MATHEMATICAL ENGINEERING
TECHNICAL REPORTS

Design and Implementation of General Tree Skeletons

Kiminori MATSUZAKI, Zhenjiang HU
and Masato TAKEICHI

METR 2005–30 October 2005

DEPARTMENT OF MATHEMATICAL INFORMATICS
GRADUATE SCHOOL OF INFORMATION SCIENCE AND TECHNOLOGY
THE UNIVERSITY OF TOKYO
BUNKYO-KU, TOKYO 113-8656, JAPAN

WWW page: http://www.i.u-tokyo.ac.jp/mi/mi-e.htm

Design and Implementation of General Tree Skeletons

Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi

Department of Mathematical Informatics, University of Tokyo
{kmatsu,hu,takeichi}@mist.i.u-tokyo.ac.jp

Abstract

Trees are important datatypes that are often used in representing structured data such as XML. Though trees are widely used in sequential programming, it is hard to write efficient parallel programs manipulating trees of arbitrary shapes, because of their irregular and ill-balanced structures. In this paper, we propose a solution for them based on the skeletal approach, in particular for general trees of arbitrary shapes, often called rose trees.

We formalize a set of skeletons (abstracted computational patterns) for rose trees based on the theory of Constructive Algorithmics. The formalization of the skeletons is an extension of those proposed for other data structures such as lists and binary trees. We then prove that the skeletons can be computed efficiently in parallel, by implementing each rose-tree skeleton in terms of parallel binary-tree skeletons for which an efficient parallel implementation is already known.

To encourage users to write efficient parallel programs in terms of parallel rose-tree skeletons, we propose a systematic method for deriving efficient skeletal programs from recursively defined sequential programs. We show the expressiveness of our parallel skeletons by three non-trivial examples. We also show a practical implementation of the rose-tree skeletons by adopting function objects and the template mechanism in C++.

As far as we are aware, we are the first who formalized and implemented a set of simple but expressive skeletons for rose trees.

1 Introduction

Trees are important datatypes that are often used in representing structured data such as XML. In recent years, the growth of computational power enables us to store huge data in the form of trees. This calls for systems and methods of manipulating huge trees efficiently, where parallel computing may potentially be a solution. Though hardware environments for parallel computing are getting widely available (e.g. PC clusters), parallel programming is still considered to be a hard task, especially for trees because of their ill-balanced and irregular structures.

We often develop sequential algorithms manipulating trees as recursive functions on trees. For example, the following recursive is a function for counting the number of the nodes:

$$size\ (RNode\ a\ ts) = 1 + \sum_{+}[size\ t_i \mid i \in [1..\#ts]]$$

which applies function *size* recursively to each of the children (t_i is the i-th value of ts), computes the summation of the results of children, and adds one to obtain the result. This recursive functions is easy to develop, and can be mapped to a parallel program of well-known divide-and-conquer style. Though being easy to develop, the divide-and-conquer approach may fail to be inefficient in parallel when the input tree is ill-balanced; for instance when the input tree is a monadic one (each child has only one child) divide-and-conquer parallel program is as inefficient as the sequential program.

For efficient parallel programs in regardless to the shapes of the input binary trees, there are important parallel algorithms called tree contraction algorithms. Miller and Reif [48]

first proposed the idea of the algorithms and many researchers [1, 3, 19, 28, 46, 49, 50] have studied them.

Unfortunately, many recursive algorithms cannot be straightforwardly mapped to parallel tree contraction programs. An example is to compute the pre-order numbering of trees. A recursive program using an accumulative parameter c may be defined as follows.

$$\begin{aligned} pre\ t &= pre'\ 0\ t \\ pre'\ c\ (RNode\ a\ ts) &= RNode\ c\ [pre'\ c_i\ t_i \mid i \in [1..\#ts]] \\ &\quad \textbf{where}\ c_i = c + 1 + l'_i \\ &\qquad\quad l'_i = \textstyle\sum_+ [size\ t_j \mid j \in [1..i-1]] \end{aligned}$$

This program can neither simply be mapped to a divide-and-conquer parallel program nor to a tree contraction algorithms, due to the dependency among children represented by the accumulative parameter.

To resolve these problems, we adopt a novel paradigm of parallel programming called *skeletal parallelism*[1], which was first proposed by Cole [16] and well discussed in [53], to the parallel programming on general trees. In skeletal parallelism, users build parallel programs by composing ready-made components called *skeletons*, which provide parallelizable computational patterns in a concise way and conceal the complicated parallel implementations from users. Skeletal parallelism has several advantages: the two most important ones are that users can build parallel programs as if they wrote sequential programs, and that the skeletal parallel programs are not only efficient but also architecture independent since the detailed implementation is hidden in parallel skeletons. There have been many studies on skeletal parallel programming for lists or arrays [10, 11, 13, 18, 27, 33, 56] and for binary trees [22, 25, 26, 44, 59], but for general trees of arbitrary shapes few have been studied in the area of skeletal parallel programming.

This paper addresses parallel programming on trees of arbitrary shapes, called *rose trees*. We start by formalizing seven basic computational patterns (skeletons) over rose trees based on the theory of Constructive Algorithmics [6, 9, 34, 47]. Constructive Algorithmics was originally proposed for systematic development of sequential algorithms, and has been applied to specification of parallel skeletons on data structures lists [57, 18, 27, 29], matrices [24], and binary trees [44, 58, 59]. Our rose-tree skeletons are straightforward extensions of binary-tree skeletons where two new skeletons are added to describe computational patterns among siblings. We then show that rose-tree skeletons can be implemented efficiently in parallel. We represent rose trees in the form of binary trees, and provide a mapping from rose-tree skeletons to computations on binary trees with parallel binary-tree skeletons. Since the binary-tree skeletons can be implemented efficiently in parallel [26, 59], our rose-tree skeletons can also be implemented in parallel, and thus we may call the skeletons as *parallel rose-tree skeletons*.

We can describe several algorithms using a single rose-tree skeleton, and more involved algorithms by composing them. To bridge the gap between recursive algorithms and skeletal parallel programs, we propose several theorems that specify general forms of recursive tree algorithms and show how they can be computed in parallel in terms of the parallel rose-tree skeletons. We give a strategy for deriving efficient parallel programs with these theorems, and demonstrate derivation of several parallel program for three non-trivial problems on general trees.

We have implemented the rose-tree skeletons in C++ and MPI as wrapper functions of the binary-tree skeletons based on the mapping from rose trees to binary trees. The derived skeletal parallel programs can be executed efficiently using our parallel skeleton library. We implemented so far a parallel skeletons library for several parallel data structures, including

[1] See `http://homepages.inf.ed.ac.uk/mic/Skeletons/`.

the binary-trees, and we adopt two distinguished mechanisms in C++, function objects and the template mechanism, to implement rose-tree skeletons. We show how these skeletons are implemented and report experimental results.

The rest of the paper is organized as follows. In Section 2, we briefly introduce the notations and review the parallel binary-tree skeletons. In Section 3, we formalize seven rose-tree skeletons with discussions about their expressiveness, and show they can be implemented in parallel with a mapping from rose trees to binary trees in Section 4. In Section 5, we specify recursive forms that can be implemented with rose-tree skeletons, and propose a strategy for systematic derivation of skeletal parallel programs from recursive functions. We apply our method to several examples in Section 6, and we explain how rose-tree skeletons are implemented in our skeleton library in Section 7. Finally we discuss related work in Section 8, and make conclusion remarks in Section 9.

2 Preliminaries

In this section, after introducing important notational conventions used in this paper, we review the binary-tree skeletons given in [44, 59].

2.1 Notations

In this paper, we use the notation of Haskell [5, 52]. In the following, we briefly review important notations and define lists, binary trees, and rose trees, and some functions on them.

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 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.

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 familiar arithmetic operators, we use operator $\uparrow$ to return the larger of two values; i.e., $a \uparrow b = \textbf{if}\ a \geq b\ \textbf{then}\ a\ \textbf{else}\ b$.

In deriving parallel programs, algebraic rules on operators such as associativity or distributivity play important roles. We introduce the following generalized rule of distributivity in terms of a closure property.

Definition 1 (Extended Distributivity [42]). Let $\otimes$ be an associative operator. The operator $\otimes$ is said to be *extended-distributive* over operator $\oplus$, if for any a, b, c, a', b', and c', there exist functions p_1, p_2, and p_3 such that 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 functions p_1, p_2, and p_3 as characteristic functions. □

Many pair of operators satisfy the extended distributivity. For example, let $\oplus$ be an associative operator, then $\oplus$ is also extended-distributive over $\oplus$ itself. If operator $\otimes$ distributes over operator $\oplus$, then of course the operator $\otimes$ is extended-distributive over $\oplus$. There are many pairs of operators where the distributivity does not hold but the extended distributivity does.

When the operator $\otimes$ is also commutative, we can simplify the definition as follows.

Lemma 1. Let $\otimes$ be an associative and commutative operator with unit. If 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 give a definition of the characteristic functions p'_1, p'_2, p'_3 of the extended-distributivity as follows.

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

□

Lists and List Comprehension Cons lists (or simply lists) are finite sequences of values of the same type. A list is constructed either by an empty list (*Nil*) or by adding a value to a list (*Cons*). The datatype of a list whose values are of type α is defined as follows.

$$\textbf{data}\ \mathit{List}\ \alpha = \mathit{Nil} \mid \mathit{Cons}\ \alpha\ (\mathit{List}\ \alpha)$$

We may use the following abbreviations: $[\alpha]$ for $\mathit{List}\ \alpha$, $[\,]$ for *Nil*, and $(a : as)$ for $(\mathit{Cons}\ a\ as)$.

We introduce two functions manipulating lists. Function *head* returns the first value of the input list, and function *tail* removes the first value from the list.

$$\begin{aligned}\mathit{head}\ (a : as) &= a\\ \mathit{tail}\ \ (a : as) &= as\end{aligned}$$

List comprehension is a syntax sugar for generating lists. The following is an example of list comprehension.

$$[f\ t_i \mid i \in [1..\#ts]]$$

List comprehension $[1..\#ts]$ generates a list of increasing integers starting from 1 and ending at the number of values in ts. In this paper, we denote t_i for the i-th value of a list ts, and we use similar notation for other lists too. Thus, the example above generates a list by applying function f to each value in ts.

We introduce another notation for consumption of lists. Let $\oplus$ be an associative operator with the unit $\iota_\oplus$, then $\sum_\oplus$ denotes the reduction of a list with the operator $\oplus$.

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

Binary Trees Binary trees are trees whose internal nodes have exactly two children. In this paper we assume the nodes in a binary tree have values of the same type. The datatype of binary trees whose nodes have values of type α is defined as follows.

$$\textbf{data}\ \mathit{BTree}\ \alpha\ \beta = \mathit{Leaf}\ \alpha \mid \mathit{Node}\ \beta\ (\mathit{BTree}\ \alpha\ \beta)\ (\mathit{BTree}\ \alpha\ \beta)$$

We introduce two functions manipulating binary trees. Function $root_b$ returns the value of the root node.

$$\begin{aligned} root_b\ (Leaf\ n) &= n \\ root_b\ (Node\ n\ l\ r) &= n \end{aligned}$$

Function $setroot_b$ takes a binary tree and a value, and replace the value of root node with the input value.

$$\begin{aligned} setroot_b\ (Leaf\ n)\ a &= Leaf\ a \\ setroot_b\ (Node\ n\ l\ r)\ a &= Node\ a\ l\ r \end{aligned}$$

Rose Trees Rose trees are trees whose internal nodes have an arbitrary number of children. In this paper we assume 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.

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

Similar to the binary trees, we introduce two functions manipulating rose trees. Function $root_r$ returns the value of the root node.

$$root_r\ (RNode\ a\ ts) = a$$

Function $setroot_r$ takes a rose tree and a value, and replaces the value of the root node with the input value.

$$setroot_r\ (RNode\ a\ ts)\ b = RNode\ b\ ts$$

2.2 Basic Binary-Tree Skeletons

In the following, we review the parallel skeletons for binary trees [44, 59]. There are five basic parallel binary-tree skeletons, which are categorized as follows.

- Independent computations: map and zipwith
- Bottom-up computations: reduce and upwards accumulate
- Top-down computation: downwards accumulate

These parallel skeletons can be efficiently implemented [26] based on the tree contraction algorithms [1, 48] which are important parallel algorithms manipulating a binary tree of an arbitrary shape efficiently. In this paper, parallel skeletons for binary trees are denoted in sans-serif font with a suffix b.

In the discussion of parallel computation cost of the parallel skeletons, we use N to indicates the number of nodes in the tree, and P the number of processors.

Map The parallel skeletons *map* and *zipwith* are computational patterns in which each node is computed independently.

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.

$$\begin{aligned} &\mathsf{map}_b :: (\alpha \to \gamma) \to (\beta \to \delta) \to BTree\ \alpha\ \beta \to BTree\ \gamma\ \delta \\ &\mathsf{map}_b\ k_L\ k_N\ (Leaf\ n) = Leaf\ (k_L\ n) \\ &\mathsf{map}_b\ k_L\ k_N\ (Node\ n\ l\ r) = Node\ (k_N\ n)\ (\mathsf{map}_b\ k_L\ k_N\ l)\ (\mathsf{map}_b\ k_L\ k_N\ r) \end{aligned}$$

The parallel computation cost of $\mathsf{map}_b\ k_L\ k_N$ is $O(N/P)$ if the functions k_L and k_N are computed in constant time.

Zipwith The parallel skeleton $\mathsf{zipwith}_b$ 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 pair of leaves and k_N to each pair of internal nodes.

$$
\begin{aligned}
&\mathsf{zipwith}_b :: (\alpha \to \alpha' \to \gamma) \to (\beta \to \beta' \to \delta) \to \mathit{BTree}\ \alpha\ \beta \to \mathit{BTree}\ \alpha'\ \beta' \to \mathit{BTree}\ \gamma\ \delta \\
&\mathsf{zipwith}_b\ k_L\ k_N\ (\mathit{Leaf}\ n)\ \ \ \ (\mathit{Leaf}\ n') = \mathit{Leaf}\ (k_L\ n\ n') \\
&\mathsf{zipwith}_b\ k_L\ k_N\ (\mathit{Node}\ n\ l\ r)\ (\mathit{Node}\ n'\ l'\ r') = \mathit{Node}\ (k_N\ n\ n')\ (\mathsf{zipwith}_b\ k_L\ k_N\ l\ l') \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad (\mathsf{zipwith}_b\ k_L\ k_N\ r\ r')
\end{aligned}
$$

The parallel computation cost of $\mathsf{zipwith}_b\ k_L\ k_N$ is $O(N/P)$, if the functions k_L and k_N are computed in constant time.

Reduce The parallel skeleton *reduce* and *upwards accumulate* are bottom-up computational patterns; the former returns a value and the latter returns a tree.

