r)\ b\ (r_i, r_r) = (l_i \wedge r_i \wedge (\neg b \vee (\neg l_r \wedge \neg r_r)), b) \end{aligned}$$

8.2 Review: Sasano et al's derivation Algorithm

A straightforward but inefficient solution to the maximum marking problems is with the so-called generate-and-test approach. We first enumerate all the ways of markings on the input tree (*genMark*), then compute the sums of elements marked as True (*sumMark*) for the markings that satisfy the predicate p, and return a marking with the maximum sum (*maxMark*).

$$mmp\ x = maxMark\ [sumMark\ x' \mid x' \leftarrow genMark\ x,\ p\ x]$$

This program is inefficient since the function *genMark* generates an exponential number of markings.

It was know that the maximum marking problem can be solved efficiently under certain conditions [14,24]. Sasano et al. [116] developed a systematic method of deriving practical linear-time programs from the specification of the predicate. Sasano et al.'s derivation assumes the second and third conditions for the marks and the predicate, but any data structure that can be written by constructors with finite arguments is allowed. In this section, we review the derivation method.

Let us call a value of type Bool^n as a *state* (from automata). By the principle of optimality, we only need for each state a marking corresponding to the maximum sum in

the computation of the maximum marking problems. The main ideas in the derivation of a sequential program are: to map a homomorphism in the predicate to another homomorphism that solves the maximum marking problem, and to pick up the case corresponding to the maximum sum for each state at each iterative step.

Sasano et al.'s derivation method generates the following two functions k_l and k_n. These functions return a list of triples (c, w, t), where c is a state, w is weight sum, and t is a marked tree. In the following definition, function *eachmax* takes a list of triples and filters the maximum one in terms of the weight sum (the second value of the triple) for each state (the first value of the triple).

$$k_l\ a = eachmax\ [(p_l\ m, \mathbf{if}\ m\ \mathbf{then}\ a\ \mathbf{else}\ 0, \mathsf{BLeaf}\ m) \mid m \leftarrow [\mathsf{True}, \mathsf{False}]]$$

$$\begin{aligned} k_n\ l\ b\ r = eachmax\ [&(p_n\ c_l\ m\ c_r, w_l + (\mathbf{if}\ m\ \mathbf{then}\ b\ \mathbf{else}\ 0) + w_r, \mathsf{BNode}\ t_l\ m\ t_r) \\ &\mid m \leftarrow [\mathsf{True}, \mathsf{False}], (c_l, w_l, t_l) \leftarrow l, (c_r, w_r, t_r) \leftarrow r] \end{aligned}$$

Using these functions we can obtain the following sequential program that solves simultaneously the maximum weight-sum problem and the maximum marking problem. The result of the maximum weight-sum problem is given as the first value of the resulting pair, and the result of the maximum marking problem is given as the second value of the resulting pair. Operator $\uparrow_{fst}$ returns the tuple whose first value is the larger of the two.

$$\begin{aligned} &(mws\ (accept \circ (\!|p_l, p_n|\!)_{\mathrm{b}}) \vartriangle mmp\ (accept \circ (\!|p_l, p_n|\!)_{\mathrm{b}}))\ x \\ &\quad = \textstyle\sum_{\uparrow_{fst}} [(w, t) \mid (c, w, t) \leftarrow (\!|k_l, k_n|\!)_{\mathrm{b}}\ x, accept\ c] \\ &\qquad \mathbf{where}\ k_l\ a = eachmax\ [(p_l\ m, \mathbf{if}\ m\ \mathbf{then}\ a\ \mathbf{else}\ 0, \mathsf{BLeaf}\ m) \\ &\qquad\qquad\qquad\qquad\qquad\qquad \mid m \leftarrow [\mathsf{True}, \mathsf{False}]] \\ &\qquad\qquad k_n\ l\ b\ r = eachmax\ [(p_n\ c_l\ m\ c_r, \\ &\qquad\qquad\qquad\qquad\qquad\qquad w_l + (\mathbf{if}\ m\ \mathbf{then}\ b\ \mathbf{else}\ 0) + w_r, \\ &\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{BNode}\ t_l\ m\ t_r) \\ &\qquad\qquad\qquad\qquad\qquad\qquad \mid m \leftarrow [\mathsf{True}, \mathsf{False}], \\ &\qquad\qquad\qquad\qquad\qquad\qquad\ (c_l, w_l, t_l) \leftarrow l, (c_r, w_r, t_r) \leftarrow r] \end{aligned}$$

We now demonstrate the derivation of sequential programs with the two examples.

For computing the height of binary trees, we only need the maximum weight sum, and thus we omit the third value of the triples. By substituting the definition of the predicate, we obtain the following sequential program that computes the height of binary trees. In the following program, function $eachmax'$ takes a list of pairs (c, w) and filters a pair for each state c that has the maximum weight sum w.

$$\begin{aligned} &height_{\mathrm{b}}\ x \\ &\quad = \textstyle\sum_{\uparrow} [w \mid (c, w) \leftarrow (\!|k_l, k_n|\!)_{\mathrm{b}}\ (\mathsf{map}_{\mathrm{b}}\ (\lambda x.1)\ (\lambda x.1)\ x), fst\ c] \\ &\qquad \mathbf{where}\ k_l\ a = [((\mathsf{True}, \mathsf{False}), a), ((\mathsf{False}, \mathsf{True}), 0)] \\ &\qquad\qquad k_n\ l\ b\ r = eachmax'\ [(p_n\ c_l\ m\ c_r, w_l + (\mathbf{if}\ m\ \mathbf{then}\ b\ \mathbf{else}\ 0) + w_r) \\ &\qquad\qquad\qquad\qquad\qquad\qquad \mid m \leftarrow [\mathsf{True}, \mathsf{False}], (c_l, w_l) \leftarrow l, (c_r, w_r) \leftarrow r] \\ &\qquad\qquad p_n\ (l_p, l_n)\ b\ (r_p, r_n) = (b \wedge ((l_p \wedge r_n) \vee (l_n \wedge r_p)), \neg\ b \wedge l_n \wedge r_n) \end{aligned}$$

Similarly, we can derive a sequential program for the party planning problem just by substituting the predicate.

$$
\begin{aligned}
ppp_{\mathrm{b}}\ x = snd\ (\textstyle\sum_{\uparrow_{fst}}[(w,t) \mid (c,w,t) \leftarrow (\!|k_l, k_n|\!)_{\mathrm{b}}\ x, fst\ c] \\
\textbf{where}\ k_l\ a = [((\mathsf{True},\mathsf{True}), a, \mathsf{BLeaf\ True}), ((\mathsf{True},\mathsf{False}), 0, \mathsf{BLeaf\ False})] \\
k_n\ l\ b\ r = eachmax\ [(p_n\ c_l\ m\ c_r, w_l + (\textbf{if}\ m\ \textbf{then}\ b\ \textbf{else}\ 0) + w_r, \\
\qquad \mathsf{BNode}\ t_l\ m\ t_r) \\
\qquad \mid m \leftarrow [\mathsf{True},\mathsf{False}], \\
\qquad\ (c_l, w_l, t_l) \leftarrow l, (c_r, w_r, t_r) \leftarrow r] \\
p_n\ (l_i, l_r)\ b\ (r_i, r_r) = (l_i \wedge r_i \wedge (\neg\ b \vee (\neg\ l_r \wedge \neg\ r_r)), b)
\end{aligned}
$$

8.3 Deriving Parallel Programs for Maximum Weight-Sum Problems

Based on the derivation of sequential programs in the previous section, in this section we develop a derivation method of parallel programs for the maximum weight-sum problems.

The tree homomorphism $(\!|k_l, k_n|\!)_{\mathrm{b}}$ in the derived sequential programs returns a list of triples. Here, the following two facts are worth remarking. First, the number of the elements in the resulting list is at most 2^n where n is the number of functions tupled in the predicate. Second, the result of the predicate depends only on marks and not on the values of nodes. Taking these facts in mind, we rewrite the sequential program in the previous section into one in which the tree homomorphism returns a tuple of 2^n elements.

To achieve this rewriting, we first define an indexing of the states. We here number the states using the following functions. Function $bf2id_n$ (named from "bit-field to ID") takes a tuple of n boolean values and returns an integer given by reading the tuple as a binary number. Function $id2bf_n$ (named from "ID to bit-field") is the inverse function of $bf2id_n$, that is, the function $id2bf_n$ takes an integer and returns a tuple of n boolean values denoting the number in the binary system. The following two examples illustrate these two functions. Here, note that ten is 1010 in the binary system.

$$
\begin{aligned}
&bf2id_5\ (\mathsf{False},\mathsf{True},\mathsf{False},\mathsf{True},\mathsf{False}) = 10 \\
&id2bf_5\ 10 = (\mathsf{False},\mathsf{True},\mathsf{False},\mathsf{True},\mathsf{False})
\end{aligned}
$$

Based on this indexing of the states, we introduce three functions $accept'$, p'_l and p'_n that are used instead of those in the predicate.

$$
\begin{aligned}
&accept' :: \mathsf{Int} \to \mathsf{Bool} \\
&accept'\ a = accept\ (id2bf_n\ a)
\end{aligned}
$$

$$
\begin{aligned}
&(\!|p'_l, p'_n|\!)_{\mathrm{b}} :: \mathsf{BTree\ Bool\ Bool} \to \mathsf{Int} \\
&p'_l\ a = bf2id_n\ (p_l\ a) \\
&p'_n\ l\ b\ r = bf2id_n\ (p_n\ (id2bf_n\ l)\ b\ (id2bf_n\ r))
\end{aligned}
$$

Using these functions, we can remove the *eachmax* function by replacing the result list of the tree homomorphism with a tuple of 2^n elements (in the following, we denote

$2^n = s$). The following sequential program solves the maximum weight-sum problem. We use the notation of list comprehension for tuples for readability.

$$
\begin{array}{l}
mws\ (accept \circ (\!|p_l, p_n|\!)_{\mathrm{b}})\ x \\
\quad = \sum\nolimits_{\uparrow}[w_i \mid (w_0, w_1, \ldots, w_{s-1}) = (\!|k'_l, k'_n|\!)_{\mathrm{b}}\ x, i \leftarrow [0..s{-}1], accept'\ i] \\
\quad\ \mathbf{where} \\
\quad\ k'_l\ a = (w_0, w_1, \ldots, w_{s-1}) \\
\qquad\qquad \mathbf{where}\ w_i = \begin{cases} a \uparrow 0 & \mathbf{if}\ p'_l\ \mathsf{True} == i \wedge p'_l\ \mathsf{False} == i \\ a & \mathbf{if}\ p'_l\ \mathsf{True} == i \wedge p'_l\ \mathsf{False} \neq i \\ 0 & \mathbf{if}\ p'_l\ \mathsf{True} \neq i \quad \wedge p'_l\ \mathsf{False} == i \\ -\infty & \mathbf{otherwise} \end{cases} \\
\quad\ k'_n\ (l_0, l_1, \ldots, l_{s-1})\ b\ (r_0, r_1, \ldots, r_{s-1}) = (w_0, w_1, \ldots, w_{s-1}) \\
\qquad\qquad \mathbf{where}\ w_i = \sum\nolimits_{\uparrow}[l_j + (\mathbf{if}\ m\ \mathbf{then}\ b\ \mathbf{else}\ 0) + r_k \\
\qquad\qquad\qquad\qquad \mid j, k \leftarrow [0..s{-}1], m \leftarrow [\mathsf{True}, \mathsf{False}], p'_n\ j\ m\ k == i]
\end{array}
$$

Now, we return to the example. For the computation of the height of binary trees, since the predicate is defined with a pair of functions ($path \vartriangle nomark$), we number the states as

$$(\mathsf{False}, \mathsf{False}) = 0,\ (\mathsf{False}, \mathsf{True}) = 1,\ (\mathsf{True}, \mathsf{False}) = 2, \text{ and } (\mathsf{True}, \mathsf{True}) = 3.$$

Under this numbering, the functions $accept'$ and p'_l are defined as

$$accept'\ x = \begin{cases} \mathsf{True} & \mathbf{if}\ i == 2 \vee i == 3 \\ \mathsf{False} & \mathbf{otherwise} \end{cases}$$

$$
\begin{array}{l}
p'_l\ \mathsf{True} = 2 \\
p'_l\ \mathsf{False} = 1\ ,
\end{array}
$$

and the function p'_n is defined by the following tables.

$p'_n\ l\ \mathsf{True}$	r	0	1	2	3
	0	0	0	0	0
	1	0	2	2	2
l	2	0	2	0	2
	3	0	2	2	2

$p'_n\ l\ \mathsf{False}$	r	0	1	2	3
	0	0	0	0	0
	1	0	1	0	1
l	2	0	0	0	0
	3	0	1	0	1

Substituting the parameter functions and simplifying the program, we obtain the following sequential program.

$$
\begin{array}{l}
height_{\mathrm{b}}\ x = \mathbf{let}\ (w_0, w_1, w_2, w_3) = (\!|k'_l, k'_n|\!)_{\mathrm{b}}\ (\mathsf{map}_{\mathrm{b}}\ (\lambda x.1)\ (\lambda x.1)\ x)\ \mathbf{in}\ w_2 \uparrow w_3 \\
\qquad\ \mathbf{where}\ k'_l\ a = (-\infty, 0, a, -\infty) \\
\qquad\qquad k'_n\ (l_0, l_1, l_2, l_3)\ b\ (r_0, r_1, r_2, r_3) = (w_0, w_1, w_2, w_3) \\
\qquad\qquad\ \mathbf{where}\ w_i = \sum\nolimits_{\uparrow}[l_j + (\mathbf{if}\ m\ \mathbf{then}\ b\ \mathbf{else}\ 0) + r_k \\
\qquad\qquad\qquad\qquad\qquad \mid j, k \leftarrow [0, 1, 2, 3], m \leftarrow [\mathsf{True}, \mathsf{False}], \\
\qquad\qquad\qquad\qquad\qquad p'_n\ j\ m\ k == i]
\end{array}
$$

We then parallelize the derived sequential program using the tupled-ring property in Section 5.2. It follows from the following two facts that the function k'_n satisfies the tupled-ring property. First, two operators $+$ and $\uparrow$ form a commutative semi-ring $\{\mathsf{Num}, \uparrow, +\}$. Second, we can rewrite the computation of the ith element w_i in the function k'_n as follows.

$$\begin{aligned} w_i &= \textstyle\sum_{\uparrow}[l_j + (\textbf{if } m \textbf{ then } b \textbf{ else } 0) + r_k \\ &\qquad \mid j, k \leftarrow [0..s{-}1], m \leftarrow [\mathsf{True}, \mathsf{False}], p'_n\ j\ m\ k == i] \\ &= \textstyle\sum_{\uparrow}[l_j + r_k + b_{jk} \mid j, k \leftarrow [0..s{-}1]] \\ &\qquad \textbf{where } b_{jk} = \begin{cases} b \uparrow 0 & \textbf{if } p'_n\ j\ \mathsf{True}\ k == i \wedge p'_n\ j\ \mathsf{False}\ k == i \\ b & \textbf{if } p'_n\ j\ \mathsf{True}\ k == i \wedge p'_n\ j\ \mathsf{False}\ k \neq i \\ 0 & \textbf{if } p'_n\ j\ \mathsf{True}\ k \neq i \wedge p'_n\ j\ \mathsf{False}\ k == i \\ -\infty & \textbf{otherwise} \end{cases} \end{aligned}$$

From the program rewritten above, we can easily prove that the function can be defined as the two bi-linear polynomial functions.

$$\begin{aligned} &\textstyle\sum_{\uparrow}[l_j + r_k + b_{jk} \mid j, k \leftarrow [0..s{-}1]] \\ &\quad = \textstyle\sum_{\uparrow}[l_j + \sum_{\uparrow}[r_k + b_{jk} \mid k \leftarrow [0..s{-}1]] \mid j \leftarrow [0..s{-}1]] \end{aligned}$$

$$\begin{aligned} &\textstyle\sum_{\uparrow}[l_j + r_k + b_{jk} \mid j, k \leftarrow [0..s{-}1]] \\ &\quad = \textstyle\sum_{\uparrow}[r_k + \sum_{\uparrow}[l_j + b_{jk} \mid j \leftarrow [0..s{-}1]] \mid k \leftarrow [0..s{-}1]] \end{aligned}$$

From the calculations above, we can apply Theorem 5.12 to derive a parallel program for the maximum weight-sum problems.

Theorem 8.1 *There exists a parallel program that solves the target maximum weight-sum problem given in Definition 8.2.*

Proof. First, from the derivations above a skeletal program for the maximum weight-sum problem is given as follows.

$$\begin{aligned} &mws\ (accept \circ (\!|p_l, p_n|\!)_{\mathrm{b}})\ x \\ &\quad = \textstyle\sum_{\uparrow}[w_i \mid (w_0, w_1, \ldots, w_{s-1}) = \mathsf{reduce}_{\mathrm{b}}\ k'_n\ (\mathsf{map}_{\mathrm{b}}\ k'_l\ id\ x), \\ &\qquad\qquad i \leftarrow [0..s{-}1], accept'\ i] \\ &\textbf{where} \\ &\quad k'_l\ a = (w_0, w_1, \ldots, w_{s-1}) \\ &\qquad \textbf{where } w_i = \begin{cases} a \uparrow 0 & \textbf{if } p'_l\ \mathsf{True} == i \wedge p'_l\ \mathsf{False} == i \\ a & \textbf{if } p'_l\ \mathsf{True} == i \wedge p'_l\ \mathsf{False} \neq i \\ 0 & \textbf{if } p'_l\ \mathsf{True} \neq i \quad \wedge p'_l\ \mathsf{False} == i \\ -\infty & \textbf{otherwise} \end{cases} \\ &\quad k'_n\ (l_0, l_1, \ldots, l_{s-1})\ b\ (r_0, r_1, \ldots, r_{s-1}) = (w_0, w_1, \ldots, w_{s-1}) \\ &\qquad \textbf{where } w_i = \textstyle\sum_{\uparrow}[l_j + b'_{ijk}\ b + r_k \mid j, k \in [0..s-1]] \\ &\quad accept'\ a = accept\ (id2bf_n\ a) \\ &\quad p'_l\ a = bf2id_n\ (p_l\ a) \\ &\quad p'_n\ l\ b\ r = bf2id_n\ (p_n\ (id2bf_n\ l)\ b\ (id2bf_n\ r)) \\ &\qquad b'_{ijk}\ b = \begin{cases} b \uparrow 0 & \textbf{if } p'_n\ j\ \mathsf{True}\ k == i \wedge p'_n\ j\ \mathsf{False}\ k == i \\ b & \textbf{if } p'_n\ j\ \mathsf{True}\ k == i \wedge p'_n\ j\ \mathsf{False}\ k \neq i \\ 0 & \textbf{if } p'_n\ j\ \mathsf{True}\ k \neq i \wedge p'_n\ j\ \mathsf{False}\ k == i \\ -\infty & \textbf{otherwise} \end{cases} \end{aligned}$$

The functions $bf2id_n$ and $id2bf_n$ were defined at the start of this section. To make the derived skeletal program runs in parallel, we need to derive auxiliary functions for the skeleton $\mathsf{reduce}_\mathrm{b}$ called with k'_n. Here, from the Theorem 5.12, we can derive auxiliary functions ϕ, ψ_n, ψ_l, and ψ_r satisfying $k'_n = \langle \phi, \psi_n, \psi_l, \psi_r \rangle_\mathrm{u}$ as follows. Note that in the following definition, the matrices are of size $s \times s$. We can remove the $(s+1)$th column and the $(s+1)$th row since they are equal to those in the identity matrix.

$$
\begin{aligned}
&\phi\ b = (b, \boldsymbol{I}) \\
&\psi_n\ \boldsymbol{l}\ (b_n, \boldsymbol{M}_n)\ \boldsymbol{r} = \boldsymbol{M}_n \times_{+,\uparrow} k'_n\ \boldsymbol{l}\ b_n\ \boldsymbol{r} \\
&\psi_l\ (b_l, \boldsymbol{M}_l)\ (b_n, \boldsymbol{M}_n)\ (r_0, r_1, \ldots, r_{s-1}) = (b_l, \boldsymbol{M}_n \times_{+,\uparrow} \{r'_{ij}\ b\} \times_{+,\uparrow} \boldsymbol{M}_l) \\
&\qquad\qquad \textbf{where}\ r'_{ij}\ b = \textstyle\sum_\uparrow [r_k + b'_{ijk}\ b \mid k \leftarrow [0..s{-}1]] \\
&\psi_r\ (l_0, l_1, \ldots, l_{s-1})\ (b_n, \boldsymbol{M}_n)\ (r_n, \boldsymbol{M}_r) = (b_r, \boldsymbol{M}_n \times_{+,\uparrow} \{l'_{ik}\ b\} \times_{+,\uparrow} \boldsymbol{M}_r) \\
&\qquad\qquad \textbf{where}\ l'_{ik}\ b = \textstyle\sum_\uparrow [l_j + b'_{ijk}\ b \mid j \leftarrow [0..s{-}1]]
\end{aligned}
$$

It follows from the skeletal program and the auxiliary functions given above that the theorem holds. □

It is worth noting that the predicate is *static* against the values of nodes and therefore we can specialize the definition of k'_n and auxiliary functions for each instance of the maximum weight-sum problems. We will discuss the optimization based on this specialization in Section 8.5.

We show a parallel program for the computation of height of binary trees in Figure 8.1. We can derive the parallel program by straightforward substitution of parameters.

8.4 Deriving Parallel Programs for Maximum Marking Problems

In the sequential program derived by Sasano et al.'s method, the marked trees are generated in a bottom-up manner as well as the weight sums. Since the generation of trees is hard to parallelize by our tree skeletons, we decompose the sequential program into two tree accumulations.

A well-known technique for dynamic programming generates marked structures by two-pass algorithms. We first compute the maximum weight sum storing intermediate maximal weight sums at each element, and then mark elements by traversing the stored values in the reversed order. For the maximum marking problems on trees, we store intermediate maximal weight sums at each node by an upwards accumulation and then mark nodes by traversing the tree by a downwards accumulation.

Here, we should pay some attentions to the implementation of marking nodes by the downwards accumulation. An internal node of a binary tree has two children, and in general the marks on these two children depend on each other as well as the parent mark. For example, when we mark a path from the root to a leaf, we should mark either of the two children if the parent node is marked. To cope with these dependency using the

$$
\begin{array}{l}
height_{\mathrm{b}} = \textbf{let } (w_0, w_1, w_2, w_3) = \mathsf{reduce}_{\mathrm{b}}\ k'_n\ (\mathsf{map}_{\mathrm{b}}\ k'_l\ (\lambda x.1)\ x) \\
\qquad\qquad \textbf{in } w_2 \uparrow w_3 \\
\textbf{where} \\
\quad k'_l\ a = (-\infty, 0, 1, -\infty) \\
\quad k'_n\ (l_0, l_1, l_2, l_3)\ b\ (r_0, r_1, r_2, r_3) = (w_0, w_1, w_2, w_3) \\
\qquad\quad \textbf{where } w_i = \sum\nolimits_{\uparrow} [l_j + b'_{ijk}\ b + r_k \mid j, k \leftarrow [0, 1, 2, 3]] \\
\quad \phi\ b = (b, \boldsymbol{I}) \\
\quad \psi_n\ \boldsymbol{l}\ (b_n, \boldsymbol{M}_n)\ \boldsymbol{r} = \boldsymbol{M}_n \times_{+,\uparrow} k'_n\ \boldsymbol{l}\ b_n\ \boldsymbol{r} \\
\quad \psi_l\ (b_l, \boldsymbol{M}_l)\ (b_n, \boldsymbol{M}_n)\ (r_0, r_1, r_2, r_3) = (b_l, \boldsymbol{M}_n \times_{+,\uparrow} \{r'_{ij}\ b_n\} \times_{+,\uparrow} \boldsymbol{M}_l) \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{where } r'_{ij}\ b_n = \sum\nolimits_{\uparrow} [r_k + b'_{ijk}\ b_n \mid k \leftarrow [0, 1, 2, 3]] \\
\quad \psi_r\ (l_0, l_1, l_2, l_3)\ (b_n, \boldsymbol{M}_n)\ (b_r, \boldsymbol{M}_r) = (b_r, \boldsymbol{M}_n \times_{+,\uparrow} \{l'_{ik}\ b_n\} \times_{+,\uparrow} \boldsymbol{M}_r) \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{where } l'_{ik}\ b_n = \sum\nolimits_{\uparrow} [l_j + b'_{ijk}\ b_n \mid j \leftarrow [0, 1, 2, 3]]
\end{array}
$$

$b'_{0jk}\ b$	k = 0	1	2	3
0	$b \uparrow 0$	$b \uparrow 0$	$b \uparrow 0$	$b \uparrow 0$
1	$b \uparrow 0$	$-\infty$	0	$-\infty$
j 2	$b \uparrow 0$	0	$b \uparrow 0$	0
3	$b \uparrow 0$	$-\infty$	0	$-\infty$

$b'_{1jk}\ b$	k = 0	1	2	3
0	$-\infty$	$-\infty$	$-\infty$	$-\infty$
1	$-\infty$	0	$-\infty$	0
j 2	$-\infty$	$-\infty$	$-\infty$	$-\infty$
3	$-\infty$	0	$-\infty$	0

$b'_{2jk}\ b$	k = 0	1	2	3
0	$-\infty$	$-\infty$	$-\infty$	$-\infty$
1	$-\infty$	b	b	b
j 2	$-\infty$	b	$-\infty$	b
3	$-\infty$	b	b	b

$b'_{3jk}\ b$	k = 0	1	2	3
0	$-\infty$	$-\infty$	$-\infty$	$-\infty$
1	$-\infty$	$-\infty$	$-\infty$	$-\infty$
j 2	$-\infty$	$-\infty$	$-\infty$	$-\infty$
3	$-\infty$	$-\infty$	$-\infty$	$-\infty$

Figure 8.1. Derived skeletal parallel program for computing the height of binary trees. In this program, two sequential $\mathsf{map}_{\mathrm{b}}$ skeletons are fused into one.

downwards accumulation, we decide at the parent node whether two children are marked or not, and we need to attach the maximal weight sums of the children to their parent.