The parallel skeleton reduce_b takes a function k and a binary tree, and collapses the tree into a value by applying the function k in a bottom-up manner.

$$
\begin{aligned}
&\mathsf{reduce}_b :: (\beta \to \alpha \to \alpha \to \alpha) \to \mathit{BTree}\ \alpha\ \beta \to \alpha \\
&\mathsf{reduce}_b\ k\ (\mathit{Leaf}\ n)\ \ \ \ = n \\
&\mathsf{reduce}_b\ k\ (\mathit{Node}\ n\ l\ r) = k\ n\ (\mathsf{reduce}_b\ k\ l)\ (\mathsf{reduce}_b\ k\ r)
\end{aligned}
$$

To guarantee existence of an efficient parallel implementation, we require existence of functions ϕ, ψ_L, ψ_R, and G, satisfying the following equations.

$$
\begin{aligned}
k\ n\ x\ y\ \ \ \ \ \ \ \ \ \ &= G\ (\phi\ n)\ x\ y \\
G\ n\ l\ (G\ n'\ x\ y) &= G\ (\psi_L\ n\ l\ n')\ x\ y \\
G\ n\ (G\ n'\ x\ y)\ r &= G\ (\psi_R\ n\ r\ n')\ x\ y
\end{aligned}
$$

We denote the function k satisfying the condition above as $k = \langle \phi, \psi_L, \psi_R, G \rangle$.

When these functions exist we can implement the reduce_b skeleton based on the tree contraction algorithms and then the parallel computation cost of $\mathsf{reduce}_b\ \langle \phi, \psi_L, \psi_R, G \rangle$ is $O(N/P + \log P)$ if the functions ϕ, ψ_L, ψ_R, and G are computed in constant time.

Upwards Accumulate The parallel skeleton uAcc_b takes a function k and a binary tree, and computes $\mathsf{reduce}_b\ k$ for each subtree. In other words, this skeleton applies the function k in a bottom-up manner, and returns a tree whose values are the results of bottom-up reduction.

$$
\begin{aligned}
&\mathsf{uAcc}_b :: (\beta \to \alpha \to \alpha \to \alpha) \to \mathit{BTree}\ \alpha\ \beta \to \mathit{BTree}\ \alpha\ \alpha \\
&\mathsf{uAcc}_b\ k\ (\mathit{Leaf}\ n)\ \ \ \ = \mathit{Leaf}\ n \\
&\mathsf{uAcc}_b\ k\ (\mathit{Node}\ n\ l\ r) = \mathbf{let}\ l' = \mathsf{uAcc}_b\ k\ l \\
&\qquad\qquad\qquad\qquad\qquad\quad r' = \mathsf{uAcc}_b\ k\ r \\
&\qquad\qquad\qquad\qquad\quad\ \mathbf{in}\ \ \mathit{Node}\ (k\ n\ (\mathit{root}_b\ l')\ (\mathit{root}_b\ r'))\ l'\ r'
\end{aligned}
$$

To guarantee existence of an efficient parallel implementation, we require the same condition on the function k as the reduce_b skeleton. The parallel computation cost of $\mathsf{uAcc}_b\ \langle \phi, \psi_L, \psi_R, G \rangle$ is also $O(N/P + \log P)$ if all the functions are computed in constant time.

Downwards Accumulate The parallel skeleton *downwards accumulate* is a top-down computational pattern.

The parallel skeleton dAcc_b takes an associative operator $\oplus$, two functions g_l and g_r, and a binary tree. This skeleton computes in a top-down manner by updating an accumulative

parameter c, whose initial value is the unit of the operator, $\iota_\oplus$. The accumulative parameter is updated with $\oplus$ and g_l for the left child, and with $\oplus$ and g_r for the right child.

$$
\begin{aligned}
&\mathsf{dAcc}_b :: (\gamma \to \gamma \to \gamma) \to (\beta \to \gamma) \to (\beta \to \gamma) \to \mathit{BTree}\ \alpha\ \beta \to \mathit{BTree}\ \gamma \\
&\mathsf{dAcc}_b\ (\oplus)\ g_l\ g_r\ t \qquad\qquad\quad = \mathsf{dAcc}'_b\ (\oplus)\ g_l\ g_r\ \iota_\oplus\ t \\
&\mathsf{dAcc}'_b\ (\oplus)\ g_l\ g_r\ c\ (\mathit{Leaf}\ n) \quad = \mathit{Leaf}\ c \\
&\mathsf{dAcc}'_b\ (\oplus)\ g_l\ g_r\ c\ (\mathit{Node}\ n\ l\ r) = \mathit{Node}\ c\ (\mathsf{dAcc}'_b\ (\oplus)\ g_l\ g_r\ (c \oplus g_l\ n)\ l) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad (\mathsf{dAcc}'_b\ (\oplus)\ g_l\ g_r\ (c \oplus g_r\ n)\ r)
\end{aligned}
$$

The condition for efficient parallel computation is the associativity of the operator $\oplus$, and the parallel computation cost of $\mathsf{dAcc}_b\ (\oplus)\ g_l\ g_r$ is $O(N/P + \log P)$, if the operator $\oplus$ and the functions g_l and g_r are computed in constant time.

2.3 Specialized Binary-Tree Skeletons

We define two communication skeletons. Though we can implement these skeletons in terms of the basic parallel skeletons, they are important for reasons of readability and efficiency.

Get Left Child The parallel skeleton getchl_b takes a value and a binary tree, and puts the left child's value for each internal node and the input value for each leaf. In other words, this skeleton shifts each left child's value to its parent.

$$
\begin{aligned}
&\mathsf{getchl}_b :: \alpha \to \mathit{BTree}\ \beta\ \beta \to \mathit{BTree}\ \alpha\ \beta \\
&\mathsf{getchl}_b\ c\ (\mathit{Leaf}\ n) \quad = \mathit{Leaf}\ c \\
&\mathsf{getchl}_b\ c\ (\mathit{Node}\ n\ l\ r) = \mathit{Node}\ (\mathit{root}_b\ l)\ (\mathsf{getchl}_b\ c\ l)\ (\mathsf{getchl}_b\ c\ r)
\end{aligned}
$$

The parallel computation cost of getchl_b is $O(N/P)$, since the dependency is local.

Get Right Child The parallel skeleton getchr_b is a symmetry of the getchl_b skeleton. This skeleton takes a value and a binary tree, and put the right child's value for each internal node and the input value for each leaf.

$$
\begin{aligned}
&\mathsf{getchr}_b :: \alpha \to \mathit{BTree}\ \beta\ \beta \to \mathit{BTree}\ \alpha\ \beta \\
&\mathsf{getchr}_b\ c\ (\mathit{Leaf}\ n) \quad = \mathit{Leaf}\ c \\
&\mathsf{getchr}_b\ c\ (\mathit{Node}\ n\ l\ r) = \mathit{Node}\ (\mathit{root}_b\ r)\ (\mathsf{getchr}_b\ c\ l)\ (\mathsf{getchr}_b\ c\ r)
\end{aligned}
$$

The parallel computation cost of getchr_b is also $O(N/P)$.

3 Rose Tree Skeletons

In this section, we formalize computational patterns on rose trees based on the theory of Constructive Algorithmics [6, 9, 34, 47]. The key idea of Constructive Algorithmics is that the computation structure of algorithms should be derivable from the data structures the algorithms manipulate. Since the data structure of rose trees is an extension of binary trees with the list structure, the computational patterns on rose trees can be specified as extensions of computational patterns on binary trees and lists. We formalize seven skeletons on rose trees, which are categorized into the following four groups.

- Independent computations: map and zipwith
- Bottom-up computations: reduce and upwards accumulate
- Top-down computation: downwards accumulate
- Computation among siblings: rightwards accumulate and leftwards accumulate

The skeletons of the former three groups are extensions of the parallel binary-tree skeletons, and skeletons of the last group are extensions of list skeletons.

We shall denote the rose-tree skeletons in sans-serif font with a suffix r. In the following of this section, we give an intuitive specification of them using list comprehension. Their formal definition in terms of mutual recursive functions is summarized in Fig. 8.

Map The rose-tree skeleton map_r takes a function k and a rose tree, and applies the function to each node. Informally the map_r skeleton is defined as follows.

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

An example of the map_r skeleton is shown in Fig. 1.

Zipwith The rose-tree skeleton $\mathsf{zipwith}_r$ 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. Informally the $\mathsf{zipwith}_r$ skeleton is defined as follows.

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

An example of the $\mathsf{zipwith}_r$ skeleton is shown in Fig. 2.

Reduce The parallel skeleton reduce_r takes two operators $\oplus$ and $\otimes$ and a rose tree, and collapses the tree into a value in a bottom-up manner by folding the siblings with operator $\otimes$ and merging the result up to the parent with operator $\oplus$. Informally, the reduce_r skeleton is defined as follows, where the operator $\otimes$ should be associative by definition.

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

An example of the reduce_r skeleton is shown in Fig. 3.

Upwards Accumulate The rose-tree skeleton uAcc_r takes two operators $\oplus$ and $\otimes$ and a rose tree, and proceed the computation in the same way as the reduce_r skeleton. The difference is that the uAcc_r skeleton stores the intermediate results on the nodes and returns a tree of the same shape as the input tree. Informally, the uAcc_r skeleton is defined as follows using the reduce_r skeleton.

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

An example of the uAcc_r skeleton is shown in Fig. 4.

Downwards Accumulate The rose-tree skeleton dAcc_r takes an associative operator $\oplus$ and proceeds the computation in a top-down manner with an accumulative parameter c. The initial value of the accumulative parameter is the unit of the operator, $\iota_\oplus$, and it is updated by the operator $\oplus$ at each node. Informally, the dAcc_r skeleton is defined as follows.

$$\begin{aligned}&\mathsf{dAcc}_r :: (\alpha \to \alpha \to \alpha) \to \mathit{RTree}\ \alpha \to \mathit{RTree}\ \alpha\\&\mathsf{dAcc}_r\ (\oplus)\ t \qquad\qquad\quad = \mathsf{dAcc}'_r\ (\oplus)\ \iota_\oplus\ t\\&\mathsf{dAcc}'_r\ (\oplus)\ c\ (\mathit{RNode}\ a\ ts) = \mathit{RNode}\ c\ [\mathsf{dAcc}'_r\ (\oplus)\ (c \oplus a)\ t_i \mid i \in [1..\#ts]]\end{aligned}$$

An example of the dAcc_r skeleton is shown in Fig. 5.

Rightwards Accumulate The rose-tree skeleton rAcc_r takes an associative operator $\oplus$ and a rose tree, and applies the scan operation to each list of siblings from left to right. The scan operation on lists takes an associative operator and a list, and accumulates values with the operator as follows.

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

Informally, the rAcc_r skeleton is defined as follows with this *scan* function.

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

An example of the rAcc_r skeleton is shown in Fig. 6.

Leftwards Accumulate The rose-tree skeleton lAcc_r takes an associative operator $\oplus$ and a rose tree, and applies the scan operation to each list of siblings from right to left. The reversed scan operation $scan'$ is defined as follows.

$$scan'\ (\oplus)\ [a_1, a_2, \ldots, a_n] = [a_2 \oplus \cdots \oplus a_n, a_3 \oplus \cdots \oplus a_n, \ldots, \iota_\oplus]$$

Informally, the lAcc_r skeleton is defined as follows with this $scan'$ function.

$$\begin{aligned}
&\mathsf{lAcc}_r :: (\alpha \to \alpha \to \alpha) \to \mathit{RTree}\ \alpha \to \mathit{RTree}\ \alpha \\
&\mathsf{lAcc}_r\ (\oplus)\ (\mathit{RNode}\ a\ ts) = \mathbf{let}\ rs = scan'\ (\oplus)\ [root_r\ t_i \mid i \in [1..\#ts]] \\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{in}\ \ \mathit{RNode}\ \iota_\oplus\ [setroot_r\ (\mathsf{lAcc}_r\ (\oplus)\ t_i)\ r_i \mid i \in [1..\#ts]]
\end{aligned}$$

An example of the lAcc_r skeleton is shown in Fig. 7.

4 Parallelization of Rose-Tree Skeletons with Binary-Tree Skeletons

In this section, we show a parallel implementation for the rose-tree skeletons proposed in the previous section. The main idea is to represent rose trees by binary trees and to implement the rose-tree skeletons by the binary-tree skeletons. Since the binary-tree skeletons can be implemented efficiently in parallel [26, 59], the rose-tree skeletons can be efficiently implemented in parallel too.

4.1 Binary-Tree Representation of Rose Trees

There have been many studies for manipulating binary trees in parallel. In particular, several efficient implementations of the parallel tree contraction algorithms have been studied [1, 3, 21, 45, 48]. The parallel binary-tree skeletons can be implemented in parallel based on these tree contraction algorithms [25, 26].

To utilize these parallel implementations, we represent the rose trees by binary trees as shown in Fig. 9. This binary-tree representation is one of the widely used representations and is also discussed in [20]. In this 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 is its left-most child in the original rose tree, and the right child of a node in the binary tree is its next sibling in the rose tree. Let n be the number of nodes of the original rose tree, then the number of nodes of the binary tree turns out to be $2n + 1$, which guarantees the asymptotic cost when we utilize the parallel binary-tree skeletons on this binary-tree representation.

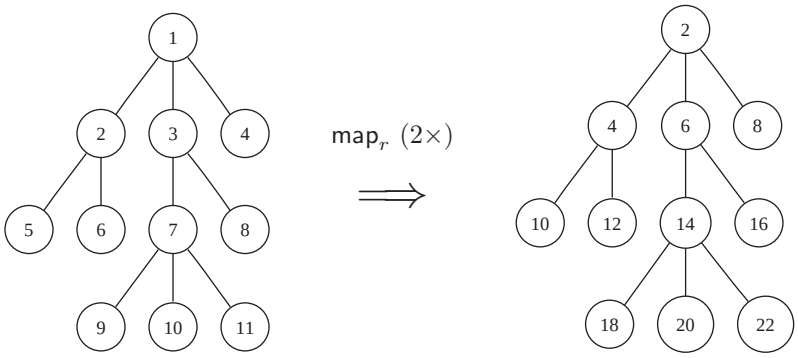

Fig. 1. An example of the map_r skeleton.

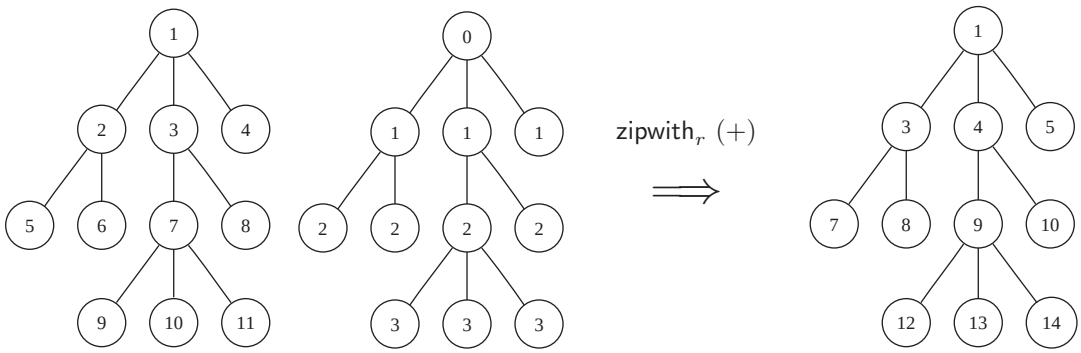

Fig. 2. An example of the $\mathsf{zipwith}_r$ skeleton.