Based on these ideas, we develop parallel programs that solve the maximum marking problems in the following four steps.

1. Compute maximal weight sums for each node in a bottom-up manner.
2. Attach the maximal weight sums for each node.
3. Compute a state for each node in a top-down manner.
4. Decide the mark for each node from the state and maximal weight sums.

In the following discussion, let t be the input tree, and $accept'$, p'_l, p'_n, k'_l and k'_n be functions given in the previous section.

In the first step, we compute maximal weight sums for each node by the map_b and uAcc_b skeletons instead of the map_b and $\mathsf{reduce}_\mathrm{b}$ skeletons.

$$bt = \mathsf{uAcc}_\mathrm{b}\ k'_n\ (\mathsf{map}_\mathrm{b}\ k'_l\ id\ t)$$

The auxiliary functions for the uAcc_b skeleton are those given for the $\mathsf{reduce}_\mathrm{b}$ skeleton in the previous section.

In the second step, we attach the maximal weight sums to the original value for each node.

$$zt = \mathsf{zip4}_\mathrm{b}\ t\ bt\ (\mathsf{getchl}_\mathrm{b}\ \_\ bt)\ (\mathsf{getchr}_\mathrm{b}\ \_\ bt)$$

Each node in the tree zt has a tuple in the form (b, ms, ls, rs), where b and ms are the original value and the maximal weight sums of the nodes, ls is the maximal weight sums of the left child, and rs is the maximal weight sums of the right child.

In the third step, we apply the dAcc_b skeleton to the zipped tree with an accumulative parameter indicating a state of the node that yields the maximum weight sum at the root. The accumulative parameter for the root can be given from the maximal weight sums at the root as follows.

$$\begin{aligned} c = \mathbf{let}\ & m = \textstyle\sum_\uparrow [w_i \mid (w_0, w_1, \ldots, w_{s-1}) = root_\mathrm{b}\ bt, i \leftarrow [0..s{-}1], accept'\ i] \\ \mathbf{in}\ & fst\ [i \mid (w_0, w_1, \ldots, w_{s-1}) = root_\mathrm{b}\ bt, i \leftarrow [0..s{-}1], accept'\ i, m == w_i] \end{aligned}$$

If there are more than or equal to two states that have the maximum weight sum, we choose one of it (here, we use function fst for it). The two parameter functions g_l and g_r for the dAcc_b skeleton find states of two children where the states of two children should yield the maximum sum corresponding to the state passed as accumulating parameter c. In the following definition, w_c indicates the cth element of the tuple $(w_0, w_1, \ldots, w_{s-1})$.

$$\begin{aligned} ct = {} & \mathsf{dAcc}_\mathrm{b}\ (g_l, g_r)\ c\ zt \\ & \mathbf{where} \\ & \quad (g_l\ c \vartriangle g_r\ c)\ (b, (w_0, w_1, \ldots, w_{s-1}), (l_0, l_1, \ldots, l_{s-1}), (r_0, r_1, \ldots, r_{s-1})) \\ & \quad\quad = fst\ [(j,k) \mid (j,k) \leftarrow [0..s{-}1], m \leftarrow [\mathsf{True}, \mathsf{False}], \\ & \quad\quad\quad\quad w_c == l_i + (\mathbf{if}\ m\ \mathbf{then}\ b\ \mathbf{else}\ 0) + r_k, p'_n\ j\ m\ k == c] \end{aligned}$$

Finally in the fourth step, we compute the resulting mark for each node based on the state computed in the previous step.

$$
\begin{array}{l}
\mathsf{zipwith_b}\ k_l''\ k_n''\ ct\ zt \\
\textbf{where} \\
\quad k_l''\ c\ (a, (w_0, w_1, \ldots, w_{s-1}), \_, \_) = (w_c == a \wedge p_l'\ \mathsf{True} == c) \\
\quad k_n''\ c\ (b, (w_0, w_1, \ldots, w_{s-1}), (l_0, l_1, \ldots, l_{s-1}), (r_0, r_1, \ldots, r_{s-1})) \\
\qquad = fst\ [m \mid (j,k) \leftarrow [0..s{-}1], m \leftarrow [\mathsf{True}, \mathsf{False}], \\
\qquad\qquad w_c == l_i + (\textbf{if}\ m\ \textbf{then}\ b\ \textbf{else}\ 0) + r_k, p_n'\ j\ m\ k == c]
\end{array}
$$

Note that the functions g_l, g_r, and k_n'' should enumerate the cases in the same order for the correctness of the resulting marks. Otherwise, the dependency between children will be broken.

We can derive the auxiliary functions for the $\mathsf{uAcc_b}$ and $\mathsf{dAcc_b}$ skeletons. For the $\mathsf{uAcc_b}$ skeleton, we can use the auxiliary functions derived for the $\mathsf{reduce_b}$ skeleton in the previous section. For the $\mathsf{dAcc_b}$ skeleton, since the number of states is finite (i.e., s), we can derive the auxiliary functions based on the finiteness property using Theorem 5.9.

Theorem 8.2 *There exists a parallel program that solves the target maximum marking problems in Definition 8.1.*

Proof. By summarizing the discussion above, we can give a skeletal program that solves the maximum marking problem as follows.

$$
\begin{array}{l}
mmp\ (accept \circ ([\![p_l, p_n]\!])_\mathrm{b})\ t \\
\quad = \textbf{let}\ bt = \mathsf{uAcc_b}\ k_n'\ (\mathsf{map_b}\ k_l'\ id\ t) \\
\qquad\quad zt = \mathsf{zip4_b}\ t\ bt\ (\mathsf{getchl_b}\ \_\ bt)\ (\mathsf{getchr_b}\ \_\ bt) \\
\qquad\quad m = \textstyle\sum_{\uparrow} [w_i \mid (w_0, w_1, \ldots, w_{s-1}) = root_\mathrm{b}\ bt, i \leftarrow [0..s{-}1], accept'\ i] \\
\qquad\quad c\ = fst\ [i \mid (w_0, w_1, \ldots, w_{s-1}) = root_\mathrm{b}\ bt, i \leftarrow [0..s{-}1], accept'\ i, m == w_i] \\
\qquad\quad ct = \mathsf{dAcc_b}\ (g_l, g_r)\ c\ zt \\
\qquad \textbf{in}\ \mathsf{zipwith_b}\ k_l''\ k_n''\ ct\ zt
\end{array}
$$

In this skeletal program, functions k_l', k_n', and $accept'$ and their auxiliary functions p_l' and p_n' are defined in Theorem 8.1. The other functions g_l, g_r, k_l'', and k_n'' are defined as follows (the same as the definitions above).

$$
\begin{array}{l}
(g_l\ c \vartriangle g_r\ c)\ (b, (w_0, w_1, \ldots, w_{s-1}), (l_0, l_1, \ldots, l_{s-1}), (r_0, r_1, \ldots, r_{s-1})) \\
\quad = fst\ [(j,k) \mid j,k \leftarrow [0..s{-}1], m \leftarrow [\mathsf{True}, \mathsf{False}], \\
\qquad\qquad w_c == l_j + (\textbf{if}\ m\ \textbf{then}\ b\ \textbf{else}\ 0) + r_k, p_n'\ j\ m\ k == c] \\
k_l''\ c\ (a, (w_0, w_1, \ldots, w_{s-1}), \_, \_) = (w_c == a \wedge p_l'\ \mathsf{True} == c) \\
k_n''\ c\ (b, (w_0, w_1, \ldots, w_{s-1}), (l_0, l_1, \ldots, l_{s-1}), (r_0, r_1, \ldots, r_{s-1})) \\
\quad = fst\ [m \mid j,k \leftarrow [0..s{-}1], m \leftarrow [\mathsf{True}, \mathsf{False}], \\
\qquad\qquad w_c == l_j + (\textbf{if}\ m\ \textbf{then}\ b\ \textbf{else}\ 0) + r_k, p_n'\ j\ m\ k == c]
\end{array}
$$

To derive auxiliary functions of the $\mathsf{uAcc_b}$ skeleton, we apply Theorem 8.1. By substituting the functions g_l and g_r for those in Theorem 5.9, we here derive auxiliary functions

for the $\mathsf{dAcc_b}$ skeleton, ϕ_l, ϕ_r, ψ_u, and ψ_d satisfying $(g_l, g_r) = \langle \phi_l, \phi_r, \psi_u, \psi_d \rangle_{\mathrm{d}}$.

$$
\begin{array}{l}
(\phi_l \vartriangle \phi_r)\ (b, (w_0, w_1, \ldots, w_{s-1}), (l_0, l_1, \ldots, l_{s-1}), (r_0, r_1, \ldots, r_{s-1})) \\
\quad = ((j'_0, j'_1, \ldots, j'_{s-1}), (k'_0, k'_1, \ldots, k'_{s-1})) \\
\quad \textbf{where} \\
\qquad (j'_i, k'_i) = \mathit{fst}\ [(j,k) \mid j,k \leftarrow [0..s{-}1], m \leftarrow [\mathsf{True}, \mathsf{False}], \\
\qquad\qquad\qquad\qquad w_i == l_j + (\textbf{if}\ m\ \textbf{then}\ b\ \textbf{else}\ 0) + r_k, p'_n\ j\ m\ k == i] \\
\psi_u\ (n_0, n_1, \ldots, n_{s-1})\ (m_0, m_1, \ldots, m_{s-1}) = (p_0, p_1, \ldots, p_{s-1}) \\
\quad \textbf{where}\ p_i = \textbf{let}\ i' = m_i\ \textbf{in}\ n_{i'} \\
\psi_d\ c\ (n_0, n_1, \ldots, n_{s-1}) = n_c
\end{array}
$$

Note that since states are represented by an integer in our derivation we use the states as the indices of tuple.

It follows from the skeletal program and the auxiliary functions derived so far that the theorem holds. □

Figure 8.2 shows a skeletal parallel program for the party planning problem derived from the specification in Section 8.1. The program is much more complicated than that shown in Section 5.3.1 due to redundant computation caused by unnecessary states and due to the generic derivation in which the dependency among children is taken into account. In the following section, we develop an optimization method of removing unnecessary computation due to the unnecessary states.

8.5 Optimization of Derived Parallel Programs

Parallel programs derived in the previous sections are often less efficient than those developed by hand. The main reason is redundant computation in derived programs caused by a large number of states. For example, we derived a program for the party planning problem on binary trees with four states in the previous section, but we can solve the same problem with only two states as seen in Section 5.3.1. In this section, we propose an optimization procedure for derived parallel programs.

8.5.1 Overview of Optimization

Figure 8.3 depicts the procedure of optimization for derived skeletal parallel programs. The input S_I is a set of 2^n states $\{0, 1, \ldots, 2^n{-}1\}$, and the output is a set S' of states, state s_z corresponding to the constant zero (if it exists), and a matrix $\boldsymbol{A}$ of abstract values used for the implementation of auxiliary functions.

The optimization consists of the following three subroutines.

- Forward optimization: We remove unnecessary states whose corresponding value must not be the maximum weight sum.
- Backward optimization: We remove unnecessary states that generate no acceptable states.

$$
\begin{array}{l}
ppp_{\mathrm{b}}\ t = \mathbf{let}\ bt = \mathsf{uAcc}_{\mathrm{b}}\ k'_n\ (\mathsf{map}_{\mathrm{b}}\ k'_l\ id\ t) \\
\qquad\qquad zt = \mathsf{zip4}_{\mathrm{b}}\ t\ bt\ (\mathsf{getchl}_{\mathrm{b}}\ bt)\ (\mathsf{getchr}_{\mathrm{b}}\ bt) \\
\qquad\qquad (w_0, w_1, w_2, w_3) = root_{\mathrm{b}}\ bt \\
\qquad\qquad c\ = \mathbf{if}\ w_2 > w_3\ \mathbf{then}\ 2\ \mathbf{else}\ 3 \\
\qquad\qquad ct = \mathsf{dAcc}_{\mathrm{b}}\ (g_l, g_r)\ c\ zt \\
\qquad\ \mathbf{in}\ \mathsf{zipwith}_{\mathrm{b}}\ k''_l\ k''_n\ ct\ zt
\end{array}
$$

$$k'_l\ a = (-\infty, -\infty, 0, a)$$

$$
\begin{array}{l}
k'_n\ (l_0, l_1, l_2, l_3)\ b\ (r_0, r_1, r_2, r_3) = (w_0, w_1, w_2, w_3) \\
\qquad \mathbf{where}\ w_i = \sum\nolimits_{\uparrow}\ [l_j + b'_{ijk}\ b + r_k \mid j, k \in [0, 1, 2, 3]]
\end{array}
$$

$$
\begin{array}{l}
(g_l\ c \vartriangle g_r\ c)\ (b, (w_0, w_1, w_2, w_3), (l_0, l_1, l_2, l_3), (r_0, r_1, r_2, r_3)) \\
\qquad = fst\ [(j, k) \mid j, k \leftarrow [0, 1, 2, 3], m \leftarrow [\mathsf{True}, \mathsf{False}], \\
\qquad\qquad\qquad w_c == l_j + (\mathbf{if}\ m\ \mathbf{then}\ b\ \mathbf{else}\ 0) + r_k, p'_n\ j\ m\ k == c]
\end{array}
$$

$$k''_l\ c\ (a, (w_0, w_1, w_2, w_3), \_, \_) = (w_c == a \wedge c == 3)$$

$$
\begin{array}{l}
k''_n\ c\ (b, (w_0, w_1, w_2, w_3), (l_0, l_1, l_2, l_3), (r_0, r_1, r_2, r_3)) \\
\qquad = fst\ [m \mid j, k \leftarrow [0, 1, 2, 3], m \leftarrow [\mathsf{True}, \mathsf{False}], \\
\qquad\qquad\qquad w_c == l_j + (\mathbf{if}\ m\ \mathbf{then}\ b\ \mathbf{else}\ 0) + r_k, p'_n\ j\ m\ k == c]
\end{array}
$$

$$\phi\ b = (b, \boldsymbol{I})$$

$$\psi_n\ \boldsymbol{l}\ (b_n, \boldsymbol{M}_n)\ \boldsymbol{r} = \boldsymbol{M} \times_{+,\uparrow} k'_n\ \boldsymbol{l}\ b_n\ \boldsymbol{r}$$

$$
\begin{array}{l}
\psi_l\ (b_l, \boldsymbol{M}_l)\ (b_n, \boldsymbol{M}_n)\ (r_0, r_1, r_2, r_3) = (b_l, \boldsymbol{M}_n \times_{+,\uparrow} \{r'_{ij}\ b_n\} \times_{+,\uparrow} \boldsymbol{M}_l) \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{where}\ r'_{ij}\ b_n = \sum\nolimits_{\uparrow}[r_k + b'_{ijk}\ b_n \mid k \in [0, 1, 2, 3]]
\end{array}
$$

$$
\begin{array}{l}
\psi_r\ (l_0, l_1, l_2, l_3)\ (b_n, \boldsymbol{M}_n)\ (b_r, \boldsymbol{M}_r) = (b_r, \boldsymbol{M}_n \times_{+,\uparrow} \{l'_{ik}\ b_n\} \times_{+,\uparrow} \boldsymbol{M}_r) \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{where}\ l'_{ik}\ b_n = \sum\nolimits_{\uparrow}[l_j + b'_{ijk}\ b_n \mid j \in [0, 1, 2, 3]]
\end{array}
$$

$$
\begin{array}{l}
(\phi_l \vartriangle \phi_r)\ (b, (w_0, w_1, w_2, w_3), (l_0, l_1, l_2, l_3), (r_0, r_1, r_2, r_3)) \\
\qquad = ((j'_0, j'_1, j'_2, j'_3), (k'_0, k'_1, k'_2, k'_3)) \\
\qquad \mathbf{where} \\
\qquad (j'_i, k'_i) = fst\ [(j, k) \mid j, k \leftarrow [0, 1, 2, 3], m \leftarrow [\mathsf{True}, \mathsf{False}], \\
\qquad\qquad\qquad\qquad w_i == l_j + (\mathbf{if}\ m\ \mathbf{then}\ b\ \mathbf{else}\ 0) + r_k, p'_n\ j\ m\ k == i]
\end{array}
$$

$$
\begin{array}{l}
\psi_u\ (n_0, n_1, n_2, n_3)\ (m_0, m_1, m_2, m_3) = (p_0, p_1, p_2, p_3) \\
\quad \mathbf{where}\ p_i = \mathbf{let}\ i' = m_i\ \mathbf{in}\ n_{i'}
\end{array}
$$

$$\psi_d\ c\ (n_0, n_1, n_2, n_3) = n_c$$

$p'_n\ l\ \mathsf{True}\ r$	r = 0	1	2	3
0	1	1	1	1
1	1	1	1	1
l 2	1	1	3	1
3	1	1	1	1

$p'_n\ l\ \mathsf{False}\ r$	r = 0	1	2	3
0	0	0	0	0
1	0	0	0	0
l 2	0	0	2	2
3	0	0	2	2

$$
b'_{ijk}\ b = \begin{cases}
b \uparrow 0 & \mathbf{if}\ p'_n\ j\ \mathsf{True}\ k == i \wedge p'_n\ j\ \mathsf{False}\ k == i \\
b & \mathbf{if}\ p'_n\ j\ \mathsf{True}\ k == i \wedge p'_n\ j\ \mathsf{False}\ k \neq i \\
0 & \mathbf{if}\ p'_n\ j\ \mathsf{True}\ k \neq i \wedge p'_n\ j\ \mathsf{False}\ k == i \\
-\infty & \mathbf{otherwise}
\end{cases}
$$

Figure 8.2. Derived skeletal parallel program for the party planning problem. The auxiliary functions satisfy $k'_n = \langle \phi, \psi_n, \psi_l, \psi_r \rangle_{\mathrm{u}}$ and $(g_l, g_r) = \langle \phi_l, \phi_r, \psi_u, \psi_d \rangle_{\mathrm{d}}$.

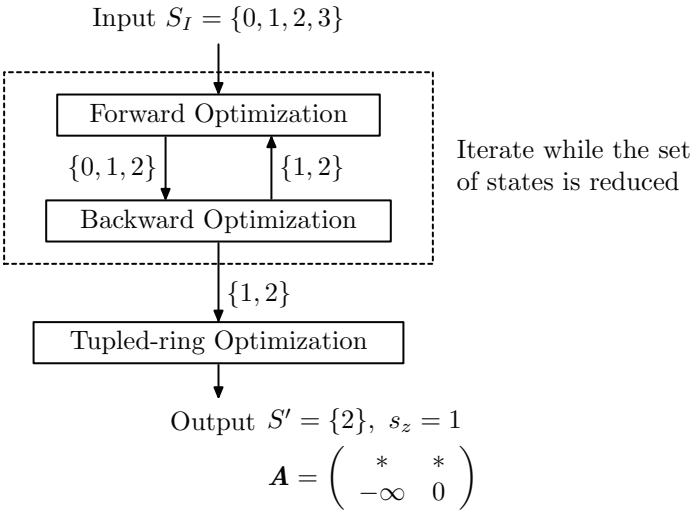


Figure 8.3. Procedure of optimization. The values are from the optimization for the example of computing the height of binary trees.

- Tupled-ring Optimization: We clarify necessary elements in the matrices used for auxiliary functions.

We apply the forward optimization and the backward optimization iteratively while the set of states is reduced.

It is worth noting that we can use the first two subroutines also for optimizing sequential programs whereas the last one is only for parallel programs. In the following, we show the three subroutines one by one.

8.5.2 Forward Optimization

The forward optimization simulates the computation of weight sums and removes unnecessary states that do not have any effect to the resulting maximum weight sum.

We perform the simulation using the following three abstract values.

- constant negative infinity, $-\infty$ (the unit of $\uparrow$),
- constant zero, 0 (the unit of $+$), and
- the other values or variables denoted by $*$.

We define three operators, $\uparrow$ for taking the larger, $+$ for addition, and $\odot$ for updating, as follows.

$\uparrow$	$-\infty$	0	$*$
$-\infty$	$-\infty$	0	$*$
0	0	0	$*$
$*$	$*$	$*$	$*$

$+$	$-\infty$	0	$*$
$-\infty$	$-\infty$	$-\infty$	$-\infty$
0	$-\infty$	0	$*$
$*$	$-\infty$	$*$	$*$

$\odot$	$-\infty$	0	$*$
$-\infty$	$-\infty$	$*$	$*$
0	$*$	0	$*$
$*$	$*$	$*$	$*$

These three operators are specialized ones for commutative semiring $\{\mathsf{Num}, \uparrow, +\}$ from those that are defined in Section 7.3 for optimizing program with tupled-ring property.

Using these operators, we can perform the forward optimization by the following three steps. The input is a set of states $S_I = \{s_0, s_1, \ldots, s_m\}$ and the output is a set of states $S_O = \{s'_0, s'_1, \ldots, s'_n\}$, where $S_I \supseteq S_O$ holds. In the following step, we use function p'_l and p'_n defined in the derivation of skeletal programs in Section 8.3.

1. Generate an initial tuple S_0 by computing an abstract value for each state.

$$S_0 = (w_{s_0}, w_{s_1}, \ldots, w_{s_m}) \qquad \textbf{where } w_{s_i} = \begin{cases} * & \textbf{if } p'_l\ \mathsf{True} == s_i \\ 0 & \textbf{else if } p'_l\ \mathsf{False} == s_i \\ -\infty & \textbf{otherwise} \end{cases}$$

2. Update the tuple by simulating the computation of weight sums until the values do not change.

$$\begin{aligned} S_{i+1} &= k'\ S_i\ +_\odot\ S_i \\ &\quad \textbf{where } k'\ (w_{s_0}, w_{s_1}, \ldots, w_{s_m}) = (w'_{s_0}, w'_{s_1}, \ldots, w'_{s_m}) \\ &\qquad w'_{s_i} = \textstyle\sum_\uparrow [w_{s_j} + (\textbf{if } m \textbf{ then } * \textbf{ else } 0) + w_{s_k} \\ &\qquad\qquad \mid s_j, s_k \leftarrow [s_0, s_1, \ldots, s_m], b \leftarrow [\mathsf{True}, \mathsf{False}], \\ &\qquad\qquad\ p'_n\ s_j\ m\ s_k == s_i] \\ &\qquad (w_{s_0}, w_{s_1}, \ldots, w_{s_m}) +_\odot (w'_{s_0}, w'_{s_1}, \ldots, w'_{s_m}) \\ &\qquad\quad = (w_{s_0} \odot w'_{s_0}, w_{s_1} \odot w'_{s_1}, \ldots, w_{s_m} \odot w'_{s_m}) \end{aligned}$$

3. Repeat step 2 until the tuple do not change, i.e., $S_{i+1} = S_i$ for some i. In the following, we denote S_∞ for such a tuple of states.

4. Remove states whose corresponding value in S_∞ is $-\infty$.

We illustrate the forward optimization using the example of computing the height of binary trees. Let the input set S_I of states be $S_I = \{0, 1, 2, 3\}$. Since the function p'_l for the computation of height is defined as $p'_l\ \mathsf{True} = 2$ and $p'_l\ \mathsf{False} = 1$, we have the initial tuple S_0 defined as follows.

$$S_0 = (-\infty, 0, *, -\infty)$$

We then update the tuple by simulating the computation of weight sums. The following calculations show the first two applications of step 2.

$$\begin{aligned} S_1 &= k'\ S_0\ +_\odot\ S_0 \\ &= k'\ (-\infty, 0, *, -\infty) +_\odot (-\infty, 0, *, -\infty) \\ &= (*, 0, *, -\infty) +_\odot (-\infty, 0, *, -\infty) \\ &= (*, 0, *, -\infty) \end{aligned}$$

$$\begin{aligned} S_2 &= k'\ S_1\ +_\odot\ S_1 \\ &= k'\ (*, 0, *, -\infty) +_\odot (*, 0, *, -\infty) \\ &= (*, 0, *, -\infty) +_\odot (*, 0, *, -\infty) \\ &= (*, 0, *, -\infty) \\ &= S_1 \end{aligned}$$

From these calculations, we have S_∞ as $S_\infty = S_1 = (*, 0, *, -\infty)$. By applying the fourth step, we can remove state 4 and obtain the reduced set of states $S_O = \{0, 1, 2\}$ as the output of the forward optimization.

8.5.3 Backward Optimization

The backward optimization removes states that affect nothing on the resulting maximum weight sum. This optimization is named *backward* since the optimization starts from the acceptable states and traverses the dependency among states in a reversed direction.

We first generate a graph representing the transition of states. Let S_I be a given set of states. A reversed transition graph G constructed from S_I is a directed graph $G = \{V, E\}$ where

- V is a set of nodes that is equal to S_I, and
- E is a set of edges given as follows.

$$E = \{(s_j, s_i) \mid \exists s \in S_I, p'_n\ s_i\ \mathsf{True}\ s == s_j \vee p'_n\ s_i\ \mathsf{False}\ s == s_j \\ \vee\ p'_n\ s\ \mathsf{True}\ s_i == s_j \vee p'_n\ s\ \mathsf{False}\ s_i == s_j\}$$

Given a reversed transition graph G, we perform the backward optimization as follows.

1. For each acceptable state $s_i \in S_I$ that satisfies $accept'\ s_i == \mathsf{True}$, compute a set R_i of states reachable from s_i in the graph G.
2. Compute the union R of R_i, i.e., $R = \bigcup_i R_i$.
3. Remove the state that are not included in the set R, that is, this set R is the output of the backward optimization.

Now we return to the example of computing the height of binary trees. Let the input set S_I be $S_I = \{0, 1, 2\}$, which is the output of the forward optimization in Section 8.5.2.

First, we generate a reversed transition graph of S_I that has three nodes $V = \{0, 1, 2\}$, and three edges $V = \{(0, 1), (0, 2), (2, 1)\}$. We omit edges whose start node and end node are the same (e.g., $(0, 0)$), because they affect nothing to reachability. Figure 8.4 illustrates the generated reversed transition graph. We then perform the backward optimization using this graph G. Since state 2 is only acceptable, by computing the reachable states from state 2 we obtain set R as $R = \{1, 2\}$. This R is the output of the backward optimization, and we can remove state 0 by this backward optimization.

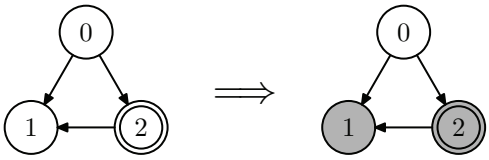


Figure 8.4. A reversed transition graph (Left) and reachable nodes from the acceptable state (Right). The acceptable state is denoted by doubly-lined circle (state 2), and reachable nodes from it are filled gray.

8.5.4 Tupled-Ring Optimization

Tupled-ring optimization is an optimization that removes unnecessary values in the computation of the $\mathsf{reduce_b}$ and $\mathsf{uAcc_b}$ skeletons. This optimization was already discussed in our code generator in Section 7.3. Here, we omit the detailed optimization steps and only show the input matrices for the optimization.

Let S_∞ be a set of values computed in the forward optimization. We consider the generation of input matrices in the following two cases: when there is no state whose corresponding value in S_∞ is constantly zero, and when there exists such a state.

When there is no state whose corresponding value is constantly zero, we generate matrices for the tupled-ring optimization in the same way as Section 8.3. Therefore, given a set of states $S_I = \{s_0, s_1, \ldots, s_{t-1}\}$, we generate $t \times t$ matrices as follows.

$$
\begin{aligned}
&g_l\ b\ (r_0, r_1, \ldots, r_{t-1}) = \{r'_{ij}\} \\
&\quad \textbf{where } r'_{ij} = \textstyle\sum_{\uparrow}[(\textbf{if } m \textbf{ then } b \textbf{ else } 0) + r_k \\
&\qquad\qquad\qquad\qquad \mid k \leftarrow [1..t{-}1], m \leftarrow [\mathsf{True}, \mathsf{False}], p'_n\ s_j\ m\ s_k == s_i]
\end{aligned}
$$

$$
\begin{aligned}
&g_r\ b\ (l_0, l_1, \ldots, l_{t-1}) = \{l'_{ik}\} \\
&\quad \textbf{where } l'_{ik} = \textstyle\sum_{\uparrow}[(\textbf{if } m \textbf{ then } b \textbf{ else } 0) + l_j \\
&\qquad\qquad\qquad\qquad \mid j \leftarrow [1..t{-}1], m \leftarrow [\mathsf{True}, \mathsf{False}], p'_n\ s_j\ m\ s_k == s_i]
\end{aligned}
$$

Based on these matrices, the definition of parameter function and auxiliary functions can be given in the same way as the code generator in Section 7.3.

When there is a state whose corresponding value is constantly zero, we can remove the state by substituting zeros in the definition of functions. Therefore, the number of values in the tuple during the computation is less by one than that in S_I. In the following we discuss the generation of the matrices in this case. Let s_z be the state whose corresponding value is constantly zero, and S' be a set of the other states, $S' = S_I - \{s_z\}$. For readability, we denote the set S' as $S' = \{s_0, s_1, \ldots, s_{t'-1}\}$ where $t' = t - 1$.

We generate matrices of the input of the tupled-ring optimization as follows. The generated matrices are of size $(t'+1) \times (t'+1)$ (or, in other words $t \times t$), and the $(t'+1)$th

column is for the state s_z and the $(t'+1)$th row is the same as that in the identity matrix.

$$
\begin{aligned}
&g_l\ b\ (r_0, r_1, \ldots, r_{t'-1}) = \{r'_{ij}\} \\
&\quad \textbf{where } r'_{nn} = 0 \\
&\qquad\quad r'_{nj} = -\infty \\
&\qquad\quad r'_{in} = \textstyle\sum_{\uparrow}[(\textbf{if } m \textbf{ then } b \textbf{ else } 0) + r_k \\
&\qquad\qquad\qquad\qquad \mid k \leftarrow [1..t'-1], m \leftarrow [\mathsf{True}, \mathsf{False}], p'_n\ s_z\ m\ s_k == s_i] \\
&\qquad\qquad\qquad \uparrow \textstyle\sum_{\uparrow}[\textbf{if } m \textbf{ then } b \textbf{ else } 0 \mid m \leftarrow [\mathsf{True}, \mathsf{False}], p'_n\ s_z\ m\ s_z == s_i] \\
&\qquad\quad r'_{ij} = \textstyle\sum_{\uparrow}[(\textbf{if } m \textbf{ then } b \textbf{ else } 0) + r_k \\
&\qquad\qquad\qquad\qquad \mid k \leftarrow [1..t'-1], m \leftarrow [\mathsf{True}, \mathsf{False}], p'_n\ s_j\ m\ s_k == s_i] \\
&\qquad\qquad\qquad \uparrow \textstyle\sum_{\uparrow}[\textbf{if } m \textbf{ then } b \textbf{ else } 0 \mid m \leftarrow [\mathsf{True}, \mathsf{False}], p'_n\ s_j\ m\ s_z == s_i]
\end{aligned}
$$

$$
\begin{aligned}
&g_r\ b\ (l_0, l_1, \ldots, l_{t'-1}) = \{l'_{ik}\} \\
&\quad \textbf{where } l'_{nn} = 0 \\
&\qquad\quad l'_{nk} = -\infty \\
&\qquad\quad l'_{in} = \textstyle\sum_{\uparrow}[(\textbf{if } m \textbf{ then } b \textbf{ else } 0) + l_j \\
&\qquad\qquad\qquad\qquad \mid j \leftarrow [1..t'-1], m \leftarrow [\mathsf{True}, \mathsf{False}], p'_n\ s_j\ m\ s_z == s_i] \\
&\qquad\qquad\qquad \uparrow \textstyle\sum_{\uparrow}[\textbf{if } m \textbf{ then } b \textbf{ else } 0 \mid m \leftarrow [\mathsf{True}, \mathsf{False}], p'_n\ s_z\ m\ s_z == s_i] \\
&\qquad\quad l'_{ik} = \textstyle\sum_{\uparrow}[(\textbf{if } m \textbf{ then } b \textbf{ else } 0) + l_j \\
&\qquad\qquad\qquad\qquad \mid j \leftarrow [1..t'-1], m \leftarrow [\mathsf{True}, \mathsf{False}], p'_n\ s_j\ m\ s_k == s_i] \\
&\qquad\qquad\qquad \uparrow \textstyle\sum_{\uparrow}[\textbf{if } m \textbf{ then } b \textbf{ else } 0 \mid m \leftarrow [\mathsf{True}, \mathsf{False}], p'_n\ s_z\ m\ s_k == s_i]
\end{aligned}
$$

Now we illustrate the generation of matrices using the example of computing the height of binary trees.

Inheriting the states from the backward transformation, we use $S_I = \{1, 2\}$ for the input of tupled-ring optimization. As seen from the forward optimization, the value corresponding to state 1 is constantly zero. Therefore, we have $S' = \{2\}$ and $s_v = 1$. By using these S' and s_v, we can derive the following matrices for functions g_l and g_r. For readability of the derived functions, we use indices from the states instead of numbers starting at 0.

$$
\begin{aligned}
g_l\ b\ r_2 &= \begin{pmatrix} b & b + r_2 \\ -\infty & 0 \end{pmatrix} \\
g_r\ b\ l_2 &= \begin{pmatrix} b & b + l_2 \\ -\infty & 0 \end{pmatrix}
\end{aligned}
$$

We simulate the computation of tree skeletons on the abstract values, and obtain the following matrix $\boldsymbol{A}$ defined with abstract values.

$$
\boldsymbol{A} = \begin{pmatrix} * & * \\ -\infty & 0 \end{pmatrix}
$$

Based on g_l, g_r, and $\boldsymbol{A}$ derived so far, we can obtain an efficient implementation for computing the height of binary trees. The result of the $\mathsf{reduce_b}$ is now a value instead of a tuple of four values, and in the auxiliary functions only two values need to be computed in the matrices. Figure 8.5 shows the optimized skeletal parallel program for the computation of the height of binary trees.

$$
\begin{array}{l}
height_{\mathrm{b}}\ x = \mathsf{reduce}_{\mathrm{b}}\ k'_n\ (\mathsf{map}_{\mathrm{b}}\ k'_l\ (\lambda x.1)\ x) \\
\textbf{where} \\
\quad k'_l\ a = 1 \\
\quad k'_n\ l_2\ b\ r_2 = b \uparrow (b + l_2) \uparrow (b + r_2) \\
\quad \phi\ b = (b, (0, -\infty)) \\
\quad \psi_n\ l_2\ (b_n\ (p_n, q_n))\ r_2 = \textbf{let}\ v = (p_n + (k'_n\ l_2\ b_n\ r_2)) \uparrow q_n \\
\quad \psi_l\ (b_l, (p_l, q_l))\ (b_n, (p_n, q_n))\ r_2 = (b_l, (p_n, q_n) \otimes (b_n, b_n + r_2) \otimes (p_l, q_l)) \\
\quad \psi_r\ l_2\ (b_n, (p_n, q_n))\ (b_r, (p_r, q_r)) = (b_r, (p_n, q_n) \otimes (b_n, b_n + l_2) \otimes (p_r, q_r)) \\
\quad (p, q) \otimes (p', q') = (p + p', (p + q') \uparrow q)
\end{array}
$$

Figure 8.5. Optimized skeletal parallel program for computing the height of binary trees. The operator $\otimes$ is a specialized operator for matrix multiplication.

8.6 Examples

In the previous sections, we have developed a derivation method and an optimization procedure of skeletal parallel programs for maximum marking problems. We also demonstrated the methods with two examples.

In this section, we furthermore demonstrate the application of the derivation method to the following two problems: the maximum connected-set sum problem and computing diameter of binary trees.

8.6.1 Maximum Connected-Set Sum Problem

The maximum connected-set sum problem is a tree version of the maximum segment sum problem on lists. Given a tree in which each node has a number, and the problem is to find a connected set such that the sum of values in the set gets the maximum. Note that a connected set is not necessarily a subtree. In this section, we develop a skeletal parallel program for the maximum connected-set sum problem on binary trees.

First of all, we write the specification of the problem. The maximum connected-set sum problem is a maximum weight-sum problem, and we can specify the predicate p by using the following four functions *accept*, *cs*, *nomark*, and $root_{\mathrm{b}}$ as $p = accept \circ (cs \vartriangle nomark \vartriangle root_{\mathrm{b}})$.

- *accept*: Function *accept* checks a marking of tree forms at most one connected set.
- *cs*: Function *cs* returns True if a marking forms exactly one connected set.
- *nomark*: Function *nomark* returns True if there exists no node marked as True.
- $root_{\mathrm{b}}$: Function $root_{\mathrm{b}}$ returns the mark of the root.

$$
\begin{array}{l}
accept\ t = cs\ t \vee nomark\ t \\
\\
cs\ (\mathsf{BLeaf}\ a) \quad\quad = a \\
cs\ (\mathsf{BNode}\ l\ b\ r) = (b \wedge (nomark\ l \vee (root_{\mathrm{b}}\ l \wedge cs\ l)) \\
\qquad\qquad\qquad\qquad \wedge (nomark\ r \vee (root_{\mathrm{b}}\ r \wedge cs\ r))) \\
\qquad\qquad\qquad \vee (\neg\ b \wedge ((nomark\ l \wedge cs\ r) \vee (nomark\ r \wedge cs\ l)))
\end{array}
$$

$$
\begin{aligned}
&\mathit{nomark}\ (\mathsf{BLeaf}\ a) &&= \neg\ a\\
&\mathit{nomark}\ (\mathsf{BNode}\ l\ b\ r) &&= \neg\ b \wedge \mathit{nomark}\ l \wedge \mathit{nomark}\ r
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{root}_{\mathrm{b}}\ (\mathsf{BLeaf}\ a) &&= a\\
&\mathit{root}_{\mathrm{b}}\ (\mathsf{BNode}\ l\ b\ r) &&= b
\end{aligned}
$$

We number the states from zero to seven. Based on the definition of the predicate, we derive three functions accept', p'_l, and p'_n that take or return states represented by numbers.

$$
\mathit{accept}'\ x = \begin{cases} \mathsf{True} & \textbf{if } x \in \{2,3,4,5,6,7\} \\ \mathsf{False} & \textbf{otherwise} \end{cases}
$$

$$
\begin{aligned}
&p'_l\ \mathsf{True} = 5\\
&p'_l\ \mathsf{False} = 2
\end{aligned}
$$

$p'_n\ l\ \mathsf{True}$	r 0	1	2	3	4	5	6	7
0	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1
2	1	1	5	5	1	5	5	5
3	1	1	5	5	1	5	5	5
l 4	1	1	1	1	1	1	1	1
5	1	1	5	5	1	5	5	5
6	1	1	5	5	1	5	5	5
7	1	1	5	5	1	5	5	5

$p'_n\ l\ \mathsf{False}$	r 0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0
2	0	0	2	2	4	4	6	6
3	0	0	2	2	4	4	6	6
l 4	0	0	4	4	0	0	4	4
5	0	0	4	4	0	0	4	4
6	0	0	6	6	4	4	6	6
7	0	0	6	6	4	4	6	6

We then remove unnecessary states by the optimization procedure in Section 8.5. The input of the optimization procedure is a set S_I of all the states, $S_I = \{0,1,2,3,4,5,6,7\}$.

In the first step of the optimization, we apply the forward optimization as shown in the following calculations.

$$
S_0 = (-\infty, -\infty, 0, -\infty, -\infty, *, -\infty, -\infty)
$$

$$
\begin{aligned}
S_1 &= k'\ S_0 +_{\odot} S_0\\
&= k'\ (-\infty, -\infty, 0, -\infty, -\infty, *, -\infty, -\infty) +_{\odot} (-\infty, -\infty, 0, -\infty, -\infty, *, -\infty, -\infty)\\
&= (*, -\infty, 0, -\infty, *, *, -\infty, -\infty) +_{\odot} (-\infty, -\infty, 0, -\infty, -\infty, *, -\infty, -\infty)\\
&= (*, -\infty, 0, -\infty, *, *, -\infty, -\infty)
\end{aligned}
$$

$$
\begin{aligned}
S_2 &= k'\ S_1 +_{\odot} S_1\\
&= k'\ (*, -\infty, 0, -\infty, *, *, -\infty, -\infty) +_{\odot} (*, -\infty, 0, -\infty, *, *, -\infty, -\infty)\\
&= (*, *, 0, -\infty, *, *, -\infty, -\infty) +_{\odot} (*, -\infty, 0, -\infty, *, *, -\infty, -\infty)\\
&= (*, *, 0, -\infty, *, *, -\infty, -\infty)
\end{aligned}
$$

$$
\begin{aligned}
S_3 &= k'\ S_2 +_{\odot} S_2\\
&= k'\ (*, *, 0, -\infty, *, *, -\infty, -\infty) +_{\odot} (*, *, 0, -\infty, *, *, -\infty, -\infty)\\
&= (*, *, 0, -\infty, *, *, -\infty, -\infty) +_{\odot} (*, *, 0, -\infty, *, *, -\infty, -\infty)\\
&= (*, *, 0, -\infty, *, *, -\infty, -\infty)\\
&= S_2
\end{aligned}
$$

From the calculations above, we can remove states 3, 6, and 7 from S_I and obtain a reduced set S'_I of states $S'_I = \{0,1,2,4,5\}$.

In the second step of the optimization, we apply the backward optimization. The reversed transition graph G constructed from S'_I is given as follows.

$$G = (\{0,1,2,4,5\},\\ \{(0,1),(0,2),(0,4),(0,5),(1,0),(1,2),(1,4),(1,5),(4,2),(4,5),(5,2)\})$$

Starting from state 2, we have reachable states R_2 as $R_2 = \{2\}$. Similarly, we have reachable states from states 4 and 5 as $R_4 = \{2,4,5\}$ and $R_5 = \{2,5\}$, respectively. The union of these three sets yields $R = \{2,4,5\}$, and we can remove states 0 and 1 from the set of states.

This set of states $\{2,4,5\}$ does not change anymore by both the forward optimization and the backward optimization, we proceed to the tupled-ring optimization. As we have seen in the calculations in the forward optimization, the value corresponding to state 2 is constantly zero. Therefore, we have the following input for the tupled-ring optimization.

$$S' = \{4,5\}, s_z = 2$$

For readability, we extract the elements related to the states 2, 4, and 5 as shown in the following tables. In the following tables, _ denotes an element that can be ignored in the following derivation since it yields a state rather that the states 2, 4, and 5.

p'_n l True	r	2	4	5
l	2	5	_	5
	4	_	_	_
	5	5	_	5

p'_n l False	r	2	4	5
l	2	2	4	4
	4	4	_	_
	5	4	_	_

Using these two tables, we can derive two matrices for functions g_l and g_r as follows.

$$g_l\ b\ (r_4, r_5) = \begin{pmatrix} 0 & 0 & r_4 \uparrow r_5 \\ -\infty & r_5 \uparrow 0 & r_5 \uparrow 0 \\ -\infty & -\infty & 0 \end{pmatrix}$$

$$g_r\ b\ (l_4, l_5) = \begin{pmatrix} 0 & 0 & l_4 \uparrow l_5 \\ -\infty & l_5 \uparrow 0 & l_5 \uparrow 0 \\ -\infty & -\infty & 0 \end{pmatrix}$$

By abstracting values we have a matrix

$$\begin{pmatrix} 0 & 0 & * \\ -\infty & * & * \\ -\infty & -\infty & 0 \end{pmatrix},$$

and simulating the computation of the $\mathsf{reduce}_{\mathrm{b}}$, we have the following matrix $\boldsymbol{A}$ with abstract values as the output of the tupled-ring optimization.

$$\boldsymbol{A} = \begin{pmatrix} 0 & * & * \\ -\infty & * & * \\ -\infty & -\infty & 0 \end{pmatrix}$$

After optimization, we have.

$$S' = \{4, 5\},\ s_v = 2,\ \boldsymbol{A} = \begin{pmatrix} 0 & * & * \\ -\infty & * & * \\ -\infty & -\infty & 0 \end{pmatrix}$$

Finally, we generate a skeletal parallel program. The generated skeletal parallel program *mcss* is given as follows. We can derive the functions k'_l and k'_n just by substituting the parameter. For the auxiliary functions, we specialize the matrix multiplication with $+$ and $\uparrow$ into operator $\otimes$ in which only four values denoted by $*$ are used.

$$\begin{aligned}
&mcss\ t = \mathsf{reduce_b}\ k'_n\ (\mathsf{map_b}\ k'_l\ id\ t) \\
&\textbf{where} \\
&\quad k'_l\ a = (-\infty, a) \\
&\quad k'_n\ (l_4, l_5)\ b\ (r_4, r_5) = (l_4 \uparrow l_5 \uparrow r_4 \uparrow r_5,\ (l_5 + r_5) \uparrow l_5 \uparrow r_5 \uparrow 0) \\
&\quad \phi\ b = \left(b, \begin{pmatrix} -\infty & -\infty \\ 0 & -\infty \end{pmatrix}\right) \\
&\quad \psi_n\ (l_4, l_5)\ \left(b_n, \begin{pmatrix} a_{01} & a_{02} \\ a_{11} & a_{12} \end{pmatrix}\right)\ (r_4, r_5) \\
&\qquad = \textbf{let}\ (x_4, x_5) = k'_n\ (l_4, l_5)\ b_n\ (r_4, r_5) \\
&\qquad\quad \textbf{in}\ (x_4 \uparrow (a_{01} + x_5) \uparrow a_{02},\ (a_{11} + x_5) \uparrow a_{12}) \\
&\quad \psi_l\ (b_l, \boldsymbol{M}_l)\ (b_n, \boldsymbol{M}_n)\ (r_4, r_5) = \left(b_l, \boldsymbol{M}_n \otimes \begin{pmatrix} 0 & r_0 \uparrow r_1 \\ r_1 \uparrow 0 & r_1 \uparrow 0 \end{pmatrix} \otimes \boldsymbol{M}_r\right) \\
&\quad \psi_r\ (l_4, l_5)\ (b_n, \boldsymbol{M}_n)\ (b_r, \boldsymbol{M}_r) = \left(b_r, \boldsymbol{M}_n \otimes \begin{pmatrix} 0 & l_0 \uparrow l_1 \\ l_1 \uparrow 0 & l_1 \uparrow 0 \end{pmatrix} \otimes \boldsymbol{M}_l\right)
\end{aligned}$$

Here, auxiliary functions satisfy $k'_n = \langle \phi, \psi_n, \psi_l, \psi_r \rangle_{\mathrm{u}}$.