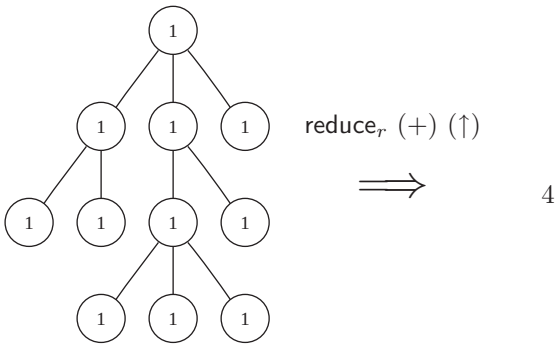

Fig. 3. An example of the reduce_r skeleton.

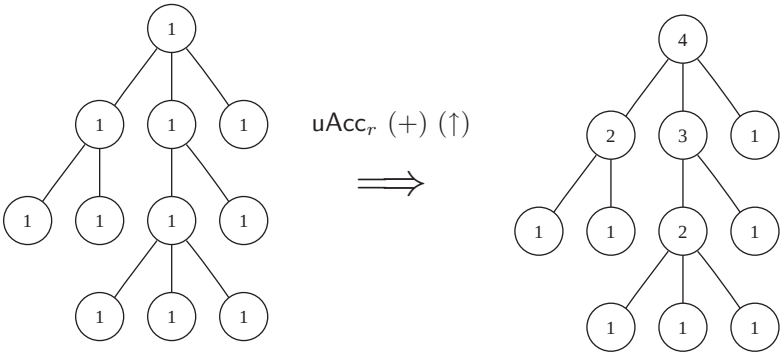

Fig. 4. An example of the uAcc_r skeleton.

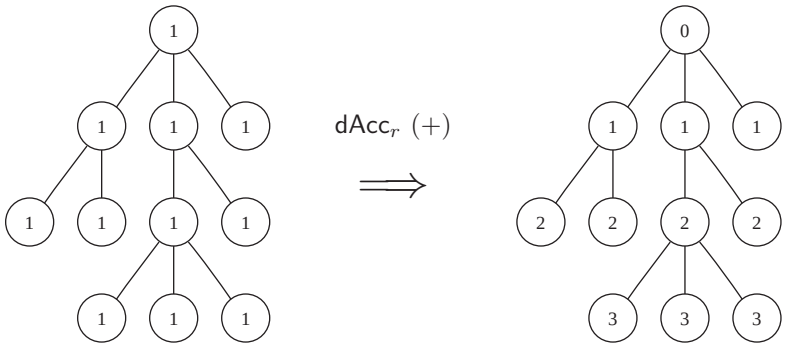

Fig. 5. An example of the dAcc_r skeleton.

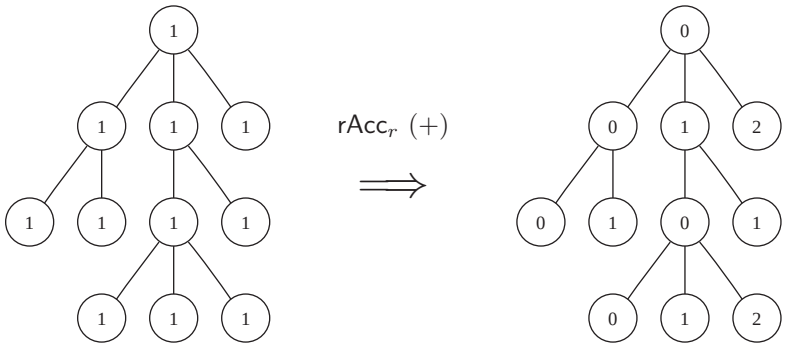

Fig. 6. An example of the rAcc_r skeleton.

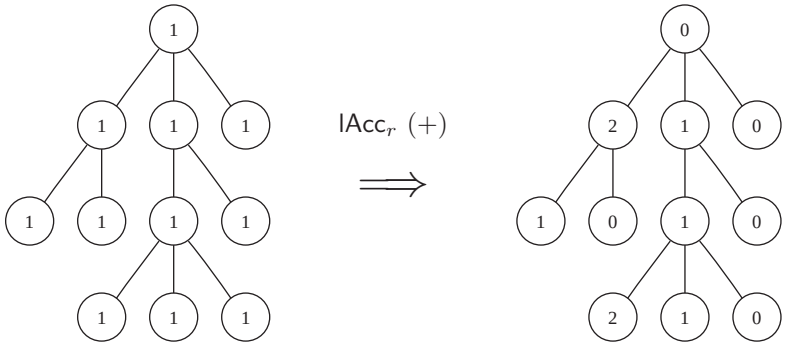

Fig. 7. An example of the lAcc_r skeleton.

$$
\begin{aligned}
&\mathsf{map}_r\ k\ (\mathit{RNode}\ a\ ts) = \mathit{RNode}\ (k\ a)\ (\mathsf{map}'_r\ k\ ts)\\
&\mathsf{map}'_r\ k\ [\,] \qquad\quad = [\,]\\
&\mathsf{map}'_r\ k\ (t : ts) \quad = \mathsf{map}_r\ k\ t : \mathsf{map}'_r\ k\ ts
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{zipwith}_r\ k\ (\mathit{RNode}\ a\ ts)\ (\mathit{RNode}\ a'\ ts') = \mathit{RNode}\ (k\ a\ a')\ (\mathsf{zipwith}'_r\ k\ ts\ ts')\\
&\mathsf{zipwith}'_r\ k\ [\,] \quad [\,] \qquad\qquad\qquad\quad = [\,]\\
&\mathsf{zipwith}'_r\ k\ (t : ts)\ (t' : ts') \qquad\quad = \mathsf{zipwith}_r\ k\ t\ t' : \mathsf{zipwith}'_r\ k\ ts\ ts'
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{reduce}_r\ (\oplus)\ (\otimes)\ (\mathit{RNode}\ a\ ts) = a \oplus \mathsf{reduce}'_r\ (\oplus)\ (\otimes)\ ts\\
&\mathsf{reduce}'_r\ (\oplus)\ (\otimes)\ [\,] \qquad\quad = \iota_\otimes\\
&\mathsf{reduce}'_r\ (\oplus)\ (\otimes)\ (t : ts) \quad = (\mathsf{reduce}_r\ (\oplus)\ (\otimes)\ t) \otimes (\mathsf{reduce}'_r\ (\oplus)\ (\otimes)\ ts)
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{uAcc}_r\ (\oplus)\ (\otimes)\ (\mathit{RNode}\ a\ ts) = \mathbf{let}\ ts' = \mathsf{uAcc}'_r\ (\oplus)\ (\otimes)\ ts\\
&\qquad\qquad\qquad\qquad\qquad\quad \mathbf{in}\ \ \mathit{RNode}\ (a \oplus \mathsf{uAcc}''_r\ (\otimes)\ ts')\ ts'\\
&\mathsf{uAcc}'_r\ (\oplus)\ (\otimes)\ [\,] \qquad\quad = [\,]\\
&\mathsf{uAcc}'_r\ (\oplus)\ (\otimes)\ (t : ts) \quad = \mathsf{uAcc}_r\ (\oplus)\ (\otimes)\ t : \mathsf{uAcc}'_r\ (\oplus)\ (\otimes)\ ts\\
&\mathsf{uAcc}''_r\ (\otimes)\ [\,] \qquad\quad = \iota_\otimes\\
&\mathsf{uAcc}''_r\ (\otimes)\ (t : ts) \quad = \mathit{root}_r\ t \otimes \mathsf{uAcc}''_r\ (\otimes)\ ts
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{dAcc}_r\ (\oplus)\ t \qquad\quad = \mathsf{dAcc}'_r\ (\oplus)\ \iota_\oplus\ t\\
&\mathsf{dAcc}'_r\ (\oplus)\ c\ (\mathit{RNode}\ a\ ts) = \mathit{RNode}\ c\ (\mathsf{dAcc}''_r\ (\oplus)\ (c \oplus a)\ ts)\\
&\mathsf{dAcc}''_r\ (\oplus)\ c\ [\,] \qquad\quad = [\,]\\
&\mathsf{dAcc}''_r\ (\oplus)\ c\ (t : ts) \quad = \mathsf{dAcc}'_r\ (\oplus)\ c\ t : \mathsf{dAcc}''_r\ (\oplus)\ c\ ts
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{rAcc}_r\ (\oplus)\ t \qquad\quad = \mathit{snd}\ (\mathsf{rAcc}'_r\ (\oplus)\ \iota_\oplus\ t)\\
&\mathsf{rAcc}'_r\ (\oplus)\ c\ (\mathit{RNode}\ a\ ts) = (c \oplus a, \mathit{RNode}\ c\ (\mathsf{rAcc}''_r\ (\oplus)\ \iota_\oplus\ ts))\\
&\mathsf{rAcc}''_r\ (\oplus)\ c\ [\,] \qquad\quad = [\,]\\
&\mathsf{rAcc}''_r\ (\oplus)\ c\ (t : ts) \quad = \mathbf{let}\ (c', t') = \mathsf{rAcc}'_r\ (\oplus)\ c\ t\\
&\qquad\qquad\qquad\qquad\qquad\quad \mathbf{in}\ \ t' : \mathsf{rAcc}''_r\ (\oplus)\ c'\ ts
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{lAcc}_r\ (\oplus)\ t \qquad\quad = \mathit{snd}\ (\mathsf{lAcc}'_r\ (\oplus)\ \iota_\oplus\ t)\\
&\mathsf{lAcc}'_r\ (\oplus)\ c\ (\mathit{RNode}\ a\ ts) = (a \oplus c, \mathit{RNode}\ c\ (\mathit{snd}\ (\mathsf{lAcc}''_r\ (\oplus)\ ts)))\\
&\mathsf{lAcc}''_r\ (\oplus)\ [\,] \qquad\quad = (\iota_\oplus, [\,])\\
&\mathsf{lAcc}''_r\ (\oplus)\ (t : ts) \quad = \mathbf{let}\ (c, ts') = \mathsf{lAcc}''_r\ (\oplus)\ ts\\
&\qquad\qquad\qquad\qquad\qquad\quad\ \ (c', t') = \mathsf{lAcc}'_r\ (\oplus)\ c\ t\\
&\qquad\qquad\qquad\qquad\qquad\quad \mathbf{in}\ \ (c', t' : ts')
\end{aligned}
$$

Fig. 8. The formal definition of seven rose-tree skeletons.

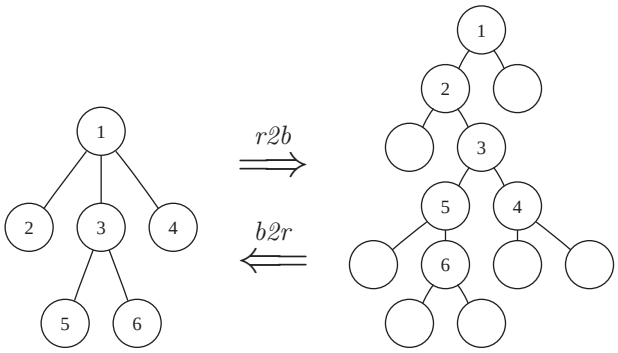

Fig. 9. The binary-tree representation of rose trees.

To formalize the binary-tree representation, we define function *r2b* to transform a rose tree into its binary-tree representation.

$$
\begin{array}{ll}
r2b :: RTree\ \alpha \to BTree\ \_\ \alpha & \\
r2b\ t & = r2b'\ t\ [\,] \\
r2b'\ (RNode\ a\ ts)\ ss & = Node\ a\ (r2b''\ ts)\ (r2b''\ ss) \\
r2b''\ [\,] & = Leaf\ \_ \\
r2b''\ (t : ts) & = r2b'\ t\ ts
\end{array}
$$

The inverse function *b2r*, which restores a rose tree from its binary-tree representation, is defined as follows.

$$
\begin{array}{ll}
b2r :: BTree\ \_\ \alpha \to RTree\ \alpha & \\
b2r\ t & = head\ (b2r'\ t) \\
b2r'\ (Node\ n\ l\ r) & = (RNode\ n\ (b2r'\ l)) : b2r'\ r \\
b2r'\ (Leaf\ n) & = [\,]
\end{array}
$$

4.2 Parallelization of Rose-Tree Skeletons with Binary-Tree Skeletons

We implement the rose-tree skeletons in terms of the parallel binary-tree skeletons based on the binary-tree representation. Generally speaking, the implementation of a rose-tree skeleton is decomposed into three steps: (1) applying the function *r2b* to transform a rose tree to a binary tree; (2) applying binary-tree skeletons to perform the computation of the skeleton; and (3) applying the function *b2r* to restore the rose-tree structure if necessary.

Map 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 can implement the map_r skeleton by simply using the map_b skeleton to apply the function to each internal node.

$$\mathsf{map}_r\ k = b2r \circ (\mathsf{map}_b\ \_\ k) \circ r2b$$

Zipwith Similar to the map_r skeleton, since there are no dependencies in the computation of the $\mathsf{zipwith}_r$ skeleton, we can implement the skeleton with the $\mathsf{zipwith}_b$ skeleton on the binary-tree representation.

$$\mathsf{zipwith}_r\ k\ t\ t' = b2r\ (\mathsf{zipwith}_b\ \_\ k\ (r2b\ t)\ (r2b\ t'))$$

Reduce Since the reduce_r skeleton returns a value rather than a rose tree, the computation of the reduce_r skeleton can be formalized as follows:

$$\mathsf{reduce}_r\ (\oplus)\ (\otimes) = f \circ r2b$$

where the function f is defined as follows.

$$\begin{aligned} f\ (\mathit{Leaf}\ n) &= \iota_\otimes \\ f\ (\mathit{Node}\ n\ l\ r) &= (n \oplus f\ l) \otimes f\ r \end{aligned}$$

This recursive function f can be decomposed into the map_b and reduce_b skeletons as follows.

$$\begin{aligned} f = (\mathsf{reduce}_b\ k) \circ (\mathsf{map}_b\ (\lambda x.\iota_\otimes)\ id) \\ \textbf{where}\ k\ n\ l\ r = (n \oplus l) \otimes r \end{aligned}$$

For the parallel implementation of the reduce_r skeleton, the function k above should satisfy the condition of the reduce_b skeleton. Since it is difficult to verify the condition in general, we assume some algebraic properties on the operators. In fact, we can derive functions for the reduce_b skeleton if the operators satisfy either of the following conditions.

- *The operators $\oplus$ and $\otimes$ construct an algebraic semi-ring*: the operators are defined on a type (domain), and the operator $\oplus$ is associative and distributive over the operator $\otimes$ in addition to the associativity and commutativity of the operator $\otimes$.
- *The operator $\otimes$ is extended-distributive over $\oplus$*: this condition includes the cases where the two operators are the same associative operator, and the operator $\otimes$ is distributive over $\oplus$.

Following the derivation method in [44], we derive the functions ϕ, ψ_L, ψ_R, and G for the reduce_b skeleton. The idea is to introduce a parametrized function which is closed under nested calls. The parametrized function is actually a binary function, and thus we need to consider two cases for the nested calls (for the left and right recursive calls). Let $F[a]$ be a set of parametrized binary functions where a is the parameter. The parametrized function is said to be closed if the following equations hold for some parameters A_l, A_r computed from a, a', l, and r.

$$\begin{aligned} F\ [a]\ l\ (F\ [a']\ x\ y) &= F\ [A_l]\ x\ y \\ F\ [a]\ (F\ [a']\ x\ y)\ r &= F\ [A_r]\ x\ y \end{aligned}$$

When we can find a parametrized function satisfying the above equations, we can derive the four functions systematically from the definition of A_l and A_r. The calculations of the nested calls are rather straightforward but become too long. In the following we only show the parametrized functions to be introduced and the results of the four functions.

Firstly, let $\oplus$ and $\otimes$ construct an algebraic semi-ring. For this case, we choose the parametrized function with three parameters a, b, and c as

$$\lambda l\ r.(a \oplus l) \otimes (b \oplus r) \otimes c$$

which is closed under nested calls. From the parametrized function above we can derive the function ϕ, ψ_L, ψ_R, and G for the reduce_b skeleton, and thus under this condition we can obtain an equivalent definition of the reduce_r skeleton on the binary-tree representation as follows.

$$\begin{aligned} \mathsf{reduce}_r\ (\oplus)\ (\otimes) = (\mathsf{reduce}_b\ \langle \phi, \psi_L, \psi_R, G \rangle) \circ (\mathsf{map}_b\ (\lambda\ x.\iota)\ id) \circ r2b \\ \textbf{where}\ \phi\ n &= (n, \iota_\oplus, \iota_\otimes) \\ \psi_L\ (a,b,c)\ l'\ (a',b',c') &= (b \oplus a', b \oplus b', (a \oplus l') \otimes (b \oplus c') \otimes c) \\ \psi_R\ (a,b,c)\ r'\ (a',b',c') &= (a \oplus a', a \oplus b', (a \oplus c') \otimes (b \oplus r') \otimes c) \\ G\ (a,b,c)\ l\ r &= (a \oplus l) \otimes (b \oplus r) \otimes c \end{aligned}$$

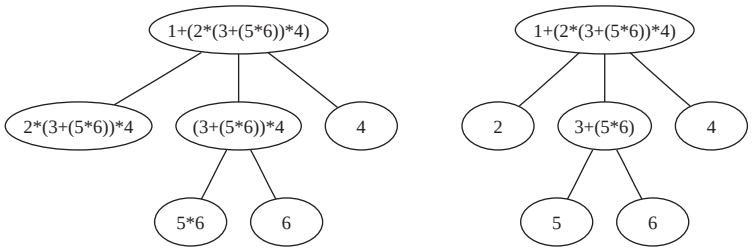

Fig. 10. The result of $b2r \circ (\mathsf{uAcc}_b\ k) \circ (\mathsf{map}_b\ (\lambda x.\iota_\otimes)\ id) \circ r2b$ (left) and the desired result for uAcc_r (right). Note that we denote $+$ for $\oplus$, and $*$ for $\otimes$.

Secondly, let the operator $\otimes$ be extended-distributive over $\oplus$ with the characteristic functions p_1, p_2, and p_3. For this case, we choose the parametrized function with four parameters a, b, c, and d as

$$\lambda l\ r.a \oplus (b \otimes (c \oplus l) \otimes r \otimes d)$$

which is closed under nested calls. From the parametrized function above we can derive the function ϕ, ψ_L, ψ_R, and G for the reduce_b skeleton as follows.

$$\begin{aligned}
&\phi\ n &&= (\iota_\oplus, \iota_\otimes, n, \iota_\otimes)\\
&\psi_L\ (a,b,c,d)\ l'\ (a',b',c',d') &&= (p_1\ tup, p_2\ tup, c', p_3\ tup)\\
&&&\quad \textbf{where}\ tup\ = (a, b \otimes (c \oplus l'), d, a', b', d')\\
&\psi_R\ (a,b,c,d)\ r'\ (a',b',c',d') &&= (p_1\ tup, p_2\ tup, c', p_3\ tup)\\
&&&\quad \textbf{where}\ tup\ = (a, b, r' \otimes d, p_1\ tup', p_2\ tup', p_3\ tup')\\
&&&\qquad\quad tup' = (c, \iota_\otimes, \iota_\otimes, a', b', d')\\
&G\ (a,b,c,d)\ l\ r &&= a \oplus (b \otimes (c \oplus l) \otimes r \otimes d)
\end{aligned}$$

Having proved that the function k satisfies the condition for the reduce_b skeleton, we obtain an equivalent definition of the reduce_r skeleton on the binary-tree representation as follows:

$$\mathsf{reduce}_r\ (\oplus)\ (\otimes) = (\mathsf{reduce}_b\ \langle \phi, \psi_L, \psi_R, G \rangle) \circ (\mathsf{map}_b\ (\lambda\ x.\iota_\otimes)\ id) \circ r2b$$

where the functions ϕ, ψ_L, ψ_R and G are defined above.

Upwards Accumulate Since the uAcc_r skeleton is very similar to the reduce_r skeleton, first let us consider applying the map_b and uAcc_b skeletons with the same functions used in parallelizing the reduce_r skeleton as follows.

$$\begin{aligned}
&b2r \circ (\mathsf{uAcc}_b\ k) \circ (\mathsf{map}_b\ (\lambda\ x.\iota)\ id) \circ r2b\\
&\textbf{where}\ k\ n\ l\ r = (n \oplus l) \otimes r
\end{aligned}$$

Unfortunately, as seen in the example in Fig. 10, the results are not what we want for the uAcc_r skeleton. This is because the result of a node includes its siblings' results.

Recall that the left child in the binary-tree representation is the left-most child in the original rose tree, and notice that the children's results are folded on the left child in the binary-tree representation. To obtain the desired result, we only need to compute again $(n \oplus l)$ for each internal node where n is the original node's value and l is the left child's value in the result of the uAcc_b skeleton. This can be realized by the getchl_b and $\mathsf{zipwith}_b$ skeletons.

Therefore, the equivalent definition of uAcc_r skeleton under the binary-tree representation is given as follows:

$$\begin{aligned}\mathsf{uAcc}_r\ (\oplus)\ (\otimes)\ t = \mathbf{let}\ & bt\ = r2b\ t \\ & bt' = \mathsf{uAcc}_b\ \langle \phi, \psi_L, \psi_R, G\rangle\ (\mathsf{map}_b\ (\lambda x.\iota_\otimes)\ id\ bt) \\ \mathbf{in}\ & b2r\ (\mathsf{zipwith}_b\ \_\ (\lambda n\ l.n \oplus l)\ bt\ (\mathsf{getchl}_b\ \_\ bt'))\end{aligned}$$

where the functions ϕ, ψ_L, ψ_R and G are specified in the same way as those of the reduce_r skeleton under the same conditions.

Downwards Accumulate The dAcc_r skeleton is a top-down computational pattern. By introducing a recursive function with an accumulative parameter, we can specify the dAcc_r skeleton on the binary-tree representation.

$$\begin{aligned}\mathsf{dAcc}_r\ (\oplus) = b2r \circ (f\ \iota_\oplus) \circ r2b \\ \mathbf{where}\ f\ c\ (Leaf\ \_) \quad &= \_ \\ f\ c\ (Node\ n\ l\ r) &= Node\ c\ (f\ (c \oplus n)\ l)\ (f\ c\ r)\end{aligned}$$

Noting that $c = c \oplus \iota_\oplus$, we can write the function f in terms of the uAcc_b skeleton. Therefore, an equivalent definition of the dAcc_r skeleton in terms of the binary-tree skeletons is as follows.

$$\mathsf{dAcc}_r\ (\oplus) = b2r \circ (\mathsf{dAcc}_b\ (\oplus)\ id\ (\lambda x.\iota_\oplus)) \circ r2b$$

Rightwards Accumulate The skeleton rAcc_r traverses the siblings from left to right on rose trees, which corresponds to a top-down traversal on binary trees. With a recursive function with an accumulative parameter, the rAcc_r skeleton is defined as follows.

$$\begin{aligned}\mathsf{rAcc}_r\ (\oplus) = b2r \circ (f\ \iota_\oplus) \circ r2b \\ \mathbf{where}\ f\ c\ (Leaf\ \_) \quad &= Leaf\ \_ \\ f\ c\ (Node\ n\ l\ r) &= Node\ c\ (f\ \iota_\oplus\ l)\ (f\ (c \oplus n)\ r)\end{aligned}$$

The function f above is a top-down computation on binary-tree representation, but in fact it cannot be simply described with the dAcc_b skeleton with its operator being $\oplus$, since we cannot update the accumulative parameter for left child as $\iota_\oplus = c \oplus g\ a$ with a suitable function g. Therefore, it is required to derive an associative operator to make a use of dAcc_b.

To derive an associative operator, we utilize the context preservation technique [13], which derives an associative operator from a parametrized function which is closed under function composition. In this case, we choose a parametrized function defined with three parameter p, a, and b, as follows:

$$\lambda x.\mathbf{if}\ p\ \mathbf{then}\ x \oplus a\ \mathbf{else}\ b$$

which is closed under function composition. Based on this parametrized function, we can derive an associative operator $\otimes$ defined as follows:

$$(p, a, b) \otimes (p', a', b') = (p' \wedge p, a \oplus a', \mathbf{if}\ p'\ \mathbf{then}\ b \oplus a'\ \mathbf{else}\ b')$$

An right unit of this operator is $\iota_\otimes = (True, \iota_\oplus, \_)$, but unfortunately there is no left unit. Thus we introduce another flag as follows:

$$\begin{aligned}(True, p, a, b) \otimes' (False, \_, \_, \_) &= (True, p, a, b) \\ (False, \_, \_, \_) \otimes' (flag, p, a, b) &= (flag, p, a, b) \\ (True, p, a, b) \otimes' (True, p', a', b') &= (True, p' \wedge p, a \oplus a', \mathbf{if}\ p'\ \mathbf{then}\ b \oplus a'\ \mathbf{else}\ b')\end{aligned}$$

where the unit is given as $(False, True, \iota_\oplus, \_)$. Using this operator, we succeed in deriving an equivalent definition of the rAcc_r skeleton in terms of the binary-tree skeletons.

$$\begin{aligned}\mathsf{rAcc}_r\ (\oplus) = & b2r \circ (\mathsf{map}_b\ \_\ k) \circ (\mathsf{dAcc}_r\ (\otimes)\ g_l\ g_r) \circ r2b \\ \mathbf{where}\ & g_l\ x \quad &= (True, False, \_, \iota_\oplus) \\ & g_r\ x \quad &= (True, True, x, \_) \\ & k\ (flag, p, a, b) &= \mathbf{if}\ p\ \mathbf{then}\ a\ \mathbf{else}\ b\end{aligned}$$

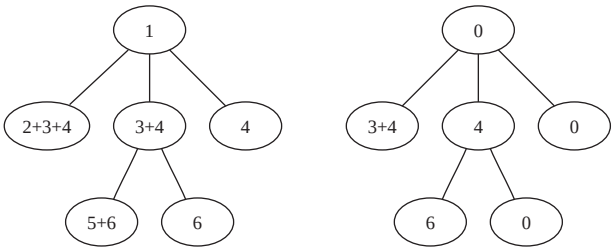

Fig. 11. The result of $b2r \circ (\mathsf{uAcc}_b\ k) \circ (\mathsf{map}_b\ (\lambda x.\iota_\oplus)\ id) \circ r2b$ (left) and the desired result of lAcc_r (right). Note that we denote $+$ for $\oplus$ and 0 for $\iota_\oplus$.

Leftwards Accumulate The skeleton lAcc_r traverses the siblings from right to left, which corresponds to a bottom-up traversal on the binary-tree representation. Therefore, an equivalent definition of the lAcc_r skeleton on the binary-tree representation may have a call of the uAcc_b skeleton. We first consider the following composition of the skeletons uAcc_b and map_b.

$$\begin{aligned} &b2r \circ (\mathsf{uAcc}_b\ k) \circ (\mathsf{map}_b\ (\lambda x.\iota_\oplus)\ id) \circ r2b \\ &\textbf{where}\ k\ n\ l\ r = n \oplus r \end{aligned}$$

The results of this computation are slightly different from what we want for the lAcc_r skeleton in the sense that the results should be shifted to the left by one on the rose tree as shown in Fig. 11. We resolve this problem by applying the getchr_b skeleton before restoring the rose-tree structure with the $b2r$ function.

Having derived an equivalent definition of the lAcc_r skeleton in terms of the binary-tree skeletons, we turn to prove that the function k satisfies the condition of the uAcc_b skeleton. For the function k above, we introduce a parametrized function with three parameters p, a, and b as

$$\lambda l\ r.\textbf{if}\ p\ \textbf{then}\ a \oplus r\ \textbf{else}\ b$$

which is closed under the nested calls.

Therefore, we obtain an equivalent implementation of the lAcc_r skeleton in terms of the binary-tree skeletons as follows.

$$\begin{aligned} \mathsf{lAcc}_r\ (\oplus) = b2r \circ (\mathsf{getchr}_b\ \_) \circ (\mathsf{uAcc}_b\ \langle \phi, \psi_L, \psi_R, G \rangle) \circ (\mathsf{map}_b\ (\lambda x.\iota_\oplus)\ id) \circ r2b \\ \textbf{where}\ \phi\ n &= (\mathit{True}, n, \_) \\ \psi_L\ (p,a,b)\ l'\ (p',a',b') &= (p \wedge p', a \oplus a', \textbf{if}\ p\ \textbf{then}\ a \oplus b'\ \textbf{else}\ b) \\ \psi_R\ (p,a,b)\ r'\ (p',a',b') &= (\mathit{False}, \_, \textbf{if}\ p\ \textbf{then}\ a \oplus r'\ \textbf{else}\ b) \\ G\ (p,a,b)\ l\ r &= \textbf{if}\ p\ \textbf{then}\ a \oplus r\ \textbf{else}\ b \end{aligned}$$

Now we summarize this section with the following theorem.

Theorem 1. The seven parallel skeletons for rose trees defined in Section 3 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 are almost self-evident from the derivations so far. We can prove this theorem by induction on the structure of rose trees using the definitions in terms of the mutual recursive functions, too. □

5 Diffusion Theorems

Despite the simplicity of the parallel skeletons for rose trees defined in Sections 3 and 4, we can write many algorithms by composing some of them. In this section, we show the

expressiveness of the parallel rose-tree skeletons by providing skeletal programs for several general forms of recursive algorithms. First, we discuss the generalization of the $\mathsf{zipwith}_r$ skeleton, bottom-up computations, and top-down computations. Then, we give more powerful theorems, *diffusion theorems*, for parallelizing recursive algorithms on rose trees. Finally, we highlight a strategy for deriving efficient parallel algorithms from the recursive functions on rose trees based on the diffusion theorems.

5.1 Generalization of $\mathsf{zipwith}_r$ skeleton

The $\mathsf{zipwith}_r$ skeleton takes two rose trees of the same shape and merges them into a single one. We can generalize this skeleton to take more than two rose trees.