We can easily extend the skeletal program to find a marking that yields the maximum connected-set sum based on the derivation methods in Section 8.4. Let *mcsm* be a function that returns a marking for the maximum connected-set sum. The derivation is done by straightforward substitution of the parameter functions $accept'$, p'_l and p'_n. In the substitution, we only consider the values corresponding to the states remaining after the optimization procedures. Note that a node in the result tree of the $\mathsf{uAcc_b}$ skeleton do not have the value zero corresponding to the state 2. Therefore, we complement it in the computation of the downwards accumulation.

$$\begin{aligned}
&mcsm\ t = \textbf{let}\ bt = \mathsf{uAcc_b}\ k'_n\ (\mathsf{map_b}\ k'_l\ id\ t) \\
&\qquad\qquad zt = \mathsf{zip4_b}\ t\ bt\ (\mathsf{getchl_b}\ bt)\ (\mathsf{getchr_b}\ bt) \\
&\qquad\qquad (w_4, w_5) = root_{\mathrm{b}}\ bt \\
&\qquad\qquad m = 0 \uparrow w_4 \uparrow w_5 \\
&\qquad\qquad c\ = \textbf{if}\ m == 0\ \textbf{then}\ 2\ \textbf{else if}\ m == w_4\ \textbf{then}\ 4\ \textbf{else}\ 5 \\
&\qquad\qquad ct = \mathsf{dAcc_b}\ (g_l, g_r)\ c\ zt \\
&\qquad\ \textbf{in}\ \mathsf{zipwith_b}\ k''_l\ k''_n\ ct\ zt
\end{aligned}$$

$$
\begin{aligned}
&(g_l\ c \vartriangle g_r\ c)\ (b,(w_4,w_5),(l_4,l_5),(r_4,r_5))\\
&\quad = \mathit{fst}\ [(j,k) \mid j,k \leftarrow [2,4,5], m \leftarrow [\mathsf{True},\mathsf{False}],\\
&\qquad\qquad w_c == l_j + (\textbf{if}\ m\ \textbf{then}\ b\ \textbf{else}\ 0) + r_k, p'_n\ j\ m\ k == c]\\
&\qquad \textbf{where}\ w_2 = l_2 = r_2 = 0\\
&k''_l\ c\ (a,(w_4,w_5),\_,\_) = (w_5 == a \wedge c == 5)\\
&k''_n\ c\ (b,(w_4,w_5),(l_4,l_5),(r_4,r_5))\\
&\quad = \mathit{fst}\ [m \mid j,k \leftarrow [2,4,5], m \leftarrow [\mathsf{True},\mathsf{False}],\\
&\qquad\qquad w_c == l_j + (\textbf{if}\ m\ \textbf{then}\ b\ \textbf{else}\ 0) + r_k, p'_n\ j\ m\ k == c]\\
&\qquad \textbf{where}\ w_2 = l_2 = r_2 = 0\\
&(\phi_l \vartriangle \phi_r)\ (b,(w_4,w_5),(l_4,l_5),(r_4,r_5))\\
&\quad = ((j'_2,j'_4,j'_5),(k'_2,k'_4,k'_5))\\
&\quad \textbf{where}\\
&\quad (j'_i,k'_i) = \mathit{fst}\ [(j,k) \mid j,k \leftarrow [2,4,5], m \leftarrow [\mathsf{True},\mathsf{False}],\\
&\qquad\qquad\qquad w_i == l_j + (\textbf{if}\ m\ \textbf{then}\ b\ \textbf{else}\ 0) + r_k, p'_n\ j\ m\ k == i]\\
&\qquad\qquad \textbf{where}\ w_2 = l_2 = r_2 = 0\\
&\psi_u\ (n_2,n_4,n_5)\ (m_2,m_4,m_5) = (p_2,p_4,p_5)\\
&\quad \textbf{where}\ p_i = \textbf{let}\ i' = m_i\ \textbf{in}\ n_{i'}\\
&\psi_d\ c\ (n_2,n_4,n_5) = n_c
\end{aligned}
$$

Here, auxiliary functions satisfy $(g_l, g_r) = \langle \phi_l, \phi_r, \psi_u, \psi_d \rangle_{\mathrm{d}}$. The functions k'_l, k'_n and auxiliary functions for k'_n are defined for the maximum connected-set sum *mcss*.

8.6.2 Diameter of Trees

Diameter of an undirected (weighted) graph is the maximum among the (weigthed) lengths of shortest paths for all pair of nodes [56]. A tree is an instance of undirected graphs where there is exactly one path for each pair of nodes, and thus diameter of a tree is given by the length of the longest path. We can specify the computation of the diameter of a given binary tree as a maximum weight-sum problem.

Note that an edge of a tree connects to a unique child node. Let input t be a binary tree in which a node except the root represents the weight of the edge that connects the node to its parent, and the root have value zero. We assume the weight of the edge to be positive. On this tree, paths are classified into the following two in terms of their appearance.

- *Complete path*: A path is complete if there exists a node from which two non branching paths go downwards to a leaf.
- *Partial path*: A path is partial if a non branching path goes downwards from the root to a leaf.

In the definition above, we omit paths that are obviously shorter than another based on the fact that the longest path on trees connects two leaves. To specify the paths on binary trees, we define the following four functions. We can define the predicate p for the maximum weight-sum problem as $p = \mathit{accept} \circ (\mathit{cpath} \vartriangle \mathit{ppath} \vartriangle \mathit{nomark})$.

- *accept*: Function *accept* checks a marking of a binary tree forms a complete path.
- *cpath*: Function *cpath* returns True if a tree has exactly one complete path inside.
- *ppath*: Function *ppath* returns True if a tree has exactly one partial path.
- *nomark*: Function *nomark* returns True if no node in a tree is marked as True.

$$accept\ t = cpath\ t$$

$$\begin{aligned} cpath\ (\mathsf{BLeaf}\ a) &= \mathsf{False} \\ cpath\ (\mathsf{BNode}\ l\ b\ r) &= \neg\ b \wedge ((cpath\ l \wedge nomark\ r) \vee (cpath\ r \wedge nomark\ l) \\ &\qquad \vee (ppath\ l \wedge ppath\ r)) \end{aligned}$$

$$\begin{aligned} ppath\ (\mathsf{BLeaf}\ a) &= a \\ ppath\ (\mathsf{BNode}\ l\ b\ r) &= b \wedge ((ppath\ l \wedge nomark\ r) \vee (ppath\ r \wedge nomark\ l) \\ &\qquad \vee (nomark\ l \wedge nomark\ r)) \end{aligned}$$

$$\begin{aligned} nomark\ (\mathsf{BLeaf}\ a) &= \neg\ a \\ nomark\ (\mathsf{BNode}\ l\ b\ r) &= \neg\ b \wedge nomark\ l \wedge nomark\ r \end{aligned}$$

Let *diameter* be a function that computes the diameter of a given binary tree, then we have $diameter = mws\ (accept \circ (cpath \vartriangle ppath \vartriangle nomark))$.

Now we derive a skeletal parallel program from this specification. First, we number the states from zero to seven, and derive three functions $accept'$, p'_l, and p'_n.

$$accept'\ x = \begin{cases} \mathsf{True} & \textbf{if } x \in \{4,5,6,7\} \\ \mathsf{False} & \textbf{otherwise} \end{cases}$$

$$\begin{aligned} p'_l\ \mathsf{True} &= 2 \\ p'_l\ \mathsf{False} &= 1 \end{aligned}$$

$p'_n\ l$ True r	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	2	2	2	0	2	2	2
2	0	2	0	2	0	2	0	2
3	0	2	2	2	0	2	2	2
l 4	0	0	0	0	0	0	0	0
5	0	2	2	2	0	2	2	2
6	0	2	0	2	0	2	0	2
7	0	2	2	2	0	2	2	2

(r: columns)

$p'_n\ l$ False r	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	1	0	1	4	5	4	5
2	0	0	4	4	0	0	4	4
3	0	1	4	5	4	5	4	5
l 4	0	4	0	4	0	4	0	4
5	0	5	0	5	4	5	4	5
6	0	4	4	4	0	4	4	4
7	0	5	4	5	4	5	4	5

(r: columns)

We then reduce the number of states by the optimization procedure. The input set of states is $S_I = \{0,1,2,3,4,5,6,7\}$.

In the first step of the optimization, we apply the forward optimization.

$$S_0 = (-\infty, 0, *, -\infty, -\infty, -\infty, -\infty, -\infty)$$

$$\begin{aligned} S_1 &= k'\ S_0 +_\odot S_0 \\ &= k'\ (-\infty, 0, *, -\infty, -\infty, -\infty, -\infty, -\infty) +_\odot (-\infty, 0, *, -\infty, -\infty, -\infty, -\infty, -\infty) \\ &= (*, 0, *, -\infty, *, -\infty, -\infty, -\infty) +_\odot (-\infty, 0, *, -\infty, -\infty, -\infty, -\infty, -\infty) \\ &= (*, 0, *, -\infty, *, -\infty, -\infty, -\infty) \end{aligned}$$

$$
\begin{aligned}
S_2 &= k'\ S_1 +_{\odot} S_1 \\
&= k'\ (*, 0, *, -\infty, *, -\infty, -\infty, -\infty) +_{\odot} (*, 0, *, -\infty, *, -\infty, -\infty, -\infty) \\
&= (*, 0, *, -\infty, *, -\infty, -\infty, -\infty) +_{\odot} (*, 0, *, -\infty, *, -\infty, -\infty, -\infty) \\
&= (*, 0, *, -\infty, *, -\infty, -\infty, -\infty) \\
&= S_1
\end{aligned}
$$

From the calculations above, we can remove states 3, 5, 6, and 7 from S_I and obtain a reduced set of states $S'_I = \{0, 1, 2, 4\}$.

In the second step of the optimization, we apply the backward optimization to S'_I. The reversed transition graph G constructed from S'_I is given as follows.

$$G = (\{0, 1, 2, 4\}, \{(0, 1), (0, 2), (0, 4), (2, 1), (4, 1), (4, 2)\})$$

State 4 is the only acceptable state in S'_I. By computing reachable states in G, we obtain the set R of reachable states as $R = \{1, 2, 4\}$. Therefore, we can remove state 0 from S'_I.

This set of states $\{1, 2, 4\}$ does not change by both the forward optimization and the backward optimization anymore, and therefore we proceed to the tupled-ring optimization. As we have seen in the calculations in the forward optimization, the value corresponding to state 1 is constantly zero. Therefore, we have the following input for the tupled-ring optimization.

$$S' = \{2, 4\}, s_z = 1$$

To make the derivation easy, we introduce the following two tables by extracting the elements related to the states 1, 2, and 4. In the following tables, _ denotes an element that can be ignored in the following derivation since it yields a state rather than the states 1, 2, and 4.

$p'_n\ l$ True r	1	2	4
1	2	2	_
l 2	2	_	_
4	_	_	_

$p'_n\ l$ False r	1	2	4
1	1	_	4
l 2	_	4	_
4	4	_	_

Using these two tables, we can derive two matrices for functions g_l and g_r.

$$
g_l\ b\ (r_2, r_4) = \begin{pmatrix} b & -\infty & b + r_2 \\ r_2 & 0 & r_4 \\ -\infty & -\infty & 0 \end{pmatrix}
$$

$$
g_r\ b\ (l_2, l_4) = \begin{pmatrix} b & -\infty & b + l_2 \\ l_2 & 0 & l_4 \\ -\infty & -\infty & 0 \end{pmatrix}
$$

By comparing these two matrices and simulating the computation of the $\mathsf{reduce}_{\mathrm{b}}$ skeleton, we obtain a matrix $\boldsymbol{A}$ with abstract values as follows.

$$
\boldsymbol{A} = \begin{pmatrix} * & -\infty & * \\ * & 0 & * \\ -\infty & -\infty & 0 \end{pmatrix}
$$

Based on the results of the optimization, we finally generate a skeletal parallel program. The generated skeletal parallel program is as follows. The operator $\otimes$ is a specialized one for the matrix multiplication with four values denoted by $*$ in the abstract matrix $\boldsymbol{A}$.

$$
\begin{array}{l}
\mathit{diameter}\ t = \mathsf{reduce}_{\mathrm{b}}\ k'_n\ (\mathsf{map}_{\mathrm{b}}\ k'_l\ \mathit{id}\ t) \\
\textbf{where} \\
k'_l\ a = (a, -\infty) \\
k'_n\ (l_0, l_1)\ b\ (r_0, r_1) = (b \uparrow (b + r_0) \uparrow (l_0 + b),\ r_1 \uparrow (l_0 + r_0) \uparrow l_1) \\
\phi\ b = \left(b, \begin{pmatrix} 0 & -\infty \\ -\infty & -\infty \end{pmatrix}\right) \\
\psi_n\ (l_2, l_4)\ \left(b_n, \begin{pmatrix} a_{00} & a_{02} \\ a_{10} & a_{12} \end{pmatrix}\right)\ (r_2, r_4) \\
\quad = \textbf{let}\ (x_2, x_4) = k'_n\ (l_2, l_4)\ b_n\ (r_2, r_4) \\
\qquad \textbf{in}\ ((a_{00} + x_2) \uparrow a_{02},\ (a_{10} + x_2) \uparrow x_4 \uparrow a_{12}) \\
\psi_l\ (b_l, \boldsymbol{M}_l)\ (b_n, \boldsymbol{M}_n)\ (r_0, r_1) = \left(b_l, \boldsymbol{M}_n \otimes \begin{pmatrix} b_n & b_n + r_0 \\ r_0 & r_1 \end{pmatrix} \otimes \boldsymbol{M}_r\right) \\
\psi_r\ (l_0, l_1)\ (b_n, \boldsymbol{M}_n)\ (b_r, \boldsymbol{M}_r) = \left(b_r, \boldsymbol{M}_n \otimes \begin{pmatrix} b_n & b_n + l_0 \\ l_0 & l_1 \end{pmatrix} \otimes \boldsymbol{M}_l\right) \\
\begin{pmatrix} a_{00} & a_{02} \\ a_{10} & a_{12} \end{pmatrix} \otimes \begin{pmatrix} b_{00} & b_{02} \\ b_{10} & b_{12} \end{pmatrix} = \begin{pmatrix} a_{00} + b_{00} & (a_{00} + b_{02}) \uparrow a_{02} \\ (a_{10} + b_{00}) \uparrow b_{10} & (a_{12} + b_{02}) \uparrow b_{12} \uparrow a_{12} \end{pmatrix}
\end{array}
$$

Here, the auxiliary functions ϕ, ψ_n, ψ_l and ψ_r satisfy $k'_n = \langle \phi, \psi_n, \psi_l, \psi_r \rangle_{\mathrm{u}}$.

8.7 Short Summary

In this chapter, we have developed a method of deriving skeletal parallel programs for the maximum marking problems and maximum weight-sum problems based on Sasano et al.'s method of deriving sequential programs. The main idea in our method is that we apply theorems of tupled-ring property and finiteness property after decomposing the computation into an upwards accumulation and a downwards accumulation. To apply these theorems, we assume the structure of the input to be binary tree, which is the only condition required in addition to Sasano et al.'s derivation method. We have also developed a procedure to optimize the derived skeletal parallel programs. Two subroutines are also applicable to the optimization of sequential programs.

As we have seen in the running example of the party planning problem, we can omit some values in the downwards accumulation of the derived skeletal parallel programs. Deciding the dependency between two children or between the parent and a child is an interesting and important for further optimization of skeletal parallel programs. Whether or not we can remove the additional condition of binary-tree structures is also an open problem.

Chapter 9

Parallelizing XPath Queries

In recent years, XML is getting widely used for storing structured data. So far several languages have been proposed for manipulating XML trees, such as XSLT [72] and XQuery [23], where XPath [13] plays a very important role in addressing selections of nodes from XML trees. There have been several studies on efficient implementations of XPath queries, for example, in the context of stream processing [55, 103, 105] and an implementation on relational databases [80].

Parallel processing has several advantages for manipulating huge XML trees, in particular, by speedups with multiple processors and wide memory spaces. However, few studies have been addressed on parallelizing XML processing due to the difficulties in parallel programming for trees. Skillicorn [121] showed several parallelizable manipulations on structured documents based on skeletal parallel programming, but his method is limited to simple manipulations of trees specified by relations between parent and children, and complicated manipulations specified with relations between siblings are out of the range of his method.

In this chapter, we give a systematic method of parallelizing XPath queries. Our method is powerful enough to parallelize an important class of XPath queries. In addition to relations between parent and children, we can specify XPath queries with relations between siblings. We can also add some conditions of subtrees using predicates. As far as we are aware, this is the first attempt to parallelize the class of XPath queries in a systematic way.

This chapter is organized as follows. Section 9.1 specifies the core XPath queries, which are the target of our parallelization algorithm, and introduces a binary-tree representation of XML trees. In Section 9.2, we define two types of homomorphisms on binary trees and discuss the relationship between these homomorphisms and tree accumulations. We give the core parallelization algorithms for XPath queries without predicates in Section 9.3 and for XPath queries in a single predicate in Section 9.4. In Section 9.5, we show the overall parallelization algorithm. Section 9.6 summarizes this chapter.

The preliminary work of this chapter is given in [133].

9.1 XPath Queries and Binary-Tree Representation of XML Trees

9.1.1 XPath Queries

The *XPath* language [13] is a language for addressing parts of an XML tree. Figure 9.1 shows a subset of XPath language, which is the target of our parallelization algorithms in this chapter. An XPath expression (*xpath-expr*) consists of a list of location steps (*step*), each of which includes one of the three forward axes, `child`, `desc` (descendant), and `foll-sib` (following-sibling)[1], a nametest with the tag name or a wildcard `*`, and a predicate that specifies the condition of the subtree rooted at the node specified. Predicates are also specified with location steps and can be nested.

We exclude several axes defined in the specification of the XPath language [13]. There are axes such as `self`, `following` and reverse axes. It is worth noting that we can rewrite XPath queries defined with these axes into ones defined with the above three forward axes only. We can transform such XPath queries by the rule

$$\texttt{following::a} \Longrightarrow \texttt{parent::*/foll-sib::*/descendant-or-self::a}\,,$$

following the *rare algorithm* developed by Olteanu et al. [103], which removes the reverse axes. For example, given an XPath query

$$\texttt{/desc::a/desc-or-self::b/parent::a}\,,$$

we can remove the axis `parent` by transforming the query into the following equivalent XPath queries

$$\texttt{/desc::a[child::b]} \quad \text{or} \quad \texttt{/desc::a/desc::a[child::b]}\,.$$

Here, the first expression `/desc::a[child::b]` may occur if the first and second `a` in the original XPath query are the same. If an XPath query includes operators || and &&, then we process it by computing the XPath queries connected by the operators one by one.

xpath-expr	::=	(*/step*)+
step	::=	*forward-axis nametest* [*predicate*]
forward-axis	::=	`child::` | `desc::` | `foll-sib::`
nametest	::=	*string* | `*`
predicate	::=	`[` *step* (*/step*)∗ `]`

Figure 9.1. The definition of core XPath queries in this chapter.

[1] We use these abbreviations to shorten the description of XPath queries.

9.1.2 Binary-Tree Representation of XML Trees

We have two set of tree skeletons, binary-tree skeletons in Chapter 2, and rose-tree skeletons in Chapter 4. Though the XML trees can be formalized as rose trees, we deal with the XML trees based on the binary-tree representation shown in Figure 4.9 to develop parallel programs by the systematic derivation methods on binary trees in Chapter 5. In the binary-tree representation of XML trees, we assume that the internal nodes and leaves have the same string type. Thus, leaves have dummy values of strings.

It is worth noting again that the left child of an internal node in the binary-tree representation indicates the left-most child in the original XML tree and the right child indicates the next right sibling. All the leaves are dummy nodes in the binary-tree representation. To make the program easy to read, in this chapter we denote an internal node as $\mathsf{BNode}\ c\ x\ s$.

9.2 Two types of Homomorphisms and Tree Accumulations

Once a recursive data structure is given, algorithms over it are often given as recursive functions along the definition of the structure. In Section 2.2, we have already defined the tree homomorphism and discussed relationship between tree homomorphisms and tree accumulations. In this section, we introduce another homomorphism on binary trees called *path homomorphism* [48, 119] and formalize two tree accumulations in another way.

9.2.1 Tree Homomorphism and Upwards Accumulation

A tree homomorphism $h = (\![k_l, k_n]\!)_{\mathrm{b}}$ is a function defined recursively on binary trees as follows.

$$\begin{aligned} h\ (\mathsf{BLeaf}\ a) &= k_l\ a \\ h\ (\mathsf{BNode}\ c\ x\ s) &= k_n\ (h\ c)\ x\ (h\ s) \end{aligned}$$

We write a tree homomorphism as $\mathit{TreeHom}(k_l, k_n)$ instead of $(\![k_l, k_n]\!)_{\mathrm{b}}$ to distinguish it from the path homomorphism introduced later.

We showed that upwards accumulations on binary trees are tree homomorphism in Section 2.2. Here, we give another specification to the upwards accumulations [47, 119]. The specification is a tree version of the scan lemma on lists:

$$\mathsf{scan}\ (\oplus) = \mathsf{map}\ (\mathsf{reduce}\ (\oplus)) \circ \mathit{inits}\ .$$

Let *subtrees* be a function that takes a binary tree and returns a tree of the same shape where each node has a subtree of the input rooted at the node. Using the notation of tree homomorphism we can define the function *subtrees* as follows. Figure 9.2 illustrates the function *subtrees*.

$$\begin{aligned} \mathit{subtrees} = (\![k'_l, k'_n]\!)_{\mathrm{b}} \\ \quad \mathbf{where}\ k'_l\ a &= \mathsf{BLeaf}\ (\mathsf{BLeaf}\ a) \\ k'_n\ c\ x\ s &= \mathsf{BNode}\ c\ (\mathsf{BNode}\ (\mathit{root}_{\mathrm{b}}\ c)\ x\ (\mathit{root}_{\mathrm{b}}))\ s \end{aligned}$$

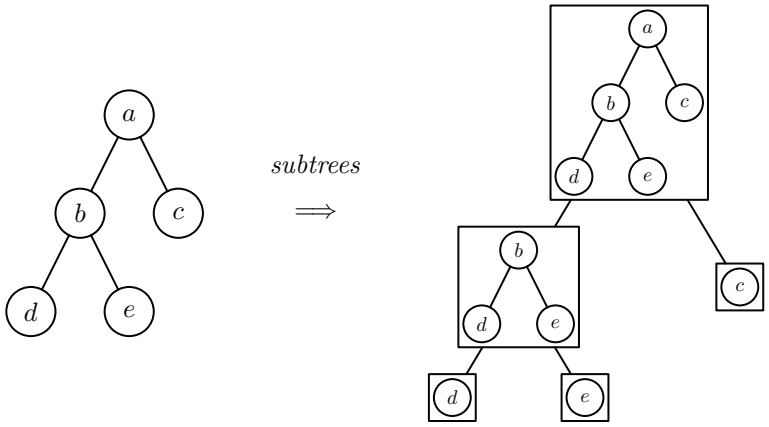


Figure 9.2. Illustration of function *subtrees*

An upwards accumulation takes a binary tree and computes a tree of the same shape by updating values in a bottom-up manner. Let k_l and k_n be given functions, we specify the upwards accumulation using tree homomorphism and the function *subtrees* as:

$$\mathsf{map_b}\ (\mathit{TreeHom}(k_l, k_n))\ (\mathit{TreeHom}(k_l, k_n)) \circ \mathit{subtrees}\ .$$

This definition says that if we can define a tree homomorphism $\mathit{TreeHom}(k_l, k_n)$ then we can also specify the upwards accumulation in which the tree homomorphism is applied to each subtree. We can implement the upwards accumulation by tree skeletons as follows.

$$\mathsf{map_b}\ (\mathit{TreeHom}(k_l, k_n))\ (\mathit{TreeHom}(k_l, k_n)) \circ \mathit{subtrees} = \mathsf{uAcc_b}\ k_n \circ \mathsf{map_b}\ k_l\ \mathit{id}\ .$$

9.2.2 Path Homomorphism and Downwards Accumulation

We then formalize another data structure on binary trees to represent paths from the root. The datatype BPath defined as follows represents a sequence of nodes from the root node to a node of binary trees. The constructors have specialized names for the binary-tree representation of XML trees.

$$\begin{aligned}&\textbf{data}\ \mathsf{BPath}\ \alpha = \mathsf{Singleton}\ \alpha\\ &\qquad\qquad\ \ |\ \mathsf{Child}\ \alpha\ (\mathsf{BPath}\ \alpha)\\ &\qquad\qquad\ \ |\ \mathsf{Sibling}\ \alpha\ (\mathsf{BPath}\ \alpha)\end{aligned}$$

Let the given binary tree be

$$\begin{aligned}&\mathsf{BNode}\ (\mathsf{BNode}\ (\mathsf{BLeaf}\ a)\ b\ (\mathsf{BLeaf}\ c))\\ &\qquad d\\ &\qquad (\mathsf{BLeaf}\ e)\ ,\end{aligned}$$

the path from the root to leaf c is given as

$$\mathsf{Child}\ d\ (\mathsf{Sibling}\ b\ (\mathsf{Sibling}\ c))\ .$$

Many top-down computations on binary trees that take a path from the root can be written on this BPath structure. We formalize such computations as the following *path homomorphisms*.