Let us consider a new skeleton $\mathsf{zipwith3}_r$ which takes a ternary function and three rose trees of the same shape, and returns a tree by merging the corresponding three nodes. Informally, this skeleton is defined just like the $\mathsf{zipwith}_r$ skeleton as follows.

$$\begin{aligned}&\mathsf{zipwith3}_r\ k\ (\mathit{RNode}\ a\ ts)\ (\mathit{RNode}\ b\ ss)\ (\mathit{RNode}\ c\ rs)\\ &\quad = \mathit{RNode}\ (k\ a\ b\ c)\ [\mathsf{zipwith3}_r\ k\ t_i\ s_i\ r_i \mid i \in [1..\#ts]]\end{aligned}$$

We can implement this $\mathsf{zipwith3}_r$ skeleton by using the $\mathsf{zipwith}_r$ skeleton twice as follows.

$$\begin{aligned}\mathsf{zipwith3}_r\ k\ t\ s\ r = \mathsf{zipwith}_r\ k'\ (\mathsf{zipwith}_r\ (,)\ t\ s)\ r\\ \textbf{where}\ k'\ (a,b)\ c = k\ a\ b\ c\end{aligned}$$

Here, the sectioned operator $(,)$ takes two values and returns a pair of them defined as $(,)\ a\ b = (a, b)$.

We can further generalize the $\mathsf{zipwith}_r$ skeleton to take more trees in the same way.

5.2 Parallelizing General Bottom-up Computations

Though the parallel skeleton reduce_r represents a bottom-up computational pattern, there are many bottom-up algorithms that cannot be written only with it. In the following we formalize a more general bottom-up computational pattern called *homomorphism* and show its implementation with the rose-tree skeletons.

The homomorphism on rose trees is a bottom-up computation pattern defined recursively as follows.

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

This computational pattern can express 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 homomorphisms that can be implemented in parallel with the parallel skeletons.

Definition 2 (Parallelizable Homomorphism). Let $\otimes$ be an associative operator, $\oplus$ be an operator, and k be a function. A function h is said to be a parallelizable homomorphism if it is defined as

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

and the operators $\oplus$ and $\otimes$ satisfy either of the following conditions.

- The operators $\oplus$ and $\otimes$ construct an algebraic semi-ring, i.e. the operator $\oplus$ is associative and distributive over the operator $\otimes$, and the operator $\otimes$ is both associative and commutative.
- The operator $\otimes$ is extended-distributive over the operator $\oplus$.

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

This parallelizable homomorphism is a generalization of reduce_r at the point of applying function k to all the nodes. Therefore, we can compute the parallelizable homomorphism by applying the map_r skeleton followed by the reduce_r skeleton.

Theorem 2. A parallelizable homomorphism $h = (\![k, \oplus, \otimes]\!)$ can be implemented in parallel with the rose-tree skeletons as follows.

$$h = (\mathsf{reduce}_r\ (\oplus)\ (\otimes)) \circ (\mathsf{map}_r\ k)$$

Proof: We can prove this theorem by the induction on the structure of rose trees. The proof is given by showing equations for the top case ($t = RNode\ a\ ts$), base case ($ts = [\,]$), and inductive case ($ts = (RNode\ a\ ts') : ss$). The detailed proof is given in Appendix A. □

It is worth noting that the theorem is an extension of the first homomorphism lemma on list [6]; for example, based on this idea we may introduce the idea of the so-called almost homomorphism [17].

The following are two examples of the parallelizable homomorphisms. Function *size* counts up the number of nodes in a rose tree, and function *height* computes the height of a rose tree.

$$\begin{aligned} size &= (\![\lambda x.1, +, +]\!) \\ &= (\mathsf{reduce}_r\ (+)\ (+)) \circ (\mathsf{map}_r\ (\lambda x.1)) \\ height &= (\![\lambda x.1, +, \uparrow]\!) \\ &= (\mathsf{reduce}_r\ (+)\ (\uparrow)) \circ (\mathsf{map}_r\ (\lambda x.1)) \end{aligned}$$

5.3 Parallelizing General Top-down Computations

The parallel skeleton dAcc_r is one of the simplest top-down computational patterns. We may consider a generalization in the sense of the following three points.

1. The initial value of the accumulative parameter is not the unit of the operator but a certain value c.
2. The accumulative parameter is updated not only with the associative operator $\oplus$ but also with a function g as $c' = c \oplus g\ a$.
3. The result value on a node is computed with the original value a and the accumulative parameter c using a function k as $(k\ a\ c)$, not the accumulative parameter itself.

Such generalization can be captured by the following recursive function f defined with an associative operator $\oplus$ and two functions g and k.

$$f\ c\ (RNode\ a\ ts) = RNode\ (k\ a\ c)\ [f\ (c \oplus g\ a)\ t_i \mid i \in [1..\#ts]]$$

We can implement this function f by calling rose-tree skeletons in the following way.

$$\begin{aligned} f\ c\ t = \mathbf{let}\ & gt = \mathsf{map}_r\ g\ t \\ & dt = \mathsf{dAcc}_r\ (\oplus)\ gt \\ & ct = \mathsf{map}_r\ (\lambda x.c \oplus x)\ dt \\ \mathbf{in}\ & \mathsf{zipwith}_r\ k\ t\ ct \end{aligned}$$

1. Since the operator $\oplus$ is associative, equation $c \oplus a = c \oplus (\iota_\oplus \oplus a)$ holds for any a and c. Therefore, we put the initial value of the accumulative parameter together using the map_r skeleton, after the computation of the dAcc_r skeleton (line 3).
2. We achieve this generalization by applying function g to every node with the map_r skeleton before applying the dAcc_r skeleton (line 1).
3. We achieve this generalization by zipping up the original tree and the result of the dAcc_r skeleton using the $\mathsf{zipwith}_r$ skeleton (line 4).

We may simplify the parallel implementation by fusing the successive calls of the map_r and the $\mathsf{zipwith}_r$ skeletons, and obtain the following theorem.

Theorem 3. Let g and k be functions, $\oplus$ be an associative operator. The recursive function f defined by

$$f\ c\ (RNode\ a\ ts) = RNode\ (k\ a\ c)\ [f\ (c \oplus g\ a)\ t_i \mid i \in [1..\#ts]]$$

can be implemented in parallel with the rose-tree skeletons as follows.

$$f\ c\ t = \mathsf{zipwith}_r\ (\lambda a\ d.k\ a\ (c \oplus d))\ t\ (\mathsf{dAcc}_r\ (\oplus)\ (\mathsf{map}_r\ g\ t))$$

Proof: We can prove this theorem by the induction on the structure of rose trees. Since it is rather straightforward, we omit the detain, which is given in Appendix A. □

We now consider another top-down computation where the accumulative parameter passed to each child is updated based on the child's value, where the values passed to the children may be different. We formalize such a generalized computational pattern as the following recursive function using two functions g and k and an associative operator $\oplus$.

$$f\ c\ (RNode\ a\ ts) = RNode\ (k\ a\ c)\ [f\ (c \oplus g\ (root_r\ t_i)\ a)\ t_i \mid i \in [1..\#ts]]$$

In this definition there is actually some redundancy caused by the fact that the result value is computed with the original value whereas the accumulative parameter is already updated with the value. In many cases one of them is used and so we can simplify the skeletal programs derived.

What makes the function f above complicated is the existence of the function $root_r$. We first remove the $root_r$ function by shifting the accumulative parameter c by one to the child, and putting it together with the original value. With this in mind, we obtain the following recursive definition.

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

It is still hard to write the newly defined function with the single dAcc_r skeleton, since we need to derive an associative operator for the pair of values. We thus shift the parent value to the children for each node and zip it up with the original value before computing with the dAcc_r skeleton. We may define such a shift operation using operator $\gg$ defined as $a \gg b = b$, but we need its right unit. To achieve this, we introduce a flag and define the shift operation $\mathsf{getparent}_r$ as follows where the unit of $(\gg')$ is $(False, \_)$.

$$\begin{aligned} \mathsf{getparent}_r = (\mathsf{map}_r\ snd) \circ (\mathsf{dAcc}_r\ (\gg')) \circ (\mathsf{map}_r\ (\lambda x.(True, x))) \\ \textbf{where}\ (\_,\_) \gg' (True, b) = (True, b) \\ (f, a) \gg' (False, \_) = (f, a) \end{aligned}$$

We can attach for each node the parent value by the $\mathsf{zipwith}_r$ skeleton after $\mathsf{getparent}_r$.

$$pt = \mathsf{zipwith}_r\ (,)\ (\mathsf{getparent}_r\ t)\ t$$

On this tree of pairs we can derive the definition as follows.

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

The function f'' has almost the same form as the first generalized top-down computation except that the different computation is performed on the root node. By assigning the

expected value on the root node (line 3 below), we can define the function f in terms of the rose-tree skeleton.

$$\begin{aligned} f\ c\ t = \textbf{let}\ & pt\ = \mathsf{zipwith}_r\ (,)\ (\mathsf{getparent}_r\ t)\ t \\ & gt\ = \mathsf{map}_r\ (\lambda(d,a).g\ d\ a)\ pt \\ & dt\ = setroot_r\ \iota_\oplus\ gt \\ & dt' = \mathsf{dAcc}_r\ (\oplus)\ dt \\ \textbf{in}\ & \mathsf{zipwith3}_r\ (\lambda a\ d\ d'.k\ a\ (c \oplus d \oplus d'))\ t\ dt'\ dt \end{aligned}$$

We may fuse the successive calls of the $\mathsf{zipwith}_r$ and the map_r skeletons into one $\mathsf{zipwith}_r$ skeleton to obtain the equivalent skeletal program for the generalized top-down computation, as summarized in the following theorem.

Theorem 4. Let g and k be functions, $\oplus$ be an associative operator. The function f defined as

$$f\ c\ (RNode\ a\ ts) = RNode\ (k\ a\ c)\ [f\ (c \oplus g\ a\ (root_r\ t_i))\ t_i \mid i \in [1..\#ts]]$$

can be implemented in parallel in terms of the rose-tree skeletons as follows.

$$\begin{aligned} f\ c\ t = \textbf{let}\ & gt = setroot_r\ \iota_\oplus\ (\mathsf{zipwith}_r\ g\ (\mathsf{getparent}_r\ t)\ t) \\ & dt = \mathsf{dAcc}_r\ (\oplus)\ gt \\ \textbf{in}\ & \mathsf{zipwith3}_r\ (\lambda a\ d\ d'.k\ a\ (c \oplus d \oplus d'))\ t\ dt\ gt \end{aligned}$$

Proof: We can prove this theorem by the induction on the structure of rose trees. We omit the proof here, and the detailed proof is given in Appendix A.

□

5.4 Diffusion Theorems

To further study the expressiveness of the rose-tree skeletons, now we consider recursive computational patterns where the value of a node depends on not only the values of the ancestors but also those of the descendants.

Firstly, we study a bottom-up computational pattern in which there is a top-down dependency. We formalize such a computational pattern by describing the top-down dependency with an accumulative parameter c updated by a function g and an associative operator $\odot$, and the bottom-up computation with a parallelizable homomorphism $(\![k, \oplus, \otimes]\!)$.

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

To compute this function we first generate a pair of the original value and the accumulative parameter for each node using a generalized top-down computation (Theorem 3), and then perform the over-all bottom-up computation which is exactly a parallelizable homomorphism (Theorem 2). According to the theorems we obtain the following skeletal program in terms of the rose-tree skeletons.

$$\begin{aligned} f\ c\ t = \textbf{let}\ & t' = \mathsf{zipwith}_r\ (\lambda a\ d.(a, c \odot d))\ t\ (\mathsf{dAcc}_r\ (\odot)\ (\mathsf{map}_r\ g\ t)) \\ \textbf{in}\ & \mathsf{reduce}_r\ (\oplus)\ (\otimes)\ (\mathsf{map}_r\ (\lambda(a,c).k\ a\ c)\ t') \end{aligned}$$

We may simplify the program above by restructuring the map_r and $\mathsf{zipwith}_r$ skeletons, and in summary we obtain the following theorem.

Theorem 5. Let $(\![k, \oplus, \otimes]\!)$ be a parallelizable homomorphism, $\odot$ be an associative operator, and g be a function, and function f be defined as follows.

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

The function f can be implemented in parallel with the rose-tree skeletons as follows.

$$\begin{aligned} f\ c\ t = \textbf{let}\ & dt = \mathsf{dAcc}_r\ (\odot)\ (\mathsf{map}_r\ g\ t) \\ \textbf{in}\ & \mathsf{reduce}_r\ (\oplus)\ (\otimes)\ (\mathsf{zipwith}_r\ k\ t\ (\mathsf{map}_r\ (c\odot)\ dt)) \end{aligned}$$

Proof: We can prove this theorem by induction on the structure of rose trees. We omit the detailed proof, which is shown in Appendix A. □

Secondly, we study a top-down computational pattern in which there is a bottom-up dependency on the nodes. Let the bottom-up dependency be represented by a parallelizable homomorphism $h = (\![k', \oplus', \otimes']\!)$, and the top-down computation be represented by an accumulative parameter c updated by an associative operator $\odot$ and functions g and k. We then formalize such a computational pattern as the following recursive function.

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

This function is an instance of paramorphism [47] defined on rose trees.

Naively implementing the function above needs multiple traversals over rose trees, but such multiple traversals can be removed by the *tupling* technique discussed in [30]. Following this idea, we firstly carry out the bottom-up computation using the uAcc_r skeleton instead of the reduce_r skeleton, and then perform the generalized top-down computation as in Theorem 3.

With these parallelization steps, we can derive a skeletal parallel program as the following theorem says.

Theorem 6. Let h be a parallelizable catamorphism $h = (\![k', \oplus', \otimes']\!)$, $\odot$ be an associative operator, and g and k be functions, and the function f be defined as follows.

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

The function f can be implemented in parallel with the rose-tree skeletons as follows.

$$\begin{aligned} f\ c\ t = \textbf{let}\ & t' = \mathsf{uAcc}_r\ (\oplus')\ (\otimes')\ (\mathsf{map}_r\ k'\ t) \\ & dt = \mathsf{dAcc}_r\ (\odot)\ (\mathsf{zipwith}_r\ g\ t\ t') \\ \textbf{in}\ & \mathsf{zipwith}_r\ (\lambda a\ d.k\ a\ (c \odot d))\ t\ dt \end{aligned}$$

Proof: To prove this theorem, we need to first prove the following equation:

$$root_r\ (\mathsf{uAcc}_r\ (\oplus)\ (\otimes)\ t) = \mathsf{reduce}_r\ (\oplus)\ (\otimes)\ t$$

by the induction on the structure of rose trees. With this equation, we can prove this theorem by the induction. The detailed proofs of the equation and this theorem are given in Appendix A. □

Finally, we study a more general and complicated computational pattern defined as a top-down computational pattern 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 a function k and an accumulative parameter c updated with an associative operator $\odot$, and function g; the bottom-up dependencies are specified with parallelizable homomorphisms $f'_l = (\![k'_l, \oplus'_l, \otimes'_l]\!)$, $f' = (\![k', \oplus', \otimes']\!)$, and $f'_r = (\![k'_r, \oplus'_r, \otimes'_r]\!)$; and the inter-siblings dependencies are specified for each child as summation of the values of its left siblings or its right siblings in terms of associative operators $\circledast_l$ and $\circledast_r$. The accumulative parameter is updated based on the value of the node (a),

the i-th child's subtree (t'_i), summation of left subtrees (l_i), and summation right subtrees (r_i). It is worth noting that the value of accumulative parameter passed to children may differ in this specification.