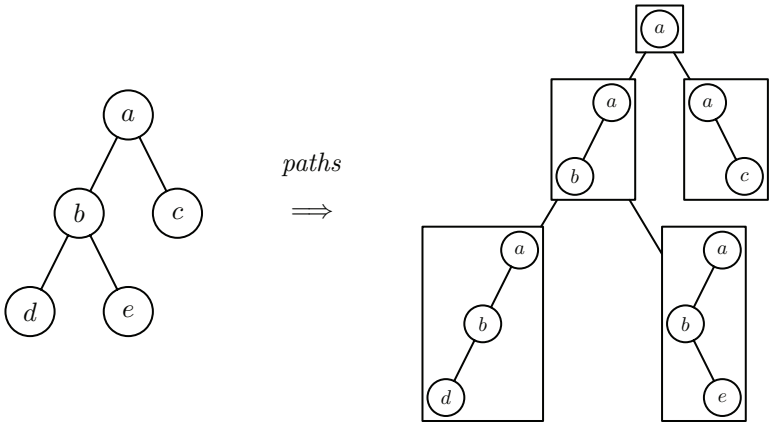


Figure 9.3. Illustration of function *paths*

Definition 9.1 (Path Homomorphism) Function h is said to be a *path homomorphism*, if it is define in the following form with some functions k_x, k_c, and k_s.

$$\begin{aligned} h\ (\mathsf{Singleton}\ x) &= k_x\ x \\ h\ (\mathsf{Child}\ x\ c) &= k_c\ x\ (f\ c) \\ h\ (\mathsf{Sibling}\ x\ s) &= k_s\ x\ (f\ s) \end{aligned}$$

We may denote path homomorphism h as $h = PathHom(k_x, k_c, k_s)$. □

Let us consider the computation of the depth of a node in an XML tree. Given a path from the root to a node in the binary-tree representation, we can compute the depth in the XML tree with the following path homomorphism *xmldepth*.

$$\begin{aligned} &xmldepth\ (\mathsf{Singleton}\ x) = 0 \\ &xmldepth\ (\mathsf{Child}\ x\ c) = 1 + xmldepth\ c \\ &xmldepth\ (\mathsf{Sibling}\ x\ s) = xmldepth\ s \end{aligned}$$

We then discuss the relation between path homomorphisms and downwards accumulations. Let *paths* be a function that takes a binary tree and returns a tree of the same shape in which each node has the path from the root to the node in the input tree. Figure 9.3 illustrates the function *paths*. We can define the function *paths* as follows. Note that we assume that the internal nodes and leaves in the input binary tree have the same type.

$$\begin{aligned} paths\ (\mathsf{BLeaf}\ a) &= \mathsf{BLeaf}\ (\mathsf{Singleton}\ a) \\ paths\ (\mathsf{BNode}\ c\ x\ s) &= \mathsf{BNode}\ (\mathsf{map_b}\ (\lambda c.\mathsf{Child}\ x\ c)\ (paths\ c))\ (\mathsf{Singleton}\ x) \\ &\qquad (\mathsf{map_b}\ (\lambda s.\mathsf{Sibling}\ x\ s)\ (paths\ s)) \end{aligned}$$

The downwards accumulation can be specified using the function *paths* and a path homomorphism in the following way:

$$\mathsf{map_b}\ (PathHom(k_x, k_c, k_s))\ (PathHom(k_x, k_c, k_s)) \circ paths\ .$$

By the same technique as the case of upwards accumulations, we can implement downwards accumulations using the $\mathsf{dAcc_b}$ and $\mathsf{zipwith_b}$ skeletons. Note that the results of the $\mathsf{dAcc_b}$ skeleton is a tree whose nodes have functions.

$$
\begin{array}{l}
(map\ (PathHom\ (k_x, k_c, k_s))\ (PathHom\ (k_x, k_c, k_s)) \circ paths)\ t \\
\quad = \mathsf{zipwith_b}\ k'_x\ k'_x\ t\ (\mathsf{dAcc_b}\ (k'_c, k'_s)\ id\ t) \\
\quad\quad \textbf{where}\ k'_x\ x\ f = f\ (k_x\ x) \\
\quad\quad\quad\quad k'_c\ c\ x = c \circ k_c\ x \\
\quad\quad\quad\quad k'_s\ c\ x = c \circ k_s\ x
\end{array}
$$

In the parallel implementation of the $\mathsf{dAcc_b}$ skeleton, a sufficient condition for existence of efficient parallel implementation is the existence of semi-associative operator $\ominus$ shared in both functions k_c and k_s.

Lemma 9.1 *Given a path homomorphism $PathHom(k_x, k_c, k_s)$, the downwards accumulation defined as*

$$\mathsf{map_b}\ (PathHom(k_x, k_c, k_s))\ (PathHom(k_x, k_c, k_s)) \circ paths$$

can be efficiently implemented in parallel, if there exists a semi-associative operator $\ominus$ and two functions k''_c and k''_s satisfying the following equations.

$$
\begin{array}{l}
k_c\ x\ c = k''_c\ x \ominus c \\
k_s\ x\ s = k''_s\ x \ominus s
\end{array}
$$

Proof. Under this condition, we can easily define auxiliary functions for the $\mathsf{dAcc_b}$ skeleton by considering functional form $f_a = \lambda c.a \ominus c$ in the same way as the proof of Theorem 5.7. Let $\oplus$ be an associative operator satisfying the equation $a \ominus (b \ominus c) = (a \oplus b) \ominus c$, then we can derive the definition of the skeletal parallel program as follows.

$$
\begin{array}{l}
(\mathsf{map_b}\ (PathHom\ (k_x, k_c, k_s))\ (PathHom\ (k_x, k_c, k_s)) \circ paths)\ t \\
\quad = \mathsf{zipwith_b}\ k'_x\ k'_x\ t\ (\mathsf{dAcc_b}\ \langle k''_c, k''_s, \psi_u, \psi_d \rangle_{\mathrm{d}}\ \iota_\oplus\ t) \\
\quad\quad \textbf{where}\ k'_x\ x\ c = c \ominus k_x\ x \\
\quad\quad\quad\quad \psi_u\ n\ m = m \oplus n \\
\quad\quad\quad\quad \psi_d\ c\ n = n \ominus c
\end{array}
$$

□

Based on Lemma 9.1, we can make use the three algebraic properties in Chapter 5 for the parallelization of downwards accumulations specified with path homomorphisms.

9.3 Parallelizing XPath Query without Predicates

We start by considering parallelization of XPath queries without predicates. Our parallelization algorithm for XPath queries without predicates consists of the following four steps.

1. Generate an automaton from the input XPath query.

2. Map the automaton to mutual recursive functions on paths.

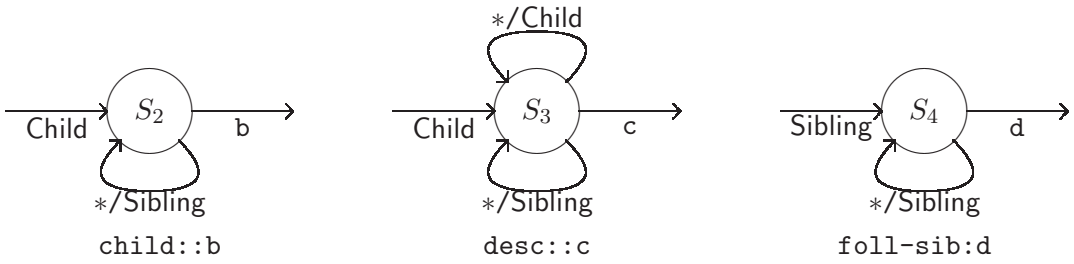


Figure 9.4. Fragments of automata for XPath queries.

3. Derive a path homomorphism by applying the tupling transformation to the mutual recursive functions.

4. Parallelize the path homomorphism using the tupled-ring property and derive a skeletal parallel program.

In the following of this section, we show the detail of the parallelization algorithm step by step. In this section, we use the following XPath query

```
/decs::a/child::b/foll-sib::c
```

as our running example.

Generate an Automaton from an XPath Query

XPath queries (without predicates) specified with three forward axes, `child`, `desc`, and `foll-sib`, is matched based on the information of the path from the root to a node on the binary-tree representation. This means that information of the path from the root to each node is essential to compute matching of an XPath query without predicates. When a XPath query matches to a path from the root, the XPath query returns the bottom-most node on the path.

It is well known that matching of regular expressions to strings can be done by automata. XPath queries without predicates are matched to paths and thus we formalize the matching by using automata. Figure 9.4 shows fragments of automata corresponding to each step of XPath queries. We can derive an automaton just by composing the fragments and putting the last state, which is the accepted state of the automaton. For our running example, we can map the three steps to the three fragments and combine them into an automaton shown in Figure 9.5.

Map the Automaton to Mutual Recursive Functions

We then derive mutual recursive functions which decide whether a path is accepted not. We can obtain mutual recursive functions from the automaton in the previous step. We map each state in the automaton to a function where we derive the definition of the

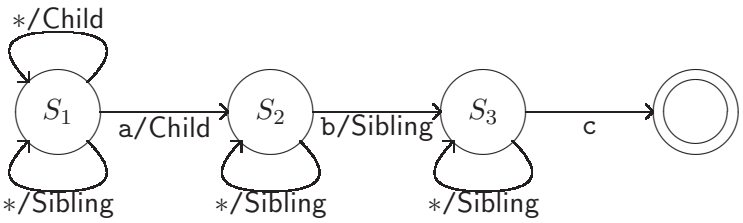


Figure 9.5. An automaton for XPath query `/desc::a/child::b/foll-sib::c`. The initial state is S_1. The transition a/Child occurs when the node is labeled as `a` and traversing to the left-most child. The transition b/Sibling occurs when the node is labeled as `b` and traversing to the right sibling.

function from the local shape of automaton around the corresponding state. Since the datatype of BPath is specified so that the root locates outside, the result of XPath query is given by the function corresponding to the initial state.

The singleton case represents the bottom-most element in the path, and thus the return value is True if there exists a transition to the accepted state of the automaton. For the child case, we recursively call functions if there exist transitions labeled as `x/Child` where `x` is the label of the node. The sibling case is similarly defined for siblings `x/Sibling` where `x` is the label of the node. The results of function calls are folded with operator $\vee$. The derivation rules of each function are summarized in Figure 9.6.

Returning to our running example, we map states to functions, S_1 to f_1, S_2 to f_2, and S_3 to f_3, and obtain the following mutual recursive functions.

$$
\begin{aligned}
f_1\ (\mathsf{Singleton}\ x) &= \mathsf{False} \\
f_1\ (\mathsf{Child}\ x\ c) &= f_1\ c \vee (x == \text{“a”} \wedge f_2\ c) \\
f_1\ (\mathsf{Sibling}\ x\ s) &= f_1\ s
\end{aligned}
$$

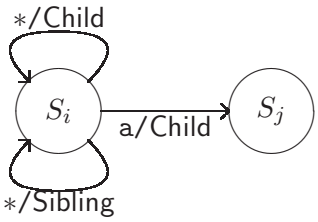
$$
\begin{aligned}
f_2\ (\mathsf{Singleton}\ x) &= \mathsf{False} \\
f_2\ (\mathsf{Child}\ x\ c) &= \mathsf{False} \\
f_2\ (\mathsf{Sibling}\ x\ s) &= f_2\ s \vee (x == \text{“b”} \wedge f_3\ s)
\end{aligned}
$$

$$
\begin{aligned}
f_3\ (\mathsf{Singleton}\ x) &= x == \text{“c”} \\
f_3\ (\mathsf{Child}\ x\ c) &= \mathsf{False} \\
f_3\ (\mathsf{Sibling}\ x\ s) &= f_3\ s
\end{aligned}
$$

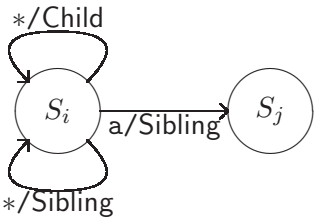
The XPath query is implemented by applying function f_1 to all the paths from the root. Let *query* be a function that applies the given XPath query, then the XPath query is implemented by the mutual recursive functions as

$$query(\texttt{/desc::a/child::b/foll-sib::c}) = \mathsf{map}_{\mathrm{b}}\ f_1\ f_1 \circ paths$$

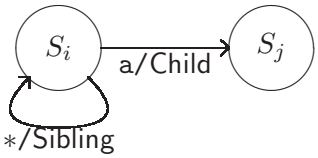
where f_1 is the function derived above.



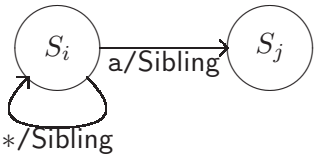
f_i (Child x c) $= f_i\ c \vee (x ==$ "a" $\wedge f_j\ c)$
f_i (Sibling x s) $= f_i\ s$



f_i (Child x c) $= f_i\ c$
f_i (Sibling x s) $= f_i\ s \vee (x ==$ "a" $\wedge f_j\ s)$



f_i (Child x c) $= x ==$ "a" $\wedge f_j\ c$
f_i (Sibling x s) $= f_i\ s$



f_i (Child x c) $=$ False
f_i (Sibling x s) $= f_i\ s \vee (x ==$ "a" $\wedge f_j\ s)$

Figure 9.6. The derivation rules on the data structure BPath.

Derive Path Homomorphisms by Tupling Transformation

The recursive functions obtained from automata may not be path homomorphisms since they often have calls of other functions in their body, which may involve multiple traversals on paths. For example, in the case of the running example, the body of function f_1 for the Child case includes a call of another function f_2, and the two calls of f_1 and f_2 to the left child cause multiple traversals.

To transform these mutual recursive functions to a path homomorphism, we apply the tupling transformation [62]. Informally speaking, for given mutual recursive functions $f_1, f_2, \ldots, f_m$, the tupling transformation generates a new function f defined as $f = (f_1 \vartriangle f_2 \vartriangle \cdots f_m)$. Mutual recursive functions derived from automata are transformed into a path homomorphism that manipulates tuples.

We can simplify the definition of the derived path homomorphism using the properties of operators: operator $\vee$ is both associative and commutative, and operator $\wedge$ distributes over $\vee$. As seen in Figure 9.6, function f_i corresponding to state S_i only includes function calls of f_i and f_j, both corresponding to states S_i and S_j, respectively. Therefore, after applying tupling transformation and simplifying functions, we obtain a path homomorphism defined in the following form:

$$
\begin{aligned}
f = \mathit{PathHom}(k_x, k_c, k_s)& \\
\quad \textbf{where } & k_x\ x = (b_{11}\ x, b_{12}\ x, \ldots, x_{1m}\ x) \\
& \quad k_c\ x\ (c_1, c_2, \ldots, c_m) \\
& \qquad = ((b_{21}\ x \wedge c_1) \vee (b'_{21}\ x \wedge c_2),\ (b_{22}\ x \wedge c_2) \vee (b'_{22}\ x \wedge c_3), \\
& \qquad\qquad \ldots,\ b_{2m}\ x \wedge c_m) \\
& \quad k_s\ x\ (s_1, s_2, \ldots, s_m) \\
& \qquad = ((b_{31}\ x \wedge c_1) \vee (b'_{31}\ x \wedge c_2),\ (b_{32}\ x \wedge c_2) \vee (b'_{32}\ x \wedge c_3), \\
& \qquad\qquad \ldots,\ b_{3m}\ x \wedge c_m)
\end{aligned}
$$

where functions b_{ij} and b'_{ij} are boolean functions (they may be constant functions).

Returning to our running example, by applying tupling transformation to functions f_1, f_2, and f_3, i.e. $f = (f_1 \vartriangle f_2 \vartriangle f_3)$, we obtain the following definition of f.

$$
\begin{aligned}
f\ (\mathsf{Singleton}\ x) &= (\mathsf{False}, \mathsf{False}, x == \text{“}\mathsf{c}\text{”}) \\
f\ (\mathsf{Child}\ x\ c) &= \mathbf{let}\ (c_1, c_2, c_3) = f\ c \\
&\quad\ \mathbf{in}\ \ (c_1 \vee (x == \text{“}\mathsf{a}\text{”} \wedge c_2), \mathsf{False}, \mathsf{False}) \\
f\ (\mathsf{Sibling}\ x\ s) &= \mathbf{let}\ (s_1, s_2, s_3) = f\ s \\
&\quad\ \mathbf{in}\ \ (s_1, s_2 \vee (x == \text{“}\mathsf{b}\text{”} \wedge s_3), s_3)
\end{aligned}
$$

The function f is in fact a path homomorphism. The following definition gives the three parameter functions of the path homomorphism.

$$
\begin{aligned}
&f = \mathit{PathHom}(k_x, k_c, k_s) \\
&\quad \mathbf{where}\ k_x\ x = (\mathsf{False}, \mathsf{False}, x == \text{“}\mathsf{c}\text{”}) \\
&\qquad\quad k_c\ x\ (c_1, c_2, c_3) = (c_1 \vee (x == \text{“}\mathsf{a}\text{”} \wedge c_2), \mathsf{False}, \mathsf{False}) \\
&\qquad\quad k_s\ x\ (s_1, s_2, s_3) = (s_1, s_2 \vee (x == \text{“}\mathsf{b}\text{”} \wedge s_3), s_3)
\end{aligned}
$$

Now we can obtain a skeletal program that matches the XPath query for all the node as follows.

$$
\begin{aligned}
&\mathit{query}(\texttt{/desc::a/child::b/foll-sib::c})\ t \\
&\quad = (\mathsf{map}_\mathrm{b}\ f_1\ f_1 \circ \mathit{paths})\ t \\
&\quad = \quad \{f_1 = \mathit{fst} \circ (f_1 \vartriangle f_2 \vartriangle f_3) = \mathit{fst} \circ \mathit{PathHom}(k_x, k_c, k_s)\} \\
&\qquad (\mathsf{map}_\mathrm{b}\ (\mathit{fst} \circ \mathit{PathHom}(k_x, k_c, k_s))\ (\mathit{fst} \circ \mathit{PathHom}(k_x, k_c, k_s)) \circ \mathit{paths})\ t \\
&\quad = \quad \{\text{distributivity of } \mathsf{map}_\mathrm{b} \text{ over function composition } \circ\} \\
&\qquad (\mathsf{map}_\mathrm{b}\ \mathit{fst}\ \mathit{fst} \circ \mathsf{map}_\mathrm{b}\ (\mathit{PathHom}(k_x, k_c, k_s))\ (\mathit{PathHom}(k_x, k_c, k_s)) \circ \mathit{paths})\ t \\
&\quad = \quad \{\text{definition of downwards accumulation}\} \\
&\qquad \mathsf{map}_\mathrm{b}\ \mathit{fst}\ \mathit{fst}\ (\mathsf{zipwith}_\mathrm{b}\ (\lambda x\ c.c\ (f_x\ x))\ (\lambda x\ c.c\ (f_x\ x))\ t\ (\mathsf{dAcc}_\mathrm{b}\ (f'_c, f'_s)\ \mathit{id}\ t) \\
&\qquad \mathbf{where}\ f'_c\ c\ x = c \circ f_c\ x \\
&\qquad\qquad\quad f'_s\ c\ x = c \circ f_s\ x
\end{aligned}
$$

Parallelize Path Homomorphism

We have derived a path homomorphism in the previous step, now we parallelize it using the algebraic properties discussed in Chapter 5.

In fact, in the definition of functions operators $\wedge$ and $\vee$ construct a commutative semiring. Furthermore, no pair of recursive calls are joined with $\wedge$, that is, there are no occurrences of $(f_j\ c \wedge f_k\ c)$ or $(f_j\ s \wedge f_k\ s)$ for any j and k. With these observations, we can apply the tupled-ring property (Theorem 5.13) to derive auxiliary functions for the dAcc_b skeleton. We insert False as the coefficients of non occurring recursive calls in the transformation from the functions k_c and k_s defined above into their linear polynomial form. Note that False is the zero-element of the commutative semiring $\{\mathsf{Bool}, \vee, \wedge\}$.

After deriving the linear polynomial forms for two functions k_c and k_s, we can apply Theorem 5.13 with Lemma 9.1 and successfully obtain skeletal parallel programs for XPath queries.

$$
\begin{array}{l}
\mathit{query}(\texttt{/desc::a/child::b/foll-sib::c})\ t \\
\quad = \mathsf{zipwith}_{\mathrm{b}}\ k'_x\ k'_x\ t\ (\mathsf{dAcc}_{\mathrm{b}}\ \langle \phi_l, \phi_r, \times'_{\wedge,\vee}, \times'_{\wedge,\vee} \rangle_{\mathrm{d}}\ \boldsymbol{I}\ t) \\
\qquad \textbf{where}\ k'_x\ x \begin{pmatrix} \_ & c_{12} & c_{13} & \_ \\ \_ & c_{22} & c_{23} & \_ \\ \_ & \_ & c_{33} & \_ \\ \_ & \_ & \_ & \_ \end{pmatrix} = (c_{13} \wedge x == \text{“c”}) \\
\qquad\qquad \phi_l\ x = \begin{pmatrix} \_ & x == \text{“a”} & \mathsf{False} & \_ \\ \_ & \mathsf{False} & \mathsf{False} & \_ \\ \_ & \_ & \mathsf{False} & \_ \\ \_ & \_ & \_ & \_ \end{pmatrix} \\
\qquad\qquad \phi_r\ x = \begin{pmatrix} \_ & \mathsf{False} & \mathsf{False} & \_ \\ \_ & \mathsf{True} & x == \text{“b”} & \_ \\ \_ & \_ & \mathsf{True} & \_ \\ \_ & \_ & \_ & \_ \end{pmatrix} \\
\qquad\qquad \boldsymbol{I} = \begin{pmatrix} \_ & \mathsf{False} & \mathsf{False} & \_ \\ \_ & \mathsf{True} & \mathsf{False} & \_ \\ \_ & \_ & \mathsf{True} & \_ \\ \_ & \_ & \_ & \_ \end{pmatrix} \\
\qquad\qquad \boldsymbol{M} \times'_{\wedge,\vee} \boldsymbol{N} = \boldsymbol{N} \times_{\wedge,\vee} \boldsymbol{M}
\end{array}
$$

Figure 9.7. Derived skeletal parallel program for XPath query `/desc::a/child::b/foll-sib::c`.

For our running example, we can verify that the function k_c and k_s can be written in the form of tupled linear polynomial functions as follows.

$$
\begin{array}{l}
k_c\ x\ (c_1, c_2, c_3) = (c'_1, c'_2, c'_3) \\
\quad \textbf{where}\ c'_1 = (\mathsf{True} \wedge c_1) \vee (x == \text{“a”} \wedge c_2) \vee (\mathsf{False} \wedge c_3) \vee \mathsf{False} \\
\qquad\quad c'_2 = (\mathsf{False} \wedge c_1) \vee (\mathsf{False} \wedge c_2) \vee (\mathsf{False} \wedge c_3) \vee \mathsf{False} \\
\qquad\quad c'_3 = (\mathsf{False} \wedge c_1) \vee (\mathsf{False} \wedge c_2) \vee (\mathsf{False} \wedge c_3) \vee \mathsf{False} \\
\\
k_s\ x\ (s_1, s_2, s_3) = (s'_1, s'_2, s'_3) \\
\quad \textbf{where}\ s'_1 = (\mathsf{True} \wedge s_1) \vee (\mathsf{False} \wedge s_2) \vee (\mathsf{False} \wedge s_3) \vee \mathsf{False} \\
\qquad\quad s'_2 = (\mathsf{False} \wedge s_1) \vee (\mathsf{True} \wedge s_2) \vee (x == \text{“b”} \wedge s_3) \vee \mathsf{False} \\
\qquad\quad s'_3 = (\mathsf{False} \wedge s_1) \vee (\mathsf{False} \wedge s_2) \vee (\mathsf{True} \wedge s_3) \vee \mathsf{False}
\end{array}
$$

Based on these definitions, we can derive skeletal parallel program for the XPath query as shown in Figure 9.7. The program is optimized by fusing the $\mathsf{map}_{\mathrm{b}}$ and $\mathsf{zipwith}_{\mathrm{b}}$ skeleton. In the program in Figure 9.7, eleven values of matrices denoted by _ do not need to be computed during the computation of $\mathsf{dAcc}_{\mathrm{b}}$ skeleton. In other words, we only compute the other five values. We can find these values using the code generator for tupled-ring property in Section 7.3.

Now we summarize the discussion so far as the following lemma.

Lemma 9.2 *XPath queries in Figure 9.1 defined without predicates can be parallelized.*

Proof. By transforming the automata into mutual recursive functions and applying tupling transformation, we obtain a path homomorphism defined on commutative semiring $\{\mathsf{Bool}, \vee, \wedge\}$. It follows from tupled-ring property (Theorem 5.13), that the XPath query without predicates can be implemented efficiently in parallel. □

9.4 Parallelizing XPath Query Inside of Single Predicate

We then consider parallelization of XPath queries defined inside of a single predicate. Our parallelization algorithm accepts an XPath query consisting of a nodetest with an unnested predicate specified with three forward axes and returns a parallel tree homomorphism. It consists of the following four steps in almost the same manner as the previous section. Difference from the parallelization algorithm in the previous section is that the query is matched to subtrees and that the derived program is a tree homomorphism instead of a path homomorphism.