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

Since it is not efficient to compute l_i, t'_i, and r_i for each node independently, we should compute them for all node at a time, and put them together by the tupling technique [30]. We implement the bottom-up computations specified as parallelizable homomorphisms in a similar way as Theorem 2 except that we use the uAcc_r skeleton instead of the reduce_r skeleton. For inter-siblings dependencies, we use the rAcc_r and lAcc_r skeletons. Therefore, such a pre-process can be performed by the following four steps.

1. Compute l_i for all the children using the map_r, uAcc_r, and rAcc_r skeletons (line 1).
2. Compute t'_i for all the children using the map_r, and uAcc_r skeletons (line 2).
3. Compute r_i for all the children using the map_r, uAcc_r, and lAcc_r skeletons (line 3).
4. Zipping up the results by using the $\mathsf{zipwith3}_r$ skeleton (line 4).

$$\begin{aligned} t'' = \textbf{let}\ lt &= \mathsf{rAcc}_r\ (\circledast_l)\ (\mathsf{uAcc}_r\ (\oplus'_l)\ (\otimes'_l)\ (\mathsf{map}_r\ k'_l\ t)) \\ t' &= \mathsf{uAcc}_r\ (\oplus')\ (\otimes')\ (\mathsf{map}_r\ k'\ t) \\ rt &= \mathsf{lAcc}_r\ (\circledast_r)\ (\mathsf{uAcc}_r\ (\oplus'_r)\ (\otimes'_r)\ (\mathsf{map}_r\ k'_r\ t)) \\ \textbf{in}\ \ &\mathsf{zipwith3}_r\ lt\ t'\ rt \end{aligned}$$

After this processing we can compute over the original rose tree t ($= RNode\ a\ ts$) and the rose tree of tuples t'' ($= RNode\ a''\ ts''$) by the following function f'.

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

Since this function f' is a top-down computation where the accumulative parameter is updated using the child's value, we can derive a skeletal program by applying the technique of Theorem 4 as follows.

$$\begin{aligned} f'\ c\ t\ t'' = \textbf{let}\ gt &= setroot_r\ \iota_\odot\ (\mathsf{zipwith}_r\ g\ (\mathsf{getparent}_r\ t)\ t'') \\ dt &= \mathsf{dAcc}_r\ (\odot)\ gt \\ \textbf{in}\ \ &\mathsf{zipwith3}_r\ (\lambda a\ d\ d'.k\ a\ (c \odot d \odot d'))\ t\ dt\ gt \end{aligned}$$

In summary, we can compute the function f by the skeletal parallel programs in terms of the rose-tree skeletons given as seen in the following theorem.

Theorem 7. Let $\odot$, $\circledast_l$ and $\circledast_r$ be associative operators, f'_l, f, and f'_r be parallelizable homomorphisms defined as $f'_l = (\!|k'_l, \oplus'_l, \otimes'_l|\!)$, $f' = (\!|k', \oplus', \otimes'|\!)$, and $f'_r = (\!|k'_r, \oplus'_r, \otimes'_r|\!)$, and g and k be functions, and with these functions the function f be defined as follows.

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

The function f can be implemented by the combinations of the rose tree skeletons as follows.

$$\begin{aligned} f\ c\ t = \textbf{let}\ & lt = \mathsf{rAcc}_r\ (\circledast_l)\ (\mathsf{uAcc}_r\ (\oplus'_l)\ (\otimes'_l)\ (\mathsf{map}_r\ k'_{\ l}\ t)) \\ & t' = \mathsf{uAcc}_r\ (\oplus')\ (\otimes')\ (\mathsf{map}_r\ k'\ t) \\ & rt = \mathsf{lAcc}_r\ (\circledast_r)\ (\mathsf{uAcc}_r\ (\oplus'_r)\ (\otimes'_r)\ (\mathsf{map}_r\ k'_{\ r}\ t)) \\ & gt = setroot_r\ \iota_\odot\ (\mathsf{zipwith4}_r\ g\ (\mathsf{getparent}_r\ t)\ lt\ t'\ rt) \\ & dt = \mathsf{dAcc}_r\ (\odot)\ gt \\ \textbf{in}\ & \mathsf{zipwith3}_r\ (\lambda a\ d\ d'.k\ a\ (c \odot d \odot d'))\ t\ dt\ gt \end{aligned}$$

The $\mathsf{zipwith4}_r$ skeleton is a generalization of the $\mathsf{zipwith}_r$ skeleton to accept four rose trees of the same shape.

Proof: Though we can prove this theorem by induction on the structure of rose trees, it becomes pessimistically long. Thus in this paper, we prove the theorem by proving the derivation steps.

First, we prove the correctness of the pre-processing of lt, t', and rt. The correctness of the pre-processing of t' is proved from the following equation, which is proved by definition.

$$\mathsf{reduce}_r\ (\oplus')\ (\otimes')\ t = root_r\ (\mathsf{uAcc}_r\ (\oplus')\ (\otimes')\ t)$$

The correctness of the pre-processing of lt is proved from the equation above and the following equation on list

$$\textstyle\sum_{\circledast_l}[a_i \mid j \in [1..i-1]] = last\ (scan\ (\circledast_l)\ [a_j \mid j \in [1..i]])$$

where *last* is a function to extract the last value from the list. The pre-processing of rt is the symmetry. This equation can be proved immediately by induction.

Then we prove the correctness of the derivation of the skeletal program. This derivation is almost the same as the Theorem 4, and we can prove it by induction on the structure of rose trees just in the same way. □

5.5 Diffusion Strategy

In general, programmers specify the algorithms using recursive functions rather than skeletons. We shall highlight a strategy of how to derive skeletal parallel programs based on the diffusion theorems. The derivation of skeletal parallel programs can be carried out by the following four steps.

1. Write a specification as a recursive function.
2. Derive associative operators and parallelizable homomorphisms.
3. Apply the diffusion theorems.
4. Optimize the derived skeletal parallel programs.

In the following, we show more in detail for each step. Examples of deriving parallel programs will be given in Section 6.

Write a Specification as a Recursive Function Many algorithms on rose trees are naturally specified by means of recursive functions. Therefore in our strategy, we first write the specification as a recursive function on rose trees. The recursive function should be in the form of either of the diffusion theorems.

The initial specification may not be exactly forms of the diffusion theorems. In such cases, we try to derive a desired one by applying program transformation techniques. We have techniques such as the tupling technique [30], the fusion technique [15], and the condition normalization techniques [14]. For example, if there are multiple traversals on a tree then we can merge them by the tupling transformation [30]. We will apply the fusion techniques to derive a recursive function in the example of the party planning problem in Section 6.3.

Derive Associative Operators and Parallelizable Homomorphisms We then check the conditions for applying the diffusion theorems. The operators used in the recursive function should be associative or satisfy the condition of parallelizable homomorphisms. When the operators satisfy these conditions, we do nothing more in this step.

When the operators do not satisfy the conditions we derive suitable operators by using the techniques such as the generalization technique and the context preservation technique [13]. We show these derivations of associative operators and parallelizable homomorphisms in Sections 6.2 and 6.3. In many cases, after deriving suitable operators we need to accordingly rewrite the specification given in the last step.

Apply the Diffusion Theorems After checking the conditions, we can easily apply the diffusion theorem to obtain a program in terms of parallel skeletons.

Optimize the Derived Skeletal Parallel Programs Since the diffusion theorems in this section are developed on general recursive functions, the derived skeletal parallel programs may have redundancy.

For example, consider a recursive function that computes the result based on the accumulative parameter only, and let us apply a diffusion theorem (e.g. Theorem 3). The theorem is redundant in the sense the result is computed from not only the accumulation parameter but also the original value. The derived skeletal program has a $\mathsf{zipwith}_r$ skeleton, but in fact the computation can be performed simply with the map_r skeleton.

Therefore, for the last step, we optimize the derived skeletal parallel programs by the replacement of the skeletons and/or the fusion of successive skeletons' calls.

6 Examples

In this section, we demonstrate the derivation of parallel programs in terms of the rose-tree skeletons based on the diffusion theorems. We derive skeletal programs for three interesting examples, the pre-order numbering problem, the breadth first numbering problem, and the party planning problem.

6.1 Pre-order Numbering Problem

The pre-order numbering problem on a rose tree takes a rose tree and assigns a number for each node in the order of the pro-order traversal. In the pre-order numbering, the value of a left-most child is larger that that of its parent by one, and the value of another child is larger than its left sibling by the number of nodes in the left-sibling's subtree. Thus, the value of a node is larger than its parent by the sum of one and the number of nodes in the left-siblings' subtrees. A recursive algorithm that solves the pre-order numbering problem is as follows.

$$\begin{aligned} pre'\ c\ (RNode\ a\ ts) = RNode\ c\ [pre'\ c_i\ t_i \mid i \in [1..\#ts]] \\ \textbf{where}\ c_i = c + 1 + l_i \\ l_i = \textstyle\sum_+ [size\ t_j \mid j \in [1..i-1]] \end{aligned}$$

The function *size* counts the number of nodes in a rose tree, and is defined as a parallelizable homomorphism, $size = (\!|\lambda x.1, +, +|\!)$.

Since the operator $+$ is associative, the operator satisfies the conditions of the diffusion theorems. Since in the update of the accumulative parameter c depends on the left-siblings, we apply Theorem 7 to this function. By simply collating the function pre' and the recursive

specification of the theorem, we have $k = \lambda a\ c.c$, $\odot = +$, $g = \lambda a\ l.1 + l$, $\circledast_l = +$, and $(\![k'_l, \oplus'_l, \otimes'_l]\!) = (\![\lambda x.1, +, +]\!)$. The intermediate values t'_i and r_i are not used.

By substituting the functions and operators above to the skeletal program in the theorem, we obtain the following parallel programs. Since the intermediate values of t'_i and r_i are not used, we omit the computation of t' and rt and use the $\mathsf{zipwith}_r$ skeleton instead of the $\mathsf{zipwith4}_r$ skeleton.

$$\begin{aligned} pre'\ t\ c = \mathbf{let}\ & lt = \mathsf{rAcc}_r\ (+)\ (\mathsf{uAcc}_r\ (+)\ (+)\ (\mathsf{map}_r\ (\lambda x.1)\ t)) \\ & gt = setroot_r\ 0\ (\mathsf{zipwith}_r\ (\lambda a\ l.1 + l)\ (\mathsf{getparent}_r\ t)\ lt) \\ & dt = \mathsf{dAcc}_r\ (+)\ gt \\ \mathbf{in}\ & \mathsf{zipwith3}_r\ (\lambda a\ d\ d'.c + d + d')\ t\ dt\ gt \end{aligned}$$

We can simplify the obtained parallel program. Firstly since the initial value of c is 0, we can substitute it. Secondly, since the function of the $\mathsf{zipwith}_r$ skeleton does not depend on its first argument, we can remove the first tree and modify the skeleton into the map_r skeleton. Similarly, since the function of the $\mathsf{zipwith3}_r$ skeleton does not depend on its first argument, we can remove the first tree and modify the skeleton into the $\mathsf{zipwith}_r$ skeleton. With these optimizations, we obtain the following simpler parallel program for the pre-order numbering.

$$\begin{aligned} pre\ t = \mathbf{let}\ & lt = \mathsf{rAcc}_r\ (+)\ (\mathsf{uAcc}_r\ (+)\ (+)\ (\mathsf{map}_r\ (\lambda x.1)\ t)) \\ & gt = setroot_r\ 0\ (\mathsf{map}_r\ (\lambda l.1 + l)\ lt) \\ & dt = \mathsf{dAcc}_r\ (+)\ gt \\ \mathbf{in}\ & \mathsf{zipwith}_r\ (+)\ dt\ gt \end{aligned}$$

6.2 Breadth First Numbering Problem

The breadth first numbering problem takes a rose tree and assigns a number for each node in the order of breadth first traversal. In this section, we provide an $O(d \log n)$ parallel algorithm for the breadth first numbering problem where n denotes the number of nodes and d denotes the maximum depth of the input rose tree.

Before defining a recursive algorithm, we introduce two functions and one operator for manipulating lists. Function *take* n returns the first n values of the input list. When there are less values in the input list, this function simply returns all of the values in the list.

$$\begin{aligned} take\ 0\ as &= [\,] \\ take\ n\ [\,] &= [\,] \\ take\ n\ (a : as) &= a : take\ (n-1)\ as \end{aligned}$$

Function *drop* n removes the first n values from the input list. When there are less values in the input list, this function returns the empty list $[\,]$.

$$\begin{aligned} drop\ 0\ as &= as \\ drop\ n\ [\,] &= [\,] \\ drop\ n\ (a : as) &= drop\ (n-1)\ as \end{aligned}$$

Operator $\Upsilon_\oplus$ takes two lists and computes with $\oplus$ for each pair of corresponding values. If the lengths of the lists are different, the values running off from the other list are not discarded but simply follow the results. Note that this definition is different from the familiar definition of the *zipwith* function for lists.

$$\begin{aligned} (a : as)\ \Upsilon_\oplus\ (b : bs) &= (a \oplus b) : (as\ \Upsilon_\oplus\ bs) \\ (a : as)\ \Upsilon_\oplus\ [\,] &= a : as \\ [\,]\ \Upsilon_\oplus\ bs &= bs \end{aligned}$$

If the operator $\oplus$ is associative, then the operator $\Upsilon_{\oplus}$ is also associative with its unit $[\,]$.

Firstly we provide a recursive algorithm for the breadth first numbering problem. In the breadth first numbering, we need to know the number of nodes in each depth for each node. The number of nodes is one in the depth 0, and the numbers of nodes in the deeper depths are computed by the summation among the children for each depth. Thus we can write a recursive bottom-up function that computes the number of nodes in each depth as follows.

$$nodes\ (RNode\ a\ ts) = [1] +\!\!+ \sum\nolimits_{\Upsilon_+} [nodes\ t_i \mid i\ \in\ [1..\#ts]]$$

In the breadth first numbering, the value of a node at depth d is larger than that of its parent node by the sum of one and the following two numbers:

- the number of nodes which are at the right of the parent node in the depth $d-1$ (the parent's depth), and
- the number of nodes which are at the left of the node in the depth d (the node's depth).

We need the number of nodes distant from a node in computing the value of the node, and we pass the information about the number of nodes out of the subtree for each depth as an accumulative parameter cs. We also need another accumulative parameter c for representing the value that should be assigned to nodes, and therefore we pass the accumulative parameters as a pair. With the function *nodes* and these accumulative parameters, we can specify a recursive algorithm that solves the breadth first numbering problem as follows. In the specification, l_i is the accumulation of the numbers of nodes in the left-siblings' subtrees, and r_i is the accumulation of the numbers of nodes in the right-siblings' subtrees.

$$\begin{aligned}
&bfn\ t = bfn'\ (0, [\,])\ t \\
&bfn'\ (c, cs)\ (RNode\ a\ ts) = RNode\ c\ [bfn'\ (c_i, cs_i)\ t_i \mid i \in [1..\#ts]] \\
&\qquad\qquad \textbf{where}\ c_i\ \ = c + head\ cs + 1 + head\ l_i \\
&\qquad\qquad\qquad\quad cs_i = tail\ cs\ \Upsilon_+\ tail\ l_i\ \Upsilon_+\ r_i \\
&\qquad\qquad\qquad\quad l_i\ \ = \textstyle\sum_{\Upsilon_+} [nodes\ t_j \mid j \in [1..i-1]] \\
&\qquad\qquad\qquad\quad r_i\ \ = \textstyle\sum_{\Upsilon_+} [nodes\ t_j \mid j \in [i+1..\#ts]]
\end{aligned}$$

In this definition, for empty list $[\,]$ the function *head* and *tail* return 0 and $[\,]$, respectively.