1. Generate an automaton from the input XPath query.
2. Map the automaton to mutual recursive functions on subtrees.
3. Derive a tree homomorphism by applying the tupling transformation to the mutual recursive functions.
4. Parallelize the tree homomorphism based on the tupled-ring property.

In this section, we use the following XPath query as our running example,

`a[desc::b/child::*]` ,

which is specified with a predicate `[desc::b/child::*]` and the nodetest of the target node `a`.

Generate an Automaton from an XPath Query

The query of a single predicate defined with three forward axes tries matching with the information of subtrees of the binary-tree representation. When the query matches to a subtree of a node, the root node of the subtree is the result of the query.

We transform the XPath query with a predicate to an automaton. First we flatten the XPath query, for example,

`a[desc::b/child::*]` $\Longrightarrow$ `a/desc::b/child::*` .

We then construct an automaton by combining the fragments shown in Figure 9.4. Since the leftmost nodetest, e.g., a in the running example, is specified without axes, we have no transition to the corresponding initial state.

For our running example, after flattening the XPath query with a predicate, we obtain an automaton shown in Figure 9.8 by composing corresponding fragments.

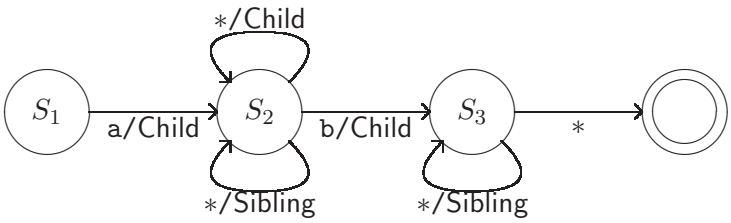


Figure 9.8. Automaton for "`a/desc::b/child::*`".

Map the Automaton to Mutual Recursive Functions

We then derive mutual recursive functions by mapping the automata to functions. In the case of an XPath query with a single predicate, the query is matched to subtrees and the matching succeeds if there exists such a path in the subtrees that matches to the predicate. Therefore, we implement mutual recursive functions that match the automaton to all the paths from a node to the descendants. Here by using the dynamic programming technique, we can implement these mutual recursive functions where they traverse a binary tree once in the bottom-up manner.

We can derive mutual recursive functions systematically as follows. First we assign a function to each state in the automaton. We then define the function based on the local shape of the automaton by the rules shown in Figure 9.9. Since every leaf in the binary-tree representation of XML trees is a dummy, the functions always return False for leaves. For internal nodes, we recursively call functions if there exist transitions labeled Child or Sibling in the automaton, and we fold the results with $\vee$. The result of a query is given by the result of the function corresponding to the initial state.

For our running example, we map the states to functions, S_1 to f_1, S_2 to f_2, and S_3 to f_3, and then obtain the following mutual recursive functions on binary trees.

$$
\begin{aligned}
f_1\ (\mathsf{BLeaf}\ a) &= \mathsf{False} \\
f_1\ (\mathsf{BNode}\ c\ x\ s) &= x == \text{“}\mathtt{a}\text{”} \wedge f_2\ c \\
\\
f_2\ (\mathsf{BLeaf}\ a) &= \mathsf{False} \\
f_2\ (\mathsf{BNode}\ c\ x\ s) &= f_2\ c \vee f_2\ s \vee (x == \text{“}\mathtt{b}\text{”} \wedge f_3\ c) \\
\\
f_3\ (\mathsf{BLeaf}\ a) &= \mathsf{False} \\
f_3\ (\mathsf{BNode}\ c\ x\ s) &= f_3\ s \vee \mathsf{True}
\end{aligned}
$$

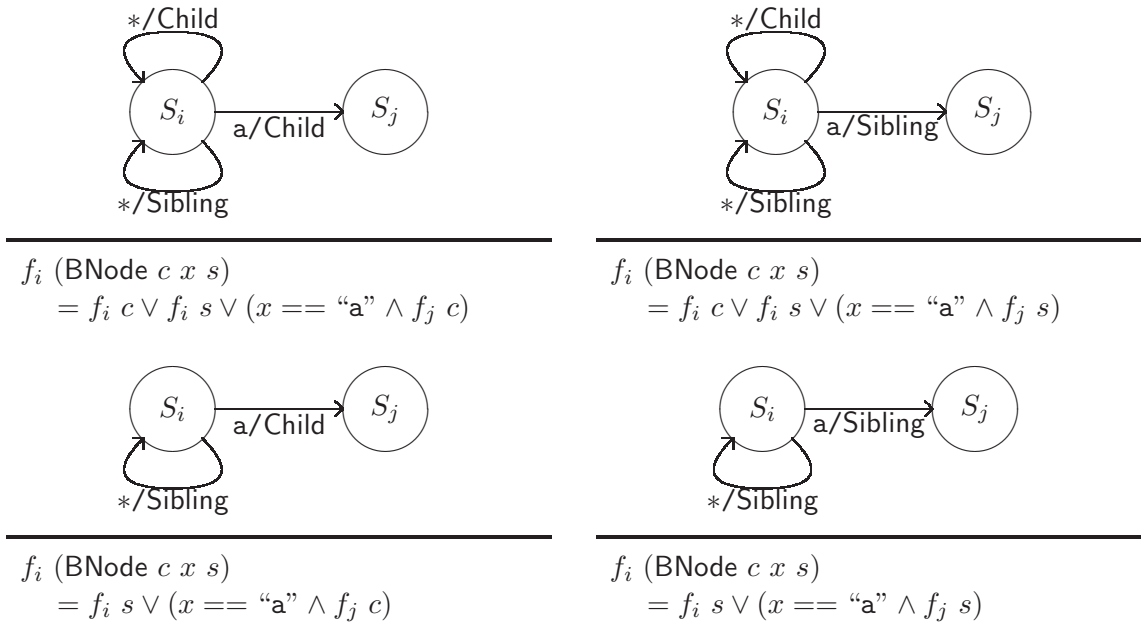
Using these mutual recursive functions, we can implement the XPath query as

$$query(\texttt{a[desc::b/child::*]})\ t = (\mathsf{map}_\mathrm{b}\ f_1\ f_1 \circ subtrees)\ t$$

where the function f_1 is the derived above.

Derive Tree Homomorphisms by Tupling Transformation

The mutual recursive functions obtained from automata may not be tree homomorphisms since functions may have function calls of other functions and involve multiple traversals



f_i (BNode c x s)
$= f_i\ c \vee f_i\ s \vee (x ==$ "a" $\wedge f_j\ c)$

f_i (BNode c x s)
$= f_i\ c \vee f_i\ s \vee (x ==$ "a" $\wedge f_j\ s)$

f_i (BNode c x s)
$= f_i\ s \vee (x ==$ "a" $\wedge f_j\ c)$

f_i (BNode c x s)
$= f_i\ s \vee (x ==$ "a" $\wedge f_j\ s)$

Figure 9.9. The derivation rules for the segment of automata for XPath query with a predicate.

on binary trees. In our running example, function f_2 called for (BNode c x s) not only has a call of another function f_3 but also has calls of functions f_2 and f_3 for the left child, which cause multiple traversals. To obtain tree homomorphisms, we again apply the tupling transformation [62] to the mutual recursive functions.

We can derive a tree homomorphism defined in the following form by using the knowledge of algebraic properties on $\vee$ and $\wedge$. Here b_{ij} and b'_{ij} represent boolean functions that may be a constant function.

$$
\begin{array}{l}
(f_1 \vartriangle f_2 \vartriangle \cdots \vartriangle f_m) \\
\quad = \mathit{TreeHom}(k_l, k_n) \\
\quad\quad \textbf{where}\ k_l\ a = (\mathsf{False}, \mathsf{False}, \ldots, \mathsf{False}) \\
\quad\quad\quad\quad k_n\ (c_1, c_2, \ldots, c_m)\ x\ (s_1, s_2, \ldots, s_m) \\
\quad\quad\quad\quad\quad = (b_{1x} \vee (b_{1c} \wedge c_1) \vee (b'_{1c} \wedge c_2) \vee (b_{1s} \wedge s_1) \vee (b'_{1s} \wedge s_2), \\
\quad\quad\quad\quad\quad\quad b_{2x} \vee (b_{2c} \wedge c_2) \vee (b'_{2c} \wedge c_3) \vee (b_{2s} \wedge s_2) \vee (b'_{2s} \wedge s_3), \\
\quad\quad\quad\quad\quad\quad\quad \vdots \\
\quad\quad\quad\quad\quad\quad b_{mx} \vee (b_{mc} \wedge c_m) \vee (b_{ms} \wedge s_m))
\end{array}
$$

In the case of our running example, by tupling functions f_1, f_2, and f_3, i.e., $f = (f_1 \vartriangle f_2 \vartriangle f_3)$, we obtain the following tree homomorphism. In deriving tree homomorphism, we simplified $s_3 \vee$ True to True.

$$
\begin{array}{l}
f = \mathit{TreeHom}(k_l, k_n) \\
\quad \textbf{where}\ k_l\ a = (\mathsf{False}, \mathsf{False}, \mathsf{False}) \\
\quad\quad k_n\ (c_1, c_2, c_3)\ x\ (s_1, s_2, s_3) \\
\quad\quad\quad = (x ==$ "a" $\wedge\, c_2, c_2 \vee s_2 \vee (x ==$ "b" $\wedge\, c_3), \mathsf{True})
\end{array}
$$

Using this tree homomorphism, we obtain a skeletal program for the XPath query as follows.

$$
\begin{aligned}
&\mathit{query}(\texttt{a[desc::b/child::*]})\ t\\
&\quad = \mathsf{map}_\mathrm{b}\ f_1\ f_1 \circ \mathit{subtrees}\\
&\quad = \quad \{f_1 = \mathit{fst} \circ (f_1 \vartriangle f_2 \vartriangle f_3) = \mathit{fst} \circ (\mathit{TreeHom}(k_l, k_n))\}\\
&\qquad \mathsf{map}_\mathrm{b}\ (\mathit{fst} \circ (\mathit{TreeHom}(k_l, k_n)))\ (\mathit{fst} \circ (\mathit{TreeHom}(k_l, k_n))) \circ \mathit{subtrees}\\
&\quad = \quad \{\text{distributivity of } \mathsf{map}_\mathrm{b} \text{ over function composition } \circ\}\\
&\qquad \mathsf{map}_\mathrm{b}\ \mathit{fst}\ \mathit{fst} \circ \mathsf{map}_\mathrm{b}\ (\mathit{TreeHom}(k_l, k_n))\ (\mathit{TreeHom}(k_l, k_n)) \circ \mathit{subtrees}\\
&\quad = \quad \{\text{definition of the upwards accumulation}\}\\
&\qquad \mathsf{map}_\mathrm{b}\ \mathit{fst}\ \mathit{fst} \circ \mathsf{uAcc}_\mathrm{b}\ k_n \circ \mathsf{map}_\mathrm{b}\ k_l\ \mathit{id}
\end{aligned}
$$

Parallelize Tree Homomorphism

The last step is to parallelize the derived tree homomorphism. Now again, noting that the operators used in the tree homomorphism construct a commutative ring $\{\mathsf{Bool}, \vee, \wedge\}$, we can derive auxiliary functions for the uAcc_b skeleton by applying the tupled-ring property (Theorem 5.12).

For our running example, we can verify that the derived tree homomorphism satisfies the tupled ring property by transforming the function k_n to a tuple of bi-linear polynomial functions.

$$
\begin{aligned}
&k_n\ (c_1, c_2, c_3)\ x\ (s_1, s_2, s_3) = (x_1, x_2, x_3)\\
&\quad \textbf{where}\ x_1 = (\mathsf{False} \wedge c_1) \vee (x == \text{“}\mathsf{a}\text{”} \wedge c_2) \vee (\mathsf{False} \wedge c_3) \vee \mathsf{False}\\
&\qquad\qquad x_2 = (\mathsf{False} \wedge c_1) \vee (\mathsf{True} \wedge c_2) \vee (x == \text{“}\mathsf{b}\text{”} \wedge c_3) \vee s_2\\
&\qquad\qquad x_3 = (\mathsf{False} \wedge c_1) \vee (\mathsf{False} \wedge c_2) \vee (\mathsf{False} \wedge c_3) \vee \mathsf{True}
\end{aligned}
$$

$$
\begin{aligned}
&k_n\ (c_1, c_2, c_3)\ x\ (s_1, s_2, s_3) = (y_1, y_2, y_3)\\
&\quad \textbf{where}\ y_1 = (\mathsf{False} \wedge s_1) \vee (\mathsf{False} \wedge s_2) \vee (\mathsf{False} \wedge s_3) \vee (x == \text{“}\mathsf{a}\text{”} \wedge c_2)\\
&\qquad\qquad y_2 = (\mathsf{False} \wedge s_1) \vee (\mathsf{True} \wedge s_2) \vee (\mathsf{False} \wedge s_3) \vee (x == \text{“}\mathsf{b}\text{”})\\
&\qquad\qquad y_3 = (\mathsf{False} \wedge s_1) \vee (\mathsf{False} \wedge s_2) \vee (\mathsf{True} \wedge s_3) \vee \mathsf{True}
\end{aligned}
$$

Based on these definitions of bi-linear polynomial functions, we can derive the auxiliary functions for the uAcc_b skeleton and obtain a skeletal parallel program as shown in Figure 9.10. In the program, we need not compute the eight values in the matrices denoted as _ and the eight values denoted as c_{ij} should be computed. We can perform this optimization using the code generator developed in Section 7.3.

Now we summarize the discussion in this section as the following lemma.

Lemma 9.3 *XPath queries specified inside of a single predicate can be computed in parallel using parallel skeletons.*

Proof. By transforming the automaton into mutual recursive functions and applying the tupling transformation to the mutual recursive functions, we obtain a tree homomorphism defined on commutative semiring $\{\mathsf{Bool}, \vee, \wedge\}$. It follows from tupled-ring property (Theorem 5.12) that the XPath query with a single predicate can be implemented efficiently in parallel. □

$$query(\texttt{a[desc::b/child::*]}) = \mathsf{map_b}\ fst\ fst \circ \mathsf{uAcc_b}\langle \phi, \psi_n, \psi_l, \psi_r \rangle_\mathrm{u} \circ \mathsf{map_b}\ k_l\ id$$

$$\textbf{where } \phi\ x = \left(\begin{pmatrix} \mathsf{True} & \mathsf{False} & \mathsf{False} & \mathsf{False} \\ - & - & \mathsf{False} & \mathsf{False} \\ - & - & \mathsf{True} & \mathsf{False} \\ - & - & - & - \end{pmatrix}, x \right)$$

$$\psi_n \begin{pmatrix} l_1 \\ l_2 \\ l_3 \end{pmatrix} \left(\begin{pmatrix} c_{11} & c_{12} & c_{13} & c_{14} \\ - & - & c_{23} & c_{24} \\ - & - & c_{33} & c_{34} \\ - & - & - & - \end{pmatrix}, b \right) \begin{pmatrix} r_1 \\ r_2 \\ r_3 \end{pmatrix}$$

$$= \textbf{let } \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = k_n\ b \begin{pmatrix} l_1 \\ l_2 \\ l_3 \end{pmatrix} \begin{pmatrix} r_1 \\ r_2 \\ r_3 \end{pmatrix}$$

$$\textbf{in } \begin{pmatrix} (c_{11} \wedge x_1) \vee (c_{12} \wedge x_2) \vee (c_{13} \wedge x_3) \vee c_{14} \\ x_2 \vee (c_{23} \wedge x_3) \vee c_{24} \\ \mathsf{True} \end{pmatrix}$$

$$\psi_l\ (\boldsymbol{M}^l, b^l)\ (\boldsymbol{M}^n, b^n) \begin{pmatrix} r_1 \\ r_2 \\ r_3 \end{pmatrix}$$

$$= \left(\boldsymbol{M}^n \times_{\wedge,\vee} \begin{pmatrix} \mathsf{False} & b^n == \text{“a”} & \mathsf{False} & \mathsf{False} \\ - & - & b^n == \text{“b”} & r_2 \\ - & - & \mathsf{False} & \mathsf{True} \\ - & - & - & - \end{pmatrix} \times_{\wedge,\vee} \boldsymbol{M}^l, b^l \right)$$

$$\psi_r \begin{pmatrix} l_1 \\ l_2 \\ l_3 \end{pmatrix} (\boldsymbol{M}^n, b^n)\ (\boldsymbol{M}^r, b^r)$$

$$= \left(\boldsymbol{M}^n \times_{\wedge,\vee} \begin{pmatrix} \mathsf{False} & \mathsf{False} & \mathsf{False} & b^n == \text{“a”} \wedge l_2 \\ - & - & \mathsf{False} & b^n == \text{“b”} \\ - & - & \mathsf{True} & \mathsf{True} \\ - & - & - & - \end{pmatrix} \times_{\wedge,\vee} \boldsymbol{M}^r, b^r \right)$$

$$k_l\ x = \begin{pmatrix} \mathsf{False} \\ \mathsf{False} \\ \mathsf{False} \end{pmatrix}$$

Figure 9.10. Skeletal parallel program for XPath query `a[desc::b/child::*]`.

9.5 Parallelizing More Complex XPath Queries

So far we have developed two procedures for parallelizing simple XPath queries, XPath queries without predicate and XPath queries inside of a single predicate, we now highlight how to extend them to parallelize more general XPath queries. The overall parallelization algorithm consists of the following steps.

1. Transform the input XPath query into core XPath queries defined in Figure 9.1.
2. Repeat while there exist predicates.
   2.1. Choose unnested predicates. For each unnested predicate, derive a parallel tree homomorphism and perform matching with the upwards accumulation.
   2.2. Zip the result trees to the original tree and remove the predicates from the query.
3. For the query that is defined without predicates, derive a parallel path homomorphism and perform matching with the downwards accumulation.

In the following of this section, we illustrate the parallelization algorithm using the following XPath query

```
/desc::b/parent::a/child::c[foll-sib::d]
```

as our running example. Figure 9.11 illustrates the execution of the parallel program derived by the parallelization algorithm.

Transform the input XPath Query

Our parallelization algorithm accepts XPath queries defined as in Figure 9.1. When the input XPath query is defined with other axes such as reverse axes, we remove them by Olteanu et al.'s algorithm [103].

In our example, there exists a reverse-axis "`parent`". We can remove it by transforming the query into the following equivalent XPath query

```
/desc::a[child::b]/child::c[foll-sib::d] ,
```

which has no reverse axis.

Derive Parallel Tree Homomorphism for Each Predicate

For each predicate in the XPath query, we derive a parallel tree homomorphism with the parallelization techniques in Section 9.4. We perform matching of the query inside of the predicate to each subtree by using the upwards accumulation.

If predicates are nested (a predicate is in another predicate), then we compute the matching of the innermost one first. If two or more predicates are not included in each other, we can compute them simultaneously.

For our running example, we derive tree homomorphisms for subqueries `a[child::b]` and `c[foll-sib::d]`, and compute the tree homomorphisms for each subtree with the upwards accumulations.

Zip Result Trees to the Original Tree

We then associate the results of the matching of predicates to the original XML tree. We then modify the query by replacing the predicate with a special nodetest that matches to the attached values.

For our running example, after computing the upwards accumulations for two predicates, we obtain a zipped tree as shown in Figure 9.11 (c). We then replace the predicates with two special nodetests ($\lambda(x, y, z).y == \mathsf{True}$) and ($\lambda(x, y, z).z == \mathsf{True}$) that matches to the second or third value, respectively.

$$\texttt{/desc::}(\lambda(x, y, z).y == \mathsf{True})\texttt{/child::}(\lambda(x, y, z).z == \mathsf{True})$$

Derive Parallel Path Homomorphism for Overall XPath

Finally we evaluate the rest of the XPath query that has no predicate. We derive a parallel path homomorphism with the parallelization techniques in Section 9.3, and evaluate it on every path from the root using the downwards accumulation.

For our running example, we derive a path homomorphism for the XPath query given in the previous step

$$\texttt{/desc::}(\lambda(x, y, z).y == \mathsf{True})\texttt{/child::}(\lambda(x, y, z).z == \mathsf{True})$$

in the same way as the derivation for

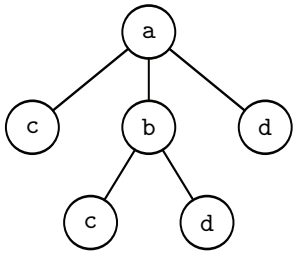
```
/desc::y/child::z
```

except for the handling of nodetests. We can perform the overall XPath query by the downwards accumulation.

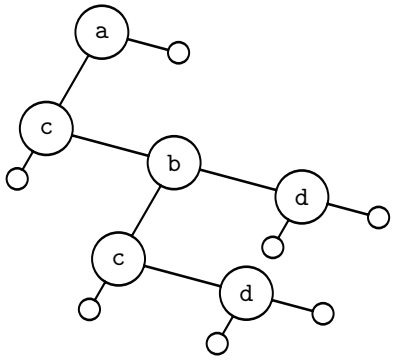
Theorem 9.1 *The XPath query defined in Figure 9.1 can be computed in parallel.*

Proof. As we have discussed so far, by processing the predicates from the innermost one we can finally obtain a query without predicates. From Lemmas 9.2 and 9.3, the matchings of queries inside of unnested predicates and queries without predicates can be done in parallel. Therefore, we can also perform the overall query in parallel. □

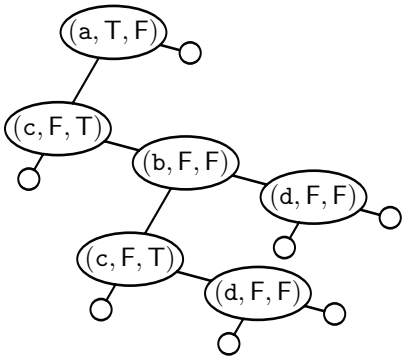
Figure 9.12 shows the derived skeletal parallel program for our example XPath query.



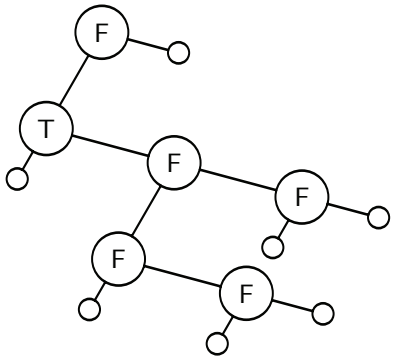
(a) Original XML tree



(b) Binary-tree representation of the XML tree



(c) After attaching results of predicates



(d) The result of query on binary-tree representation

Figure 9.11. Illustration of computational steps of the derived parallel program for XPath query `/desc::a[child::b]/child::c[foll-sib::d]`.

$$
\begin{array}{l}
query(\texttt{/desc::a[child::b]/child::c[foll-sib::d]})\ bt \\
\quad = \textbf{let}\ bt'_1 = query(\texttt{a[child::b]}) \\
\qquad\quad\ bt'_2 = query(\texttt{c[foll-sib::d]}) \\
\qquad\quad\ zt = \mathsf{zip3_b}\ bt\ bt'_1\ bt'_2 \\
\qquad \textbf{in}\ query(\texttt{/desc::}(\lambda(x,y,z).y == \mathsf{True})\texttt{/child::}(\lambda(x,y,z).z == \mathsf{True}))
\end{array}
$$

Figure 9.12. Outline of a skeletal parallel program for a complex XPath query.

9.6 Short Summary

In this chapter, we have shown a parallel implementation algorithm for a subset of XPath queries based on the theory of constructive algorithmics and skeletal parallel programs. The main idea is deriving automata corresponding to the input XPath query followed by the mapping of the automata to two kinds of homomorphisms on binary trees. The tupled-ring property on the commutative semiring $\{\mathsf{Bool}, \vee, \wedge\}$ contributes to the parallelization of the derived homomorphisms.

The derived parallel program may be optimized if we can apply the branch-and-bound technique often used in implementing efficient sequential programs for queries. This is an interesting challenge both in theory and in practice.

Chapter 10

Related Work

10.1 Tree Contraction Algorithms

Tree contraction algorithms, the idea for which was first proposed by Miller and Reif [98], are very important parallel algorithms for efficient manipulation of trees. Many researchers have devoted considerable effort to developing efficient implementations of tree contraction algorithms for various parallel models [2, 8, 10, 35, 38, 46, 66, 94, 95, 125]. Among them, Gibbons and Rytter developed a cost-optimal algorithm on a concurrent-read concurrent-write (CREW) PRAM [46]; Abrahamson et al. developed a cost-optimal and practical algorithm on an EREW PRAM [2]; Miller and Reif showed implementations on hypercubes or related networks [94,95]; more efficient implementations were recently developed [8,125] for symmetric multiprocessors and chip-level multiprocessing.

The formalization of the ternary-tree representation and tree associativity in Chapter 3 is general in terms of tree division, so the skeletal parallel programs developed in this thesis can be implemented using these tree contraction algorithms.

Many types of tree programs have been described using the tree contraction algorithms [10,35,46,57,86,99–102]. Many of these programs, however, compute a single value instead of tupled values at each contraction step. For example, Cole and Vishkin [35] and He [57] developed parallel algorithms based on the finiteness of the domain, e.g. for the minimum covering-set problem and the maximum independent-set problem. Though the maximum independent-set problem is a simpler version of the party planning problem, their approaches are not applicable to development of parallel programs for the party planning problem. In this sense, the tupled-ring property described in Section 5.2.4 is a more useful property for developing parallel programs. Miller and Teng [101] proposed a method for developing parallel programs for computational trees with *min* and *max* functions in addition to $+$ and $\times$ by focusing on the algebraic properties. They also extended their idea to the evaluation of computational circuits (trees in which each leaf has a value and each internal node has an operator) with finite-sized matrices [100, 102]. Though their approaches are interesting in theory, they impose many restrictions on the operators associated with each internal node. The condition presented in this thesis for

tree contraction algorithms is concise and practical — any computation on the internal nodes can be defined.

10.2 Parallel Computing on Rose Trees and Nested Lists

In contrast to parallel computing for regular lists and matrices, parallel computing for rose trees has been considered as irregular parallelism, and several approaches have been taken to tackle this difficult problem. They can be classified as

- approaches based on binary-tree representation, and.
- approaches based on tree flattening.

Parallel tree contraction algorithms are now the basis for efficient parallel computation on trees. Though the original idea proposed by Miller and Reif [98] did not limit the shapes of trees to binary, many researchers have developed more efficient tree contraction algorithms based on the assumption of binary trees [2, 8, 35, 95]. Several researchers developed parallel algorithms after representing rose trees as binary trees to utilize the efficient parallel tree contraction algorithms. Cole and Vishkin [35], Diks and Hagerup [41], Skillicorn [119], and a previous paper of ours [86] represented rose trees as binary trees by using dummy nodes to expand children of internal nodes. Though these representations work well for specifying bottom-up and top-down computations, they are poor at specifying intra-sibling computation. In this thesis, another binary-tree representation [36] is used that enables us to formalize and implement intra-sibling computation as well. As shown by the example using the prefix numbering problem in Chapter 4, this intra-sibling computation plays a important role in the manipulation of rose trees.

Tree flattening is another data-parallel approach, and there have been several studies [20, 70, 73, 124]. For example, the NESL programming language [20–22] provides computational patterns for nested computations, and NESL programs execute efficiently in parallel in shared-memory environments. Palmer et al. discussed how nested computations can be compiled using this paradigm [104]. We can treat rose trees as (infinitely) nested lists. Keller and Chakravarty developed a parallel programming technique for single-nested lists in which the lists are flattened to lists with size information. Takahashi et al. [124] examined the problem of flattened lists with flags. Kakehi et al. [69, 70] developed an efficient parallel implementation of reductions for more deeply nested cases. In contrast to the implementation using binary-tree representation described in this thesis, their implementation would be more efficient if the rose tree is rather flat.

10.3 Parallel Tree Skeletons

Though trees are important data structures, writing general and efficient parallel programs for manipulating trees is complicated by their irregular structures. This calls for helpful methods for parallel programming on trees, and here the skeletal approach is a promising paradigm. Deldari et al. [39,40] designed and implemented parallel skeletons for constructive solid geometry as domain-specific skeletons. Skillicorn [119] first formalized a set of binary-tree skeletons based on constructive algorithmics [15, 18, 67] for general-purpose tree skeletons. The implementations of these binary-tree skeletons based on the tree contraction algorithms have been investigated [47, 50, 87]. Several studies have been done based on parallel tree skeletons for several applications [63, 88, 120, 121]. However, no libraries have been implemented that support the parallel tree skeletons formalized by these studies.

Gibbons [49] investigated generic computational patterns based on the theory of constructive algorithmics for general trees and general recursive types, and Skillicorn [119] and Ahn et al. [3] gave specifications for skeletons. However, the patterns by Skillicorn lack expressiveness and the patterns by Ahn et al. do not support parallel implementations. These problems were tackled in this thesis, and a new set of rose-tree skeletons with efficient implementation was proposed. The rose-tree skeletons presented in Chapter 4 are not only theoretically simple but also practically expressive.

10.4 Automatic and Systematic Parallelization

Automatic parallelization of programs is quite a big challenge, and there have been several studies on automatic parallelization of loops over arrays. Fisher and Ghuloum [43] developed a parallelization system for loops on arrays based on the isomorphism of the shape of program code. Lu and Mellor-Crummey [79] developed more powerful pattern-matching and code-generation mechanisms for distributed memory environments. Xu et al. [128,129] focused on not only associativity but also distributivity to derive parallel programs from user programs for lists and arrays and developed an automatic parallelization system. These studies succeeded in automatically generating efficient parallel code from user sequential code. Though there have been several studies on parallelizing loops, to the best of our knowledge, there is no (semi-)automatic parallelization system that can be applied to tree structures.

This thesis is also related to systematic derivation of parallel programs. Systematic parallelization has been actively studied in the framework of skeletal parallel programming, and many studies have been done [32,52,63,93,117] on lists and arrays. Several researchers have studied systematic parallelization for trees. Skillicorn formalized five primitive computational patterns [118], and tree reduction is one of them. Ahn and Han [3] and my group [88] developed systematic methods for decomposing complex recursive programs

into combinations of the primitive patterns. Chapter 5 addresses the generation of efficient parallel programs for the case of general trees and the bridging of the gap between sequential and parallel implementations of skeletons.

10.5 Skeletal Environments

Many skeleton libraries have been developed [4, 11, 25, 37, 75, 110] to help users enjoy the benefit of skeletal parallel programming. Among them, Google's MapReduce programming model [37] is the most well known and successful one. SkeTo was constructed on a data parallel programming model with a solid foundation based on constructive algorithmics and can provide data parallel skeletons for a wide variety of distributed data structures; it supports trees in addition to arrays and matrices. In addition, SkeTo has a skeleton *accumulate* [61,65] that abstracts a general and good combination of data parallel skeletons.

The implementation of the SkeTo library was inspired by the development of the Muesli skeleton library [75,76]. The SkeTo library is a practical result of research on constructive algorithmics. One of its important features is systematic program optimization by fusion transformation. This transformation merges two successive function calls into a single one and eliminates the overhead of both function calls and of the generation of intermediate data structures passed between them. SkeTo is equipped with automatic fusion transformation based on the idea of shortcut deforestation [51] with modifications that make it adaptable to parallel data structures [61]. Use of shortcut deforestation enables the number of transformation rules to be reduced and simplifies the implementation of SkeTo. This is in sharp contrast to other transformation approaches [5, 54] with a large number of transformation rules. This simple optimization mechanism by fusion transformation is SkeTo's distinguishing feature, one that has not been implemented in other systems.

Chapter 11

Conclusion

This thesis has investigated the principles and practices of parallel programming with tree skeletons based on constructive algorithmics theory.

11.1 Principles of Parallel Programming for Trees

Many studies have been conducted on tree contraction algorithms for efficient parallel programming for trees, but it is still difficult for sequential programmers to use tree contraction algorithms. One obstacle is that the formalization is not sufficient for deriving parallel programs by using tree contraction algorithms.

To overcome this obstacle, in this thesis, parallel tree skeletons were formalized for binary trees and for rose trees based on the theory of constructive algorithmics. The parallel rose-tree skeletons are straightforward extensions of binary-tree skeletons. The parallel tree skeletons provide users with a sequential interface and a parallel implementation. To utilize the parallel implementation of the skeletons, users must have their programs to meet some conditions imposed on the parallel skeletons. In this thesis, we formalized the conditions for parallel computations on binary trees based on the balanced ternary-tree representation of binary trees. Our conditions placed on the parallel skeletons can capture a wider class of applications than those captured by Skillicorn's specification.

These specifications and a solid theory enabled us to develop a systematic methodology for deriving skeletal parallel programs from given sequential recursive programs (Chapter 5). Some of the steps in this derivation can be done automatically (Section 7.3). These systematic and automatic derivations should help users to develop parallel programs for manipulating trees more easily.

11.2 Practices of Parallel Programming for Trees

We implemented the SkeTo parallel skeleton library to make the theories underlying parallel computing for trees and other data structures more widely applicable. The SkeTo

library is a skeleton library implemented in C++ and MPI for distributed-memory parallel computers. It is unique in the implementation of skeletons for trees and the use of an optimization mechanism based on fusion transformation, which are contributions of this thesis.

In the implementation of binary-tree skeletons, we carefully designed the data structure and algorithms to enhance the locality and load balance. Since these two properties are uncertain in the case of trees, a cost model was developed for this implementation, and the data distribution was analyzed. This implementation of binary-tree skeletons showed good speedup.

To prove the effectiveness of the parallel tree skeletons and the skeletal parallel programming paradigm, we developed parallelization algorithms for two classes of applications: the maximum marking problems and XPath queries. As far as we are aware, there have been no parallel algorithms for these two classes of applications, meaning that skeletal parallel programming is effective not only for facilitating parallel programming but also for deriving new parallel algorithms.

11.3 Future Directions

The design of parallel tree skeletons based on the theory of constructive algorithmics is theoretically beautiful and powerful, as exemplified by this thesis. But it also has some drawbacks. The most significant drawback of the current definition of parallel tree skeletons is that the shape of the data structures manipulated in programs cannot be changed without the cost of additional redistribution. The same problem can be seen in the design of list and matrix skeletons. To make parallel skeletons applicable to more problems, it is necessary to overcome this change in the shape of data structures. In relation to this problem, bridging one data structure to another, for example, flattening transformation from trees to lists, is also useful in developing programs.

In many applications, trees often expand; for example, in game programming, a game tree grows from a single node up to a huge tree. The data-parallel paradigm is not suitable for such cases — a task-parallel paradigm should be applied. Future work includes investigating parallel computing for trees using task-parallel programming model based on the theory of constructive algorithmics as was done in this thesis for data-parallel computing on trees.

Bibliography

[1] David Abrahams and Aleksey Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*. Addison-Wesley, 2004.

[2] Karl R. Abrahamson, N. Dadoun, David G. Kirkpatrick, and Teresa M. Przytycka. A simple parallel tree contraction algorithm. *Journal of Algorithms*, Vol. 10, No. 2, pp. 287–302, June 1989.

[3] Joonseon Ahn and Taisook Han. An analytical method for parallelization of recursive functions. *Parallel Processing Letters*, Vol. 10, No. 1, pp. 87–98, March 2000.

[4] Marco Aldinucci, Sonia Campa, Pierpaolo Ciullo, Massimo Coppola, Silvia Magini, Paolo Pesciullesi, Laura Potiti, Roberto Ravazzolo, Massimo Torquati, Marco Vanneschi, and Corrado Zoccolo. The implementation of ASSIST, an environment for parallel and distributed programming. In Harald Kosch, László Böszörményi, and Hermann Hellwagner, editors, *Euro-Par 2003. Parallel Processing, 9th International Euro-Par Conference, Klagenfurt, Austria, August 26–29, 2003. Proceedings*, Vol. 2790 of *Lecture Notes in Computer Science*, pp. 712–721. Springer, 2003.

[5] Marco Aldinucci, Sergei Gorlatch, Christian Lengauer, and Susanna Pelagatti. Towards parallel programming by transformation: the FAN skeleton framework. *Parallel Algroithms and Applications*, Vol. 16, No. 2–3, pp. 87–121, March 2001.

[6] Carlos E. R. Alves, Edson Cáceres, and Siang W. Song. BSP/CGM algorithms for maximum subsequence and maximum subarray. In Dieter Kranzlmüller, Péter Kacsuk, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 11th European PVM/MPI Users' Group Meeting, Budapest, Hungary, September 19–22, 2004, Proceedings*, Vol. 3241 of *Lecture Notes in Computer Science*, pp. 139–146. Springer, 2004.

[7] Bruno Bacci, Marco Danelutto, Salvatore Orlando, Susanna Pelagatti, and Marco Vanneschi. P^3L: a structured high-level parallel language and its structured support. *Concurrency: Practice and Experience*, Vol. 7, No. 3, pp. 225–255, May 1995.

[8] 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*, Vol. 2552 of *Lecture Notes in Computer Science*, pp. 63–78. Springer, 2002.

[9] Sung Eun Bae and Tadao Takaoka. Algorithms for the problem of k maximum sums and a VLSI algorithm for the k maximum subarrays problem. In *7th International Symposium on Parallel Architectures, Algorithms, and Networks (I-SPAN 2004), 10–12 May 2004, Hong Kong, SAR, China*, pp. 247–253. IEEE Computer Society, 2004.

[10] Raja P. K. Banerjee, Vineet Goel, and Amar Mukherjee. Efficient parallel evaluation of CSG tree using fixed number of processors. In *ACM Symposium on Solid Modeling Foundations and CAD/CAM Applications, May 19–21, 1993, Montreal, Canada*, pp. 137–146, 1993.

[11] Anne Benoit, Murray Cole, Stephen Gilmore, and Jane Hillston. Flexible skeletal programming with eSkel. In José C. Cunha and Pedro D. Medeiros, editors, *Euro-Par 2005, Parallel Processing, 11th International Euro-Par Conference, Lisbon, Portugal, August 30–September 2, 2005, Proceedings*, Vol. 3648 of *Lecture Notes in Computer Science*, pp. 761–770. Springer, 2005.

[12] Jon Bentley. *Programming Pearls*. Addison-Wesley, 1st edition, 1986.

[13] 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/`.

[14] Marshall W. Bern, Eugene L. Lawler, and A. L. Wong. Linear-time computation of optimal subgraphs of decomposable graphs. *Journal of Algorithms*, Vol. 8, No. 2, pp. 216–235, June 1987.

[15] Richard S. Bird. An introduction to the theory of lists. In Manfred Broy, editor, *Logic of Programming and Calculi of Discrete Design*, Vol. 36 of *NATO ASI Series F*, pp. 5–42. Springer, October 1987.

[16] Richard S. Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall Series in Computer Science. Prentice Hall, 2nd edition, April 1998.

[17] Richard S. Bird. Maximum marking problems. *Journal of Functional Programming*, Vol. 11, No. 4, pp. 411–424, July 2001.

[18] Richard S. Bird and Oege de Moor. *Algebra of Programming.* International Series in Computer Science. Prentice Hall, September 1996.

[19] Guy E. Blelloch. Scans as primitive operations. *IEEE Transactions on Computers*, Vol. 38, No. 11, pp. 1526–1538, November 1989.

[20] Guy E. Blelloch. *Vector Models for Data-Parallel Computing.* MIT Press, 1990.

[21] Guy E. Blelloch. NESL: A nested data-parallel language. Technical Report CMU-CS-95-170, School of Computer Science, Carnegie Mellon University, September 1995.

[22] 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*, Vol. 21, No. 4, pp. 4–14, April 1994.

[23] Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie, and Jérôme Siméon, editors. *XQuery 1.0: An XML Query Language.* W3C Candidate Recommendation, June 2006.
Available from `http://www.w3.org/TR/xquery/`.

[24] Richard B. Borie, R. Gary Parker, and Craig A. Tovey. Automatic generation of linear-time algorithms from predicate calculus descriptions of problems on recursively constructed graph families. *Algorithmica*, Vol. 7, No. 5&6, pp. 555–581, 1992.

[25] George Horatiu Botorog and Herbert Kuchen. Skil: An imperative language with algorithmic skeletons for efficient distributed programming. In *The 5th International Symposium on High Performance Distributed Computing (HPDC '96), August 6–9, 1996, Syracuse, NY, USA, Proceedings.* IEEE Computer Society, 1996.

[26] Silvia Breitinger, Rita Loogen, Yolanda Ortega-Mallén, and Ricardo Peña. The Eden coordination model for distributed memory systems. In *1997 Workshop on High-Level Programming Models and Supportive Environments (HIPS '97), 1 April 1997, Geneva, Switzerland*, Vol. 1123 of *Lecture Notes in Computer Science*, pp. 120–124. Springer, 1997.

[27] Bernard Chazelle. A theorem on polygon cutting with applications. In *23rd Annual Symposium on Foundations of Computer Science, 3–5 November 1982, Chicago, Illinois, USA*, pp. 339–349. IEEE Press, 1982.

[28] Shigeru Chiba. A metaobject protocol for C++. In *OOPSLA'95, Tenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications, 15–19 October 1995, Austin, Texas, USA, Proceedings*, Vol. 30 of *SIGPLAN Notices*, pp. 285–299. ACM Press, 1995.

[29] Wei-Ngan Chin, John Darlington, and Yike Guo. Parallelizing conditional recurrences. 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 I*, Vol. 1123 of *Lecture Notes in Computer Science*, pp. 579–586. Springer, 1996.

[30] Wei-Ngan Chin and Zhenjiang Hu. Towards a modular program derivation via fusion and tupling. In Don S. Batory, Charles Consel, and Walid Taha, editors, *Generative Programming and Component Engineering, ACM SIGPLAN/SIGSOFT Conference, GPCE 2002, Pittsburgh, PA, USA, October 6–8, 2002, Proceedings*, Vol. 2487 of *Lecture Notes in Computer Science*, pp. 140–155. Springer, 2002.

[31] Wei-Ngan Chin, Siau-Cheng Khoo, Zhenjiang Hu, and Masato Takeichi. Deriving parallel codes via invariants. In Jens Palsberg, editor, *Static Analysis, 7th International Symposium, SAS 2000, Santa Barbara, CA, USA, June 29–July 1, 2000, Proceedings*, Vol. 1824 of *Lecture Notes in Computer Science*, pp. 75–94. Springer, 2000.

[32] 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*, pp. 153–162. IEEE Computer Society, 1998.

[33] Murray Cole. *Algorithmic Skeletons: Structural Management of Parallel Computation*. Research Monographs in Parallel and Distributed Computing. MIT Press, 1989.

[34] Murray Cole. Parallel programming with list homomorphisms. *Parallel Processing Letters*, Vol. 5, No. 2, pp. 191–203, June 1995.

[35] Richard Cole and Uzi Vishkin. The accelerated centroid decomposition technique for optimal parallel tree evaluation in logarithmic time. *Algorithmica*, Vol. 3, pp. 329–346, March 1988.

[36] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, second edition, September 2001.

[37] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *6th Symposium on Operating System Design and Implementation (OSDI2004), December 6–8, 2004, San Francisco, California, USA*, pp. 137–150, 2004.

[38] Frank K. H. A. Dehne, Afonso Ferreira, Edson Cáceres, Siang W. Song, and Alessandro Roncato. Efficient parallel graph algorithms for coarse-grained multicomputers and BSP. *Algorithmica*, Vol. 33, No. 2, pp. 183–200, January 2002.

[39] Hossain Deldari, John R. Davy, and Peter M. Dew. Parallel CSG, skeletons and performance modeling. In *Proceedings of the Second Annual CSI Computer Conference (CSICC'96)*, pp. 115–122, 1996.

[40] Hossain Deldari, John R. Davy, and Peter M. Dew. A skeleton for parallel CSG with a performance model. Technical report, School of computer studies research report series, University of Leeds, 1997.

[41] Krzysztof Diks and Torben Hagerup. More general parallel tree contraction: Register allocation and broadcasting in a tree. *Theorecital Computer Science*, Vol. 203, No. 1, pp. 3–29, August 1998.

[42] Kento Emoto, Zhenjiang Hu, Kazuhiko Kakehi, and Masato Takeichi. A compositional framework for developing parallel programs on two-dimensional arrays. *International Journal of Parallel Programming*, 2007. Available online.

[43] Allan L. Fisher and Anwar M. Ghuloum. Parallelizing complex scans and reductions. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI), Orlando, Florida, June 20–24, 1994*, Vol. 29 of *SIGPLAN Notices*, pp. 135–146. ACM Press, 1994.

[44] Maarten M. Fokkinga. Tupling and mutumorphisms. *The Squiggolist*, Vol. 1, No. 4, pp. 81–82, June 1990.

[45] Hillel Gazit, Gary L. Miller, and Shang-Hua Teng. Optimal tree contraction in EREW model. In *Proceedings of the Princeton Workshop on Algorithms, Architectures, and Technical Issues for Models of Concurrent Computation*, pp. 139–156, 1987.

[46] Alan Gibbons and Wojciech Rytter. An optimal parallel algorithm for dynamic expression evaluation and its applications. In Kesav V. Nori, editor, *Foundations of Software Technology and Theoretical Computer Science, Sixth Conference, New Delhi, India, December 18–20, 1986, Proceedings*, Vol. 241 of *Lecture Notes in Computer Science*, pp. 453–469. Springer, 1986.

[47] Jeremy Gibbons. Computing downwards accumulations on trees quickly. In Gopal Gupta, George Mohay, and Rodney Topor, editors, *Proceedings of the 16th Australian Computer Science Conference*, pp. 685–691, 1993.

[48] Jeremy Gibbons. Upwards and downwards accumulations on trees. In Richard S. Bird, Carroll Morgan, and Jim Woodcock, editors, *Mathematics of Program Construction, Second International Conference, Oxford, U.K., June 29–July 3, 1992, Proceedings*, Vol. 669 of *Lecture Notes in Computer Science*, pp. 122–138. Springer, 1993.

[49] Jeremy Gibbons. Generic downward accumulations. *Science of Computer Programming*, Vol. 37, No. 1–3, pp. 37–65, May 2000.

[50] Jeremy Gibbons, Wentong Cai, and David B. Skillicorn. Efficient parallel algorithms for tree accumulations. *Science of Computer Programming*, Vol. 23, No. 1, pp. 1–18, October 1994.

[51] Andrew J. Gill, John Launchbury, and Simon L. Peyton Jones. A short cut to deforestation. In *FPCA '93 Conference on Functional Programming Languages and Computer Architecture. Copenhagen, Denmark, 9–11 June 1993*, pp. 223–232. ACM Press, 1993.

[52] 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*, Vol. 1124 of *Lecture Notes in Computer Science*, pp. 401–408. Springer, 1996.

[53] Sergei Gorlatch and Susanna Pelagatti. A transformational framework for skeletal programs: Overview and case study. In José D. P. Rolim, Frank Mueller, Albert Y. Zomaya, Fikret Erçal, Stephan Olariu, Binoy Ravindran, Jan Gustafsson, Hiroaki Takada, Ronald A. Olsson, Laxmikant V. Kalé, Peter H. Beckman, Matthew Haines, Hossam A. ElGindy, Denis Caromel, Serge Chaumette, Geoffrey Fox, Yi Pan, Keqin Li, Tao Yang, G. Ghiola, Gianni Conte, Luigi V. Mancini, Dominique Méry, Beverly A. Sanders, Devesh Bhatt, and Viktor K. Prasanna, editors, *Parallel and Distributed Processing, 11 IPPS/SPDP'99 Workshops Held in Conjunction with the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing, San Juan, Puerto Rico, USA, April 12–16, 1999, Proceedings*, Vol. 1586 of *Lecture Notes in Computer Science*, pp. 123–137. Springer, 1999.

[54] 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*, pp. 492–499. IEEE Computer Society, 1999.

[55] Ashish Kumar Gupta and Dan Suciu. Stream processing of XPath queries with predicates. In Alon Y. Halevy, Zachary G. Ives, and AnHai Doan, editors, *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9–12, 2003*, pp. 419–430. ACM Press, 2003.

[56] Frank Harary. *Graph Theory*. Westview Press, November 1994.

[57] Xin He. Efficient parallel algorithms for solving some tree problems. In *24th Allerton Conference on Communication, Control and Computing*, pp. 777–786, 1986.

[58] Zhenjiang Hu, Hideya Iwasaki, and Masato Takeichi. Construction of list homomorphisms by tupling and fusion. In Wojciech Penczek and Andrzej Szalas, editors, *Mathematical Foundations of Computer Science 1996, 21st International Symposium, MFCS'96, Cracow, Poland, September 2–6, 1996, Proceedings*, Vol. 1113 of *Lecture Notes in Computer Science*, pp. 407–418. Springer, 1996.