In order to apply one of the diffusion theorems, we should rewrite the top-down computation so that it updates the accumulative parameters (c, cs) with an associative operator. We may derive such an associative operator by using the *context preservation technique* [13]. The idea of the context preservation theorem is that we can derive an associative operator if we find a parametrized function which is closed under function composition. To derive such a parametrized function, we first abstract sub-terms which do not depend on both c and cs. By this abstraction we have the following parametrized function in which α and β are parameters.

$$\lambda(c, cs).(c + head\ cs + \alpha, tail\ cs\ \Upsilon_+\ \beta)$$

Here, functions *head* and *tail* become obstacles in finding closed parametrized function. We replace the functions with *take* and *drop* respectively, and obtain the following parametrized function.

$$\lambda(c, cs).(c + \textstyle\sum_+ (take\ 1\ cs) + \alpha, drop\ 1\ cs\ \Upsilon_+\ \beta)$$

After abstracting 1 into another parameter γ, we have the following parametrized function.

$$\lambda(c, cs).(c + \textstyle\sum_+ (take\ \gamma\ cs) + \alpha, drop\ \gamma\ cs\ \Upsilon_+\ \beta)$$

We now validate the closure property of this parametrized function by calculating the function composition of it. Here we only show the result of the calculation since the calculations are rather straightforward.

$$
\begin{aligned}
&\lambda(c, cs).(c + \textstyle\sum_+(take\ \gamma'\ cs) + \alpha', drop\ \gamma'\ cs\ \Upsilon_+\ \beta') \\
&\quad \circ \lambda(c, cs).(c + \textstyle\sum_+(take\ \gamma\ cs) + \alpha, drop\ \gamma\ cs\ \Upsilon_+\ \beta) \\
&= \lambda(c, cs).(c + \textstyle\sum_+(take\ \gamma''\ cs) + \alpha'', drop\ \gamma''\ \Upsilon_+\ \beta'') \\
&\quad \textbf{where}\ \alpha'' = \textstyle\sum_+(take\ \gamma'\ \beta) + \alpha' + \alpha \\
&\qquad\qquad \beta'' = drop\ \gamma'\ \beta\ \Upsilon_+\ \beta' \\
&\qquad\qquad \gamma'' = \gamma + \gamma'
\end{aligned}
$$

Based on the closed parametrized function above, we derive an associative operator $\otimes$ for the top-down computation and the unit of $\otimes$ as follows.

$$
\begin{aligned}
&(\alpha, \beta, \gamma) \otimes (\alpha', \beta', \gamma') = (\textstyle\sum_+(take\ \gamma'\ \beta) + \alpha' + \alpha, drop\ \gamma'\ \beta\ \Upsilon_+\ \beta', \gamma + \gamma') \\
&\iota_\otimes = (0, [\,], 0)
\end{aligned}
$$

Using this associative operator, we rewrite the algorithm. We first lift the accumulative parameter to a tuple of (α, β, γ), and then perform computation using the associative operator $\otimes$, and finally put the results down to the desired ones. The initial value of the new accumulative parameter is the unit $(0, [\,], 0)$, and the final computation is given by the first value of the definition of the parametrized function, i.e. $\lambda\ (\alpha, \beta, \gamma).c + \sum_+(take\ \gamma\ cs) + \alpha$. Since the initial values of c and cs are 0 and $[\,]$ respectively, we substitute them and obtain simpler definition $\lambda\ (\alpha, \beta, \gamma).\alpha$. The obtained recursive algorithm is as follows.

$$
\begin{aligned}
&bfn\ t = bfn''\ (0, [\,], 0)\ t \\
&bfn''\ (\alpha, \beta, \gamma)\ (RNode\ a\ ts) = RNode\ \alpha\ [bfn''\ (\alpha_i, \beta_i, \gamma_i)\ t_i \mid i \in [1..\#ts]] \\
&\qquad\qquad \textbf{where}\ (\alpha_i, \beta_i, \gamma_i) = (\alpha, \beta, \gamma) \otimes (\alpha'_i, \beta'_i, 1) \\
&\qquad\qquad\qquad \alpha'_i = 1 + head\ l_i \\
&\qquad\qquad\qquad \beta'_i = tail\ l_i\ \Upsilon_+\ r_i \\
&\qquad\qquad\qquad l_i = \textstyle\sum_{\Upsilon_+}[nodes\ t_j \mid j \in [1..i-1]] \\
&\qquad\qquad\qquad r_i = \textstyle\sum_{\Upsilon_+}[nodes\ t_j \mid j \in [i+1..\#ts]]
\end{aligned}
$$

We then verify the function *nodes* to be a parallelizable homomorphism. The recursive definition of *nodes* has the form of parallelizable homomorphism $(\![k', \oplus', \otimes']\!)$ with $k' = \lambda a.[1]$, $\oplus' = +\!\!+$, and $\otimes' = \Upsilon_+$. Though the operators $+\!\!+$ and Υ_+ are respectively associative, the operator $+\!\!+$ does not distribute over the operator Υ_+, and the operator Υ_+ does not distribute over the operator $+\!\!+$ either.

$$
\begin{aligned}
&a +\!\!+ (b\ \Upsilon_+\ c) \neq (a +\!\!+ b)\ \Upsilon_+\ (a +\!\!+ c) \\
&a\ \Upsilon_+\ (b +\!\!+ c) \neq (a\ \Upsilon_+\ b) +\!\!+ (a\ \Upsilon_+\ c)
\end{aligned}
$$

We now validate the operator Υ_+ to be extended-distributive over $+\!\!+$. Since Υ_+ is not only associative but also commutative, we calculate a simpler definition (Lemma 1). We again show the result of the calculations only.

$$
\begin{aligned}
(\lambda x.a +\!\!+ (b\ \Upsilon_+\ x)) \circ (\lambda x.a' +\!\!+ (b'\ \Upsilon_+\ x)) &= \lambda x.A +\!\!+ (B\ \Upsilon_+\ x) \\
&\quad \textbf{where}\ A = a +\!\!+ (take\ (\#a')\ b\ \Upsilon_+\ a') \\
&\qquad\quad B = drop\ (\#a')\ b\ \Upsilon_+\ b'
\end{aligned}
$$

From the calculation result, the operator Υ_+ is surely extended-distributive over $+\!\!+$, and the function *nodes* is a parallelizable homomorphism.

Since the accumulative parameter is updated with the values of the left-siblings and the right-siblings, we apply Theorem 7. By a simple matching the algorithm bfn'' with the recursive specification of the theorem, we have $k = \lambda a\ (\alpha, \beta, \gamma).\alpha$, $\odot = \otimes$, $g = \lambda a\ l\ r.(1 + head\ l, tail\ l\ \Upsilon_+\ r, 1)$, $\circledast_l = \circledast_r = \Upsilon_+$, and $f'_l = f'_r = (\![\lambda x.[1], +\!\!+, \Upsilon_+]\!)$. The intermediate

value t' is not used. By substituting the function and the operators above to the skeletal program of the theorem, we obtain the following skeletal program. Note that the two uAcc_rs and map_rs are merged into ones, that the computation of t' is omitted, and that the $\mathsf{zipwith4}_r$ skeleton is simplified into the $\mathsf{zipwith3}_r$ skeleton. (We omit the definitions of g and $\otimes$ in the following program.)

$$
\begin{aligned}
bfn''\ c\ t = \mathbf{let}\ & ut = \mathsf{uAcc}_r\ (+)\ (\Upsilon_+)\ (\mathsf{map}_r\ (\lambda x.[1])\ t) \\
& lt\ = \mathsf{rAcc}_r\ (\Upsilon_+)\ ut \\
& rt\ = \mathsf{lAcc}_r\ (\Upsilon_+)\ ut \\
& gt\ = setroot_r\ (0, [\,], 0)\ (\mathsf{zipwith3}_r\ g\ (\mathsf{getparent}_r\ t)\ lt\ rt) \\
& dt = \mathsf{dAcc}_r\ (\otimes)\ gt \\
\mathbf{in}\ & \mathsf{zipwith3}_r\ (\lambda a\ d\ d'.fst\ (c \otimes d \otimes d'))\ t\ dt\ gt
\end{aligned}
$$

The function *fst* is a function that returns the first value of the tuple.

We can simplify the obtained parallel program: since the functions for both of the $\mathsf{zipwith3}_r$ skeletons do not depend on the first argument, we replace the skeletons by the $\mathsf{zipwith}_r$ skeleton. We can also substitute the initial value of the accumulative parameter. Finally, we obtain the following parallel program for the breadth first numbering problem.

$$
\begin{aligned}
bfn\ t = \mathbf{let}\ & ut = \mathsf{uAcc}_r\ (+)\ (\Upsilon_+)\ (\mathsf{map}_r\ (\lambda x.[1])\ t) \\
& lt\ = \mathsf{rAcc}_r\ (\Upsilon_+)\ ut \\
& rt\ = \mathsf{lAcc}_r\ (\Upsilon_+)\ ut \\
& gt\ = setroot_r\ (0, [\,], 0)\ (\mathsf{zipwith}_r\ g'\ lt\ rt) \\
& dt = \mathsf{dAcc}_r\ (\otimes)\ gt \\
\mathbf{in}\ & \mathsf{zipwith}_r\ (\lambda\ d\ d'.fst\ (d \otimes d'))\ dt\ gt \\
\mathbf{where}\ & g'\ l\ r = (1 + head\ l, tail\ l\ \Upsilon_+\ r, 1) \\
& (\alpha, \beta, \gamma) \otimes (\alpha', \beta', \gamma') = (\textstyle\sum_+(take\ \gamma'\ \beta) + \alpha' + \alpha, drop\ \gamma'\ \beta\ \Upsilon_+\ \beta', \gamma + \gamma')
\end{aligned}
$$

6.3 Party Planning Problem

As the third example, we derive a parallel program for a dynamic programming problem on rose trees called the party planning problem [20]. The party planning problem is a generalization of the maximum independent-set sum problem whose parallelization is discussed in [28], and is an instance of maximum marking problems studied by several researchers in the context of derivation of sequential programs [8, 55].

> 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, and the personnel office has ranked each employee with a conviviality rating of a real number. Given the structure of the company and the ratings of employees, the problem is to mark the guests so that the sum of the conviviality ratings of marked guests is its maximum.

To simplify our problem, we assume the conviviality rating associated to each node to be positive.

It is helpful to derive an algorithm for computing the maximum sum before deriving one for the marking problem. The condition of this marking problem is that no two adjacent nodes should be marked. In the following, we call a set of node, whose every two marked nodes are not adjacent, as an independent-set, and then the subproblem becomes to find the maximum of all independent-set sum. Based on the dynamic programming technique, we can specify the subproblem as the following recursive algorithm that returns a pair of values. The values returned by the following mis' function are

– i: the maximum sum of all the independent-sets in which the root node is included, and
– e: the maximum sum of all the independent-sets in which the root node is excluded.

$$\begin{aligned}
mis\ t &= \textbf{let}\ (i,e) = mis'\ t\ \textbf{in}\ r \uparrow s \\
mis'\ (RNode\ a\ ts) &= \textbf{let}\ ies = [mis'\ t_j \mid j \in [1..\#ts]] \\
&\qquad\quad e' = \textstyle\sum_+ [e_j \mid j \in [1..\#ts]] \\
&\qquad\quad m' = \textstyle\sum_+ [i_j \uparrow e_j \mid j \in [1..\#ts]] \\
&\quad\ \textbf{in}\ \ (a + e', m')
\end{aligned}$$

By introducing a function g defined as $g\ (i,e) = (e, i \uparrow e)$, and an operator $\otimes$ defined as $(e,m) \otimes (e',m') = (e+e', m+m')$, we can specify the function mis' in a simpler form as follows. Here, function *snd* returns the second value.

$$\begin{aligned}
mis'\ (RNode\ a\ ts) = \textbf{let}\ &em = \textstyle\sum_\otimes [g\ (mis'\ t_j) \mid j\ \textbf{in}\ [1..\#ts]] \\
\textbf{in}\ \ &(a + fst\ em, snd\ em)
\end{aligned}$$

This function is not in the form of parallelizable homomorphisms since the function g is applied after the recursive calls and before folding. To make it a parallelizable homomorphism, we fuse the function g and mis' and derive the following function $mis'' = g \circ mis'$. We need to modify the top case of *mis* accordingly.

$$\begin{aligned}
mis\ t &= snd\ (mis''\ t) \\
mis''\ (RNode\ a\ ts) &= \textbf{let}\ em = \textstyle\sum_\otimes [mis''\ t_j \mid j\ \textbf{in}\ [1..\#ts]] \\
&\quad\ \textbf{in}\ \ (snd\ em, (a + fst\ em) \uparrow snd\ em)
\end{aligned}$$

We now validate that the function mis'' is a parallelizable homomorphism, by showing the function and the operators satisfy the condition of parallelizable homomorphisms. Firstly, the operator $\otimes$ is associative because of the associativity of the operator $+$. Secondly, we may give a specification of the function k and the operator $\oplus$ as follows from the definition of mis'' above.

$$\begin{aligned}
k\ a &= a \\
a \oplus (e,m) &= (m, (a+e) \uparrow m)
\end{aligned}$$

It is easy to prove that the operator $\oplus$ above is not distributive over $\otimes$ as shown in the following calculations.

$$\begin{aligned}
a \oplus ((e,m) \otimes (e',m')) &= (m+m', (a+e+e') \uparrow (m+m')) \\
(a \oplus (e,m)) \otimes (a \oplus (e',m')) &= (m+m', ((a+e) \uparrow m) + ((a+e') \uparrow m'))
\end{aligned}$$

It is also easy prove that the operator $\otimes$ is not distributive over the operator $\oplus$ because of mismatch of the types. We thus turn to validate the associative and commutative operator $\otimes$ is extended-distributive over the operator $\oplus$. We show the results of the calculations here since the computation is rather straightforward.

$$\begin{aligned}
&(\lambda(x_e,x_m).\ a \oplus ((e,m) \otimes (x_e,x_m))) \circ (\lambda(x_e,x_m).\ a' \oplus ((e',m') \otimes (x_e,x_m))) \\
&= \lambda(x_e,x_m).\ ((m+a+e'+x_e) \uparrow (m+m'+x_m), \\
&\qquad\qquad ((m+a+e'+x_e) \uparrow (((a+e+m') \uparrow (m+m')) + x_m))) \\
&\lambda(x_e,x_m).\ A \oplus ((E,M) \otimes (x_e,x_m)) \\
&= \lambda(x_e,x_m).\ (M+x_m, (A+E+x_e) \uparrow (M+x_m))
\end{aligned}$$

As seen from the calculation, the operator $\otimes$ is not extended-distributive over the operator $\oplus$. To derive an operator satisfying the condition, we generalize the definitions of the function

k and the operator $\oplus$ satisfying $k'\ a \oplus' (e,m) = k\ a \oplus (e,m)$. A generalization is given as follows.

$$\begin{aligned}
&k'\ a = (-\infty, 0, a, 0)\\
&(\alpha,\beta,\gamma,\delta)\ \oplus'\ (e,m) = ((\alpha+e)\uparrow(\beta+m), (\gamma+e)\uparrow(\delta+m))
\end{aligned}$$

With this generalized definition of $\oplus'$, we can prove that the operator $\otimes$ is extended-distributive over the operator $\oplus'$ with the following characteristic functions.

$$\begin{aligned}
&p_1\ ((a,b,c,d),(e_1,m_1),(e_2,m_2),(a',b',c',d'),(e'_1,m'_1),(e'_2,m'_2))\\
&\quad = ((a+e_1+e_2+a'+e'_1+e'_2) \quad \uparrow (b+m_1+m_2+c'+e'_1+e'_2),\\
&\qquad (a+e_1+e_2+b'+m'_1+m'_2)\uparrow(b+m_1+m_2+d'+m'_1+m'_2),\\
&\qquad (c+e_1+e_2+a'+e'_1+e'_2) \quad \uparrow (d+m_1+m_2+c'+e'_1+e'_2),\\
&\qquad (c+e_1+e_2+b'+m'_1+m'_2)\uparrow(d+m_1+m_2+d'+m'_1+m'_2))\\
&p_2\ ((a,b,c,d),(e_1,m_1),(e_2,m_2),(a',b',c',d'),(e'_1,m'_1),(e'_2,m'_2)) = (0,0)\\
&p_3\ ((a,b,c,d),(e_1,m_1),(e_2,m_2),(a',b',c',d'),(e'_1,m'_1),(e'_2,m'_2)) = (0,0)
\end{aligned}$$

Therefore the function mis'' is in fact a parallelizable homomorphism defined as $mis'' = (\![k', \oplus', \otimes]\!)$.