[59] Zhenjiang Hu, Hideya Iwasaki, and Masato Takeichi. Formal derivation of parallel program for 2-dimensional maximum segment sum problem. 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 I*, Vol. 1123 of *Lecture Notes in Computer Science*, pp. 553–562. Springer, 1996.

[60] Zhenjiang Hu, Hideya Iwasaki, and Masato Takeichi. Formal derivation of efficient parallel programs by construction of list homomorphisms. *ACM Transactions on Programming Languages and Systems*, Vol. 19, No. 3, pp. 444–461, May 1997.

[61] Zhenjiang Hu, Hideya Iwasaki, and Masato Takeichi. An accumulative parallel skeleton for all. In Daniel Le Métayer, editor, *Programming Languages and Systems, 11th European Symposium on Programming, ESOP 2002, held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8–12, 2002, Proceedings*, Vol. 2305 of *Lecture Notes in Computer Science*, pp. 83–97. Springer, 2002.

[62] Zhenjiang Hu, Hideya Iwasaki, Masato Takeichi, and Akihiko Takano. Tupling calculation eliminates multiple data traversals. In *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP '97), Amsterdam, The Netherlands, June 9–11, 1997*, Vol. 32 of *SIGPLAN Notices*, pp. 164–175. ACM Press, 1997.

[63] 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*, pp. 85–94. University of Aarhus, 1999. Technical report BRICS-NS-99-1.

[64] IBM. Cell broadband engine architecture. Available online, October 2006.

[65] Hideya Iwasaki and Zhenjiang Hu. A new parallel skeleton for general accumulative computations. *International Journal of Parallel Programming*, Vol. 32, No. 5, pp. 389–414, October 2004.

[66] Gustedt Jens. Communication and memory optimized tree contraction and list ranking. Technical report, INRIA, Unité de recherche, Rhône-Alpes, Montbonnot-Saint-Martin, FRANCE, December 2000.

[67] Johan Jeuring. *Theories for Algorithm Calculation*. Ph.D thesis, Faculty of Science, Utrecht University, 1993. Parts of the thesis appeared in *the Lecture Notes of the STOP 1992 Summerschool on Constructive Algorithmics*.

[68] Kazuhiko Kakehi, Zhenjiang Hu, and Masato Takeichi. List homomorphism with accumulation. In Walter Dosch and Roger Y. Lee, editors, *Proceedings of the ACIS Fourth International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD'03), October 16–18, 2003, Lübeck, Germany*, pp. 250–259. ACIS, 2003.

[69] Kazuhiko Kakehi, Kiminori Matsuzaki, and Kento Emoto. Efficient parallel tree reductions on distributed memory environments. In Yong Shi, Dick van Albada, Jack Dongarra, and Peter Sloot, editors, *ICCS 2007: 7th International Conference, Beijing, China, May 27–30, 2007, Proceedings, Part II*, Vol. 4488 of *Lecture Notes in Computer Science*, pp. 601–608. Springer, 2007.

[70] Kazuhiko Kakehi, Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi. A unified approach toward nested parallelism. 日本ソフトウェア科学会 第 21 回大会 論文集, September 2004.

[71] Björn Karlssonn. *Beyond the C++ Standard Library : An Introduction to Boost*. Addison-Wesley, August 2005.

[72] Michael Kay, editor. *XSL Transformations (XSLT) Version 2.0*. W3C Candidate Recommendation, June 2006.
Available from `http://www.w3.org/TR/xslt20/`.

[73] Gabriele Keller and Manuel M. T. Chakravarty. Flattening trees. In David J. Pritchard and Jeff Reeve, editors, *Euro-Par '98 Parallel Processing, 4th International Euro-Par Conference, Southampton, UK, September 1–4, 1998, Proceedings*, Vol. 1470 of *Lecture Notes in Computer Science*, pp. 709–719. Springer, 1998.

[74] Peter M. Kogge and Harold S. Stone. A parallel algorithm for the efficient solution of a general class of recurrence equations. *IEEE Transactions on Computers*, Vol. 22, No. 8, pp. 786–793, 1973.

[75] Herbert Kuchen. A skeleton library. In Burkhard Monien and Rainer Feldmann, editors, *Euro-Par 2002, Parallel Processing, 8th International Euro-Par Conference Paderborn, Germany, August 27–30, 2002, Proceedings*, Vol. 2400 of *Lecture Notes in Computer Science*, pp. 620–629. Springer, 2002.

[76] Herbert Kuchen and Jörg Striegnitz. Higher-order functions and partial applications for a C++ skeleton library. In José E. Moreira, Geoffrey Fox, and Vladimir Getov, editors, *Proceedings of the 2002 Joint ACM-ISCOPE Conference on Java Grande 2002, Seattle, Washington, USA, November 3–5, 2002*, pp. 122–130. ACM Press, 2002.

[77] Sivaramakrishnan Lakshmivarahan and Sudarshan K. Dhall. *Parallel Computing Using the Prefix Problem*. Oxford University Press, July 1994.

[78] Charles E. Leiserson and Bruce M. Maggs. Communication-efficient parallel algorithms for distributed random-access machines. *Algorithmica*, Vol. 3, pp. 53–77, 1988.

[79] Bo Lu and John M. Mellor-Crummey. Compiler-optimization of implicit reductions for distributed memory multiprocessors. In *12th International Parallel Processing Symposium/9th Symposium on Parallel and Distributed Processing (IPPS/SPDP '98), March 30–April 3, 1998, Orlando, Florida, USA, Proceedings*, pp. 42–51. IEEE Computer Society, 1998.

[80] Kevin Lü, Yuanling Zhu, Wenjun Sun, Shouxun Lin, and Jianping Fan. Parallel processing XML documents. In *International Database Engineering and Applications Symposium (IDEAS 2002), Proceedings*, pp. 96–105. IEEE Press, 2002.

[81] Kiminori Matsuzaki. Balanced ternary structure for parallel computing. Poster Presentation at the Fourth Workshop on Programming Structured Documents, the University of Tokyo, December 2005.

[82] Kiminori Matsuzaki. Balanced ternary-tree representation for parallel computing on trees. Presentation at Informal Workshop on Skeletal Parallel Programming, the University of Tokyo, March 2006.

[83] Kiminori Matsuzaki. Efficient implementation of tree accumulations on distributed-memory parallel computers. In Yong Shi, Dick van Albada, Jack Dongarra, and Peter Sloot, editors, *ICCS 2007: 7th International Conference, Beijing, China, May 27–30, 2007, Proceedings, Part II*, Vol. 4488 of *Lecture Notes in Computer Science*, pp. 609–616. Springer, 2007.

[84] Kiminori Matsuzaki and Zhenjiang Hu. Efficient implementation of tree skeletons on distributed-memory parallel computers. Technical Report METR 2006-65, Department of Mathematical Informatics, Graduate School of Information Science and Technology, the University of Tokyo, December 2006.

[85] Kiminori Matsuzaki, Zhenjiang Hu, Kazuhiko Kakehi, and Masato Takeichi. Systematic derivation of tree contraction algorithms. In Sergei Gorlatch, editor, *CMPP 2004, 4th International Workshop on Constructive Methods for Parallel Programming*, pp. 109–123. Westfälische Wilhelms-universität Münster, July 2004.

[86] Kiminori Matsuzaki, Zhenjiang Hu, Kazuhiko Kakehi, and Masato Takeichi. Systematic derivation of tree contraction algorithms. *Parallel Processing Letters*, Vol. 15, No. 3, pp. 321–336, September 2005.

[87] Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi. Implementation of parallel tree skeletons on distributed systems. In *The Third Asian Workshop on Programming Languages and Systems, APLAS'02, Shanghai Jiao Tong University, Shanghai, China, November 29–December 1, 2002, Proceedings*, pp. 258–271, 2002.

[88] Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi. Parallelization with tree skeletons. In Harald Kosch, László Böszörményi, and Hermann Hellwagner, editors, *Euro-Par 2003. Parallel Processing, 9th International Euro-Par Conference, Klagenfurt, Austria, August 26–29, 2003. Proceedings*, Vol. 2790 of *Lecture Notes in Computer Science*, pp. 789–798. Springer, 2003.

[89] Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi. Design and implementation of general tree skeletons. Technical Report METR2005-30, Department of Mathematical Informatics, Graduate School of Information Science and Technology, the University of Tokyo, 2005.

[90] Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi. Parallel skeletons for manipulating general trees. *Parallel Computing*, Vol. 32, No. 7–8, pp. 590–603, September 2006.

[91] Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi. Towards automatic parallelization of tree reductions in dynamic programming. In Phillip B. Gibbons and Uzi Vishkin, editors, *SPAA 2006: Proceedings of the 18th Annual ACM Symposium on Parallelism in Algorithms and Architectures, July 30–August 2, 2006, Cambridge, Massachusetts, USA*, pp. 39–48. ACM Press, 2006.

[92] 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*, Vol. 152 of *ACM International Conference Proceeding Series*. ACM Press, 2006.

[93] Kiminori Matsuzaki, Kazuhiko Kakehi, Hideya Iwasaki, Zhenjiang Hu, and Yoshiki Akashi. A fusion-embedded skeleton library. In Marco Danelutto, Marco Vanneschi, and Domenico Laforenza, editors, *Euro-Par 2004 Parallel Processing, 10th International Euro-Par Conference, Pisa, Italy, August 31–September 3, 2004, Proceedings*, Vol. 3149 of *Lecture Notes in Computer Science*, pp. 644–653. Springer, 2004.

[94] Ernst W. Mayr and Ralph Werchner. Optimal routing of parentheses on the hypercube. *Journal of Parallel and Distributed Computing*, Vol. 26, No. 2, pp. 181–192, April 1995.

[95] Ernst W. Mayr and Ralph Werchner. Optimal tree contraction and term matching on the hypercube and related networks. *Algorithmica*, Vol. 18, No. 3, pp. 445–460, July 1997.

[96] Lambert Meertens. First steps towards the theory of rose trees. CWI, Amsterdam; IFIP Working Group 2.1 working paper 592 ROM-25, 1988.

[97] Erik Meijer, Maarten M. Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In John Hughes, editor, *Functional Programming Languages and Computer Architecture, 5th ACM Conference, Cambridge, MA, USA, August 26–30, 1991, Proceedings*, Vol. 523 of *Lecture Notes in Computer Science*, pp. 124–144. Springer, 1991.

[98] 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*, pp. 478–489. IEEE Computer Society, 1985.

[99] Gary L. Miller and John H. Reif. Parallel tree contraction, part 2: Further applications. *SIAM Journal on Computing*, Vol. 20, No. 6, pp. 1128–1147, 1991.

[100] Gary L. Miller and Shang-Hua Teng. Dynamic parallel complexity of computational circuits. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing, 25–27 May 1987, New York City, NY, USA*, pp. 254–263. ACM Press, 1987.

[101] Gary L. Miller and Shang-Hua Teng. Tree-based parallel algorithm design. *Algorithmica*, Vol. 19, No. 4, pp. 369–389, December 1997.

[102] Gary L. Miller and Shang-Hua Teng. The dynamic parallel complexity of computational circuits. *SIAM Journal on Computing*, Vol. 28, No. 5, pp. 1664–1688, 1999.

[103] Dan Olteanu, Holger Meuss, Tim Furche, and François Bry. XPath: Looking forward. In Akmal B. Chaudhri, Rainer Unland, Chabane Djeraba, and Wolfgang Lindner, editors, *XML-Based Data Management and Multimedia Engineering — EDBT 2002 Workshops, EDBT 2002 Workshops XMLDM, MDDE, and YRWS, Prague, Czech Republic, March 24–28, 2002, Revised Papers*, Vol. 2490 of *Lecture Notes in Computer Science*, pp. 109–127. Springer, 2002.

[104] Daniel W. Palmer, Jan Prins, Siddhartha Chatterjee, and Rickard E. Faith. Piecewise execution of nested data-parallel programs. In Chua-Huang Huang, P. Sadayappan, Utpal Banerjee, David Gelernter, Alexandru Nicolau, and David A. Padua, editors, *Languages and Compilers for Parallel Computing, 8th International Workshop, LCPC'95, Columbus, Ohio, USA, August 10–12, 1995, Proceedings*, Vol. 1033 of *Lecture Notes in Computer Science*, pp. 346–361. Springer, 1996.

[105] Feng Peng and Sudarshan S. Chawathe. XPath queries on streaming data. In Alon Y. Halevy, Zachary G. Ives, and AnHai Doan, editors, *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9–12, 2003*, pp. 431–442. ACM Press, 2003.

[106] Kalyan S. Perumalla and Narsingh Deo. Parallel algorithms for maximum subsequence and maximum subarray. *Parallel Processing Letters*, Vol. 5, No. 3, pp. 367–373, 1995.

[107] Simon Peyton Jones and John Hughes. Report on the programming language Haskell 98: A non-strict, purely functional language.
Available from `http://www.haskell.org/`, February 1999.

[108] William M. Pottenger. The role of associativity and commutativity in the detection and transformation of loop-level parallelism. In *ICS '98, Proceedings of the 1998 International Conference on Supercomputing, July 13–17, 1998, Melbourne, Australia*, pp. 188–195. ACM Press, 1988.

[109] Ke Qiu and Selim G. Akl. Parallel maximum sum algorithms on interconnection networks. Technical report, Department of Computing and Information Science, Queen's University, Kingston, Ontario, September 1999.

[110] Fethi A. Rabhi and Sergei Gorlatch, editors. *Patterns and Skeletons for Parallel and Distributed Computing*. Springer, 2002.

[111] Keith H. Randall. *Cilk: Efficient Multithreaded Computing*. PhD thesis, MIT Department of Electrical Engineering and Computer Science, June 1998.

[112] John H. Reif, editor. *Synthesis of Parallel Algorithms*. Morgan Kaufmann Publishers, February 1993.

[113] 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*, pp. 114–121. ACM Press, 1994.

[114] Thomas Richert. Skil: Programming with algorithmic skeletons — a practical point of view. In *Draft Proceedings of the 12th International Workshop on Implementation of Functional Languages, Aachen, Germany*, pp. 15–30. Aachener Informatik Bericht, 2000. No. 2000-07.

[115] Isao Sasano, Zhenjiang Hu, and Masato Takeichi. Generation of efficient programs for solving maximum multi-marking problems. In Walid Taha, editor, *Semantics, Applications, and Implementation of Program Generation, Second International Workshop, SAIG 2001, Florence, Italy, September 6, 2001, Proceedings*, Vol. 2196 of *Lecture Notes in Computer Science*, pp. 72–91. Springer, 2001.

[116] Isao Sasano, Zhenjiang Hu, Masato Takeichi, and Mizuhito Ogawa. Make it practical: A generic linear-time algorithm for solving maximum-weightsum problems. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18–21, 2000*, Vol. 35 of *SIGPLAN Notices*, pp. 137–149. ACM Press, 2000.

[117] David B. Skillicorn. The Bird-Meertens formalism as a parallel model. In Janusz S. Kowalik and Lucio Grandinetti, editors, *Software for Parallel Computation*, Vol. 106 of *NATO ASI Series F*, pp. 120–133. Springer, April 1993.

[118] David B. Skillicorn. *Foundations of Parallel Programming*, Vol. 6 of *Cambridge International Series on Parallel Computation*. Cambridge University Press, 1994.

[119] David B. Skillicorn. Parallel implementation of tree skeletons. *Journal of Parallel and Distributed Computing*, Vol. 39, No. 2, pp. 115–125, December 1996.

[120] David B. Skillicorn. A parallel tree difference algorithm. *Information Processing Letters*, Vol. 60, No. 5, pp. 231–235, December 1996.

[121] David B. Skillicorn. Structured parallel computation in structured documents. *Journal of Univarsal Computer Science*, Vol. 3, No. 1, pp. 42–68, January 1997.

[122] Jörg Striegnitz. Making C++ ready for algorithmic skeletons. Technical Report FZJ-ZAM-IB-2000-08, ZAM, Forschungszentrum, Jülich, 2000.

[123] Supercomputing Technologies Group. Cilk-5.4 reference manual. Available from `http://supertech.csail.mit.edu/cilk/papers/`, June 2000.

[124] Tomonari Takahashi, Hideya Iwasaki, and Zhenjiang Hu. Efficient parallel skeletons for nested data structures. In Hamid R. Arabnia, editor, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA 2001, June 25–28, 2001, Las Vegas, Nevada, USA*, pp. 728–734. CSREA Press, 2001.

[125] Uzi Vishkin. A no-busy-wait balanced tree parallel algorithmic paradigm. In Gary Miller and Shang-Hua Teng, editors, *SPAA 2000: Proceedings of the 12th Annual ACM Symposium on Parallel Algorithms and Architectures, July 9–13, 2000, Bar Harbor, Maine, USA*, pp. 147–155. ACM Press, 2000.

[126] 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*, Vol. 300 of *Lecture Notes in Computer Science*, pp. 344–358. Springer, 1988.

[127] Zhaofang Wen. Fast parallel algorithms for the maximum sum problem. *Parallel Computing*, Vol. 21, No. 3, pp. 461–466, March 1995.

[128] Dana N. Xu. A type-based approach to parallelization. Master's thesis, Department of Computer Science, National University of Singapore, 2003.

[129] 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*, Vol. 3302 of *Lecture Notes in Computer Science*, pp. 197–212. Springer, 2004.

[130] 陳慰, 中野浩嗣, 増澤利光, 辻野嘉宏, 都倉信樹. 単純多角形内の最短経路を求める最適並列アルゴリズム. 電子情報通信学会論文誌 (D-I), Vol. J75-D-I, No. 12, pp. 814–825, 1991.

[131] 藤原暁宏, 陳慰, 増澤利光, 都倉信樹. 2 分木の平衡分解木を求めるコスト最適な並列アルゴリズム. 電子情報通信学会論文誌 (D-I), Vol. J83-D-I, No. 1, pp. 90–98, 2000.

[132] 明石芳樹, 松崎公紀, 岩崎英哉, 筧一彦, 胡振江. 最適化機構を持つ C++並列スケルトンライブラリ. コンピュータソフトウェア, Vol. 22, No. 3, pp. 214–222, 2005.

[133] 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.

Publication Lists

International Journals

1. Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi:
   Parallel Skeletons for Manipulating General Trees.
   Parallel Computing, Vol. 32, No. 7–8, pp. 590–603, Elsevier B.V., 2006.

2. Kiminori Matsuzaki, Zhenjiang Hu, Kazuhiko Kakehi, and Masato Takeichi:
   Systematic Derivation of Tree Contraction Algorithms.
   In *Parallel Processing Letters*, Vol. 15, No. 3, pp. 321–336, 2005.

Domestic Journals

3. 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.

4. 明石 良樹, 松崎 公紀, 岩崎 英哉, 筧 一彦, 胡 振江:
   最適化機構を持つ C++並列スケルトンライブラリ.
   コンピュータソフトウェア, Vol. 22, No. 3, pp. 214–222, 2005.

Refereed Papers (International Conferences and Workshops)

5. Kento Emoto, Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi:
   Domain-Specific Optimization Strategy for Skeleton Programs.
   In *13th International European Conference on Parallel and Distributed Computing (EuroPar 2007), IRISA, Rennes, France, August 28–31, 2007*, to appear.

6. Kazutaka Morita, Akimasa Morihata, Kiminori Matsuzaki, Zhenjiang Hu,
   and Masato Takeichi:
   Automatic Inversion Generates Divide-and-Conquer Parallel Programs.
   In *ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007), San Diego, CA, June 10–13, 2007*, to appear.

7. Kiminori Matsuzaki:
Efficient Implementation of Tree Accumulations on Distributed-Memory Parallel Computers.
In *Computational Science — ICCS 2007*, Lecture Notes in Computer Science 4488, pp. 609–616, Springer, 2007.

8. 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.

9. Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi:
Towards Automatic Parallelization of Tree Reductions in Dynamic Programming.
In *SPAA 2006: 18th Annual ACM Symposium on Parallelism in Algorithms and Architectures, July 30–August 2, 2006, Cambridge, Massachusetts, USA*, pp. 39–48, 2006.

10. 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.

11. 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.

12. Kiminori Matsuzaki, Kazuhiko Kakehi, Hideya Iwasaki, Zhenjiang Hu, and Yoshiki Akashi:
A Fusion-Embedded Skeleton Library.
In *10th International Euro-Par Conference, Pisa, Italy, August/September 2004, Proceedings*, Lecture Notes in Computer Science 3149, pp. 644–653, Springer, 2004.

13. Kiminori Matsuzaki, Zhenjiang Hu, Kazuhiko Kakehi, and Masato Takeichi:
Systematic Derivation of Tree Contraction Algorithms.
In *CMPP 2004: 4th International Workshop on Constructive Methods for Parallel Programming*, Technical Report of Westfalische Wilhelms-Univeersitat Munster, pp. 109–123, 2004.

14. Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi:
Parallelization with Tree Skeletons.
In *9th International Euro-Par Conference, Klagenfurt, Austria, August 2003*, Lecture Notes in Computer Science 2790, pp. 789–798, Springer, 2003.

Domestic Conference and Workshop or Non-Refereed Papers

15. 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.

16. Kiminori Matsuzaki and Noriyuki Ohkawa:
A Parallelizaiton Tool for Tree Reductions.
In *Proceedings of the 2nd DIKU-IST Joint Workshop on Foundations of Software*, Technical Report no. 06/07, Dept. of Computer Science, University of Copenhagen, 2006.

17. Kiminori Matsuzaki:
Parallel Tree Reduction and its Implementation in C++.
In *Proceedings of the 1st DIKU-IST Joint Workshop on Foundations of Software*, Technical Report no. 05/07, Dept. of Computer Science, University of Copenhagen, 2005.

18. 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.

19. 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.

20. 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.

21. Kazuhiko Kakehi, Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi:
A Uniform Approach toward Nested Parallelism.
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.

22. 明石 良樹, 松崎 公紀, 岩崎 英哉, 筧 一彦, 胡 振江:
最適化機構を持つ C++並列スケルトンライブラリ.
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.

23. Kiminori Matsuzaki, Kazuhiko Kakehi, Zhenjiang Hu, and Masato Takeichi:
Parallelizing Polytypic Programs with Accumulations.
日本ソフトウェア科学会第 20 回大会論文集, 愛知県立大学, 2003.

24. Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi:
Implementation of parallel tree skeletons on distributed systems.
In *Proceedings of the Third Asian Workshop on Programming Languages and Systems (APLAS '02)*, pp. 258–271, Shanghai, China, 2002.

25. 松崎 公紀, 胡 振江, 武市 正人:
分散メモリ型並列計算機上での木に対する並列スケルトンの実現.
日本ソフトウェア科学会第 19 回大会論文集, 産業技術総合研究所臨海副都心センター・日本科学未来館, 2002.

Technical Reports

26. 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.

27. Kiminori Matsuzaki and Zhenjiang Hu:
Efficient Implementation of Tree Skeletons on Distributed-Memory Parallel Computers.
Technical Report METR 2006-65, Department of Mathematical Engineering and Information Physics, the University of Tokyo, 2006.

28. 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.

29. Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi:
Design and Implementation of General Tree Skeletons.
Technical Report METR 2005-30, Department of Mathematical Engineering and Information Physics, the University of Tokyo, 2005.

30. Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi:
Parallelization with Tree Skeletons.
Technical Report METR 2003-21, Department of Mathematical Engineering and Information Physics, the University of Tokyo, 2003.

Presentations

31. 松崎 公紀, 胡 振江, 武市 正人:
Towards Automatic Parallelization of Tree Reductions in Dynamic Programming.
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),
カテゴリ 2, 2007.

32. 森田 和孝, 森畑 明昌, 松崎 公紀, 胡 振江, and 武市 正人:
Automatic Inversion Generates Divide-and-Conquer Parallel Programs.
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),
カテゴリ 2, 2007.

33. Kiminori Matsuzaki:
Parallel Tree Skeletons in SkeTo Library.
Presentation at Informal Workshop on Skeletal Parallel Programming, the University of Tokyo, 2006.

34. Kiminori Matsuzaki:
Balanced Ternary-Tree Representation for Parallel Computing on Trees.
Presentation at Informal Workshop on Skeletal Parallel Programming, the University of Tokyo, 2006.

35. Kiminori Matsuzaki:
Balanced Ternary Structure for Parallel Computing.
Poster Presentation at the Fourth Workshop on Programming Structured Documents, the University of Tokyo, 2005.

36. 松崎 公紀, 筧 一彦, 胡 振江, 武市 正人:
Systematic Derivation of Tree Contraction Algorithms.
第 7 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 2005),
カテゴリ 2, 2005.

37. Kiminori Matsuzaki:
Parallelization with Tree Skeletons.
Presentation at Workshop on Robust Software Construction (WRSC 2003), Hayama, Kanagawa, 2003.