By using this mis'' function, we can write a recursive function that solves the party planning problem as follows. Here, the accumulative parameter denotes whether the parent is selected or not, and thus its initial value should be *False*.

$$\begin{aligned}
&ppp\ t &&= ppp'\ False\ t\\
&ppp'\ c\ (RNode\ a\ ts) &&= \mathbf{let}\ em = mis''\ (RNode\ a\ ts)\\
&&&\qquad\quad c' \ = \neg c \wedge (fst\ em < snd\ em)\\
&&&\quad\ \mathbf{in}\ \ RNode\ c'\ [ppp'\ c'\ t_j \mid j \in [1..\#ts]]
\end{aligned}$$

In this definition the accumulative parameter is updated as $c' = \neg c \wedge (fst\ em < snd\ em)$, but we require the update should be in the form of $c' = c \odot g\ a\ em$ with some operator $\odot$. It is worth noting that the domain of the accumulative parameter is finite, i.e. *True* and *False*. It is widely known that the finiteness of the domain is helpful in deriving associative operators [19, 28, 36, 60], where the main idea is to compute the results for all the possible values in the domain. Based on this idea, we can derive the following recursive function. Here, the operator $\odot$ is associative.

$$\begin{aligned}
&ppp\ t &&= ppp''\ (True, False)\ t\\
&ppp''\ (c_t,c_f)\ (RNode\ a\ ts) &&= \mathbf{let}\ em \quad\ = mis''\ (RNode\ a\ ts)\\
&&&\qquad\quad (c'_t,c'_f) = (c_t,c_f)\odot(False, fst\ em < snd\ em)\\
&&&\quad\ \mathbf{in}\ \ RNode\ c'_f\ [ppp''\ (c'_t,c'_f)\ t_j \mid j \in [1..\#ts]]\\
&&&\quad\ \mathbf{where}\ (c_t,c_f)\odot(c'_t,c'_f) = (\mathbf{if}\ c_t\ \mathbf{then}\ c'_t\ \mathbf{else}\ c'_f,\\
&&&\qquad\qquad\qquad\qquad\quad \mathbf{if}\ c_f\ \mathbf{then}\ c'_t\ \mathbf{else}\ c'_f)
\end{aligned}$$

We have developed a recursive algorithm, we now apply the diffusion theorem to derive a skeletal parallel program. This recursive function is very similar to the recursive specification in Theorem 6, whereas the difference is that the overall results are also computed with the result of the bottom-up computation. Therefore, we slightly modify Theorem 6 to use the result of uAcc_r skeleton instead of the original tree, and apply it to obtain a skeletal parallel program. By matching the definitions of the functions with the specification, we have $k\ a'\ (c_t,c_f) = \neg c_f \wedge (fst\ a' < snd\ a')$, and $g\ a\ (e,m) = (False, e < m)$, and the others are already derived. By substituting these functions, we successfully obtain a skeletal parallel program as follows. (We omit the definitions of $\oplus'$, $\otimes$, k' and $\odot$ here.)

$$\begin{aligned}
ppp'\ (c_t,c_f)\ t = \mathbf{let}\ &t' = \mathsf{uAcc}_r\ (\oplus')\ (\otimes)\ (\mathsf{map}_r\ k'\ t)\\
&dt = \mathsf{dAcc}_r\ (\odot)\ (\mathsf{zipwith}_r\ (\lambda a\ (e,m).(False, e<m))\ t\ t')\\
\mathbf{in}\ \ &\mathsf{zipwith}_r\ (\lambda(e,m)\ d.\neg(\mathbf{if}\ c_f\ \mathbf{then}\ fst\ d\ \mathbf{else}\ snd\ d) \wedge (e<m))\ t'\ dt
\end{aligned}$$

We can simplify the derived skeletal parallel programs. One may have noticed that the initial value of the accumulative parameter is fixed as $(c_t, c_f) = (True, False)$, and one of arguments in the $\mathsf{zipwith}_r$ skeleton is not used. Based on these facts, we can derive more efficient parallel program as follows.

$$
\begin{aligned}
ppp\ t = \ & \textbf{let}\ t' = \mathsf{uAcc}_r\ (\oplus')\ (\otimes)\ (\mathsf{map}_r\ k'\ t) \\
& \quad dt = \mathsf{dAcc}_r\ (\odot)\ (\mathsf{map}_r\ (\lambda(e,m).(False, e < m))\ t') \\
& \textbf{in}\ \ \mathsf{zipwith}_r\ (\lambda(e,m)\ d.\neg(snd\ d) \wedge (e < m))\ t'\ dt \\
& \textbf{where}\ k'\ a = (-\infty, 0, a, 0) \\
& \qquad (\alpha,\beta,\gamma,\delta) \oplus' (e,m) = ((\alpha + e) \uparrow (\beta + m), (\gamma + e) \uparrow (\delta + m)) \\
& \qquad (e,m) \otimes (e',m') = (e + e', m + m') \\
& \qquad (c_t, c_f) \odot (c'_t, c'_f) = (\textbf{if}\ c_t\ \textbf{then}\ c'_t\ \textbf{else}\ c'_f, \textbf{if}\ c_f\ \textbf{then}\ c'_t\ \textbf{else}\ c'_f)
\end{aligned}
$$

7 Implementation of Rose-Tree Skeletons with SkeTo

We have implemented the rose-tree skeletons based on the binary-tree skeletons in our skeleton library. In this section, we show the implementation of rose-tree skeletons with source-code and then report experimental results.

7.1 Overview of Our Tree Skeleton Library

We are implementing a parallel skeleton library in the SkeTo Project. The SkeTo library [41] is a library based on the theory of Constructive Algorithmics [7] and currently provides parallel skeletons for lists, matrices, and binary trees. The SkeTo library is implemented in standard C++ and MPI.

The binary-tree skeletons in the SkeTo library are implemented based on the tree contraction algorithms with some modifications to perform good scalability even on the distributed-memory environments. We utilize m-bridge technique [54] to partition trees into segments and perform computations on the distributed trees. Such partition and distribution of the trees are done implicitly by the `dist_tree` class.

Our SkeTo library provides not only basic five binary-tree skeletons but also several communication skeletons. The skeletons are implemented to take function objects for their functional arguments and instances of the `dist_tree` class for the input tree. We adopt the function objects since we can deal functions in a smart way with the function objects in the C++ programs.

We show the interfaces of binary-tree skeletons map_b, uAcc_b, and getchl_b (`shiftl`) in Fig. 12, which are used later in implementing rose-tree skeletons. One can download the source-code of the skeletons from our project's website[2].

7.2 Implementation of Rose-Tree Skeletons

We first show the implementation of the data structure for rose trees, and then the parallel skeletons using the map_r and lAcc_r skeletons for our 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. We can implement the class for the binary-tree representation by just using the class for distributed binary trees, `dist_tree`. The following is a segment of code to achieve this where `A` is a template type representing the type of node in a rose tree.

[2] SkeTo Project Homepage. `http://www.ipl.t.u-tokyo.ac.jp/sketo/`

```
class tree_skeletons
{
public:
  template< typename K1, typename K2 >
  static dist_tree< typename K1::result_type > *
  map( const K1 &k1, const K2 &k2,
       dist_tree< typename K1::argument_type > *t );

  template< typename K1, typename K2, typename PSI, typename PHIL,
            typename PHIR, typename GG >
  static dist_tree< typename K1::result_type > *
  uAcc( const K1 &k1, const K2 &k2, const PSI &psi,
        const PHIL &phiL, const PHIR &phiR, const GG &G,
        const dist_tree< typename K1::argument_type > *t );

  template< typename A >
  static dist_tree< A >*
  shiftl( const A& leafval, const dist_tree< A >* tree );
```

Fig. 12. Interfaces of binary-tree skeletons in the SkeTo library.

```
template< typename A >
class dist_rose_tree
{
  dist_tree< A >* btree;
  ...
```

In our implementation, we also provide several wrapper functions for input/output.

Implementation of the map_r skeleton We show the implementation of the map_r skeleton, which is the simplest skeleton of our seven skeletons.

We implement the rose-tree skeletons in another class `rose_tree_skeletons`. One reason is to separate the parallel data structures and their operations. The other reason is to enable the access to the private members of the `dist_rose_tree` class; template classes do not permit to access to their private members if they are instantiated with different template parameters.

Interface of the map_r skeleton is given in the following code. The functional 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 following 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.

Function objects are one of the outstanding features of the C++, and they can be manipulated in a smart way with the template mechanism. For example, we can implement function compositions and partial bindings. Striegnitz [61] discussed features of C++ for skeletal parallel programming.

For the map_r skeleton, we need to define the function denoted by _, which does nothing but keeping consistency of the types. We 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 { return static_cast< B >( 0 ); }
};
```

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 will be instantiated from the `map` function. It is worth noting that with the interface of the `map` function above we need not to specify template types, because they are identified by the compiler, while we need to specify `B` for the following function `map_adapter`. 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 > *bt
    = tree_skeletons::map( UnaryUndef< A, B >( ), k, t->btree );
  return new dist_rose_tree< B >( bt );
}
```

Implementation of lAcc_r skeleton We show the implementation of another more complicated skeleton lAcc_r. In the implementation of the lAcc_r skeleton, we utilize triple (p, a, b) in the upward computation on the binary-tree representation. We define the triple as the following structure `lAcc_in_t`.

```
template < typename A >
struct lAcc_in_t {
  bool p; A a; A b;
  lAcc_in_t( bool p_, const A& a_, const A& b_ )
    : p( p_ ), a( a_ ), b( b_ ) { }
};
```

As in the case of the map_r skeleton, we implement the function objects for the 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_L, ψ_R and G for the uAcc_b skeleton. The function object for ϕ_L is defined as the following `func_lAcc_psiL`.

```
template < typename A, typename OPLUS >
struct func_lAcc_psiL : public skeleton::ternary_function<
                          lAcc_in_t< A >, A, lAcc_in_t< A >, lAcc_in_t< A > > {
  const OPLUS &oplus;
  func_lAcc_psiL( OPLUS oplus_ ) : oplus( oplus_ ) {};
  lAcc_in_t< A > operator()( const lAcc_in_t< A >& arg1, const A& /* l */,
                             const lAcc_in_t< A >& arg2 ) const {
    return lAcc_in_t< A >( arg1.p && arg2.p, oplus( arg1.a, arg2.a ),
                           arg1.p ? oplus( arg1.a, arg2.b ) : arg1.b );
  }
};
```

The other functions are defined in the same way.

After defining the function objects, we can straightforwardly implement the skeleton as follows. Note that the uAcc_b skeleton in the SkeTo library includes the computation of the map_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 > *bt1 = tree_skeletons::uAcc( UnaryConst< A, A >( unit_oplus ),
      func_lAcc_k< A, OPLUS >( oplus ), func_lAcc_phi< A >( ),
      func_lAcc_psiL< A, OPLUS >( oplus ), func_lAcc_psiR< A, OPLUS >( oplus ),
      func_lAcc_G< A, OPLUS >( oplus ), t->btree );
  dist_tree< A > *bt2 = tree_skeletons::shiftr( unit_oplus, bt1 );
  if ( bt1 ) delete bt1;
  return new dist_rose_tree< A >( bt2 );
}
```

As seen so far, we can implement the lAcc_r skeleton without much effort using the function objects and the template mechanism in C++. The implemented skeletons however may be worse in efficiency due to the intermediate data structures passed between the skeletons. The main aim of the implementation in this paper is to verify the scalability of the parallel skeletons, and improvement of the efficiency of the parallel skeletons is our future work.

7.3 An Experiment

We have made an experiment using the rose-tree skeletons implemented above, to see the scalability of them. We used the derived skeletal program for the pre-order numbering problem in Section 6.1.

The environment is our PC cluster which consists of sixteen uniform PCs connected with Gigabit Ethernet. Each PC has a CPU of Pentium4 3.0GHz (Hyper Threading ON) and 1GB memory. The OS, the C++ compiler, and the MPI library are Linux 2.6.8, gcc 2.95, and mpich 1.2.6, respectively.

We carried out the derived program on two trees of $2^{22} - 1$ nodes (almost four million) with varying the number of CPUs used. The first tree is a perfect binary tree whose height is twenty-two, the second tree is a randomly generated tree whose height is seven (this height comes from the average height of XMLs [39]).

The experimental results are shown in Fig. 13, where the execution times do not include initial distribution and final gathering. For both data, the skeletal program shows good scalability; the execution times for the binary tree are 13.0 (sec) with one processor and 0.87 (sec) with sixteen processors, the execution times for the random tree are 12.5 (sec) with one processor and 1.41 (sec) with sixteen processors, and the speedup is 14.9 with sixteen processors in the case of the binary tree. In some cases, the execution times become a little worse, which is caused by the ill-balanced structure of trees and the difficulty in dividing trees into segments of almost the same size.

8 Related Work

Parallel Tree Skeletons Though trees are important data structures, it is known to be hard to write general and efficient parallel programs manipulating trees. This calls for helpful methods for parallel programming on trees and the skeletal approach is one of promising paradigms.

As domain specific skeletons, Deldari et al. [22] designed and implemented parallel skeletons for Constructive Solid Geometry (CSG), based on the parallel algorithms developed by Banerjee et al. [4]. For general-purpose tree skeletons, Skillicorn [59] formalized a set of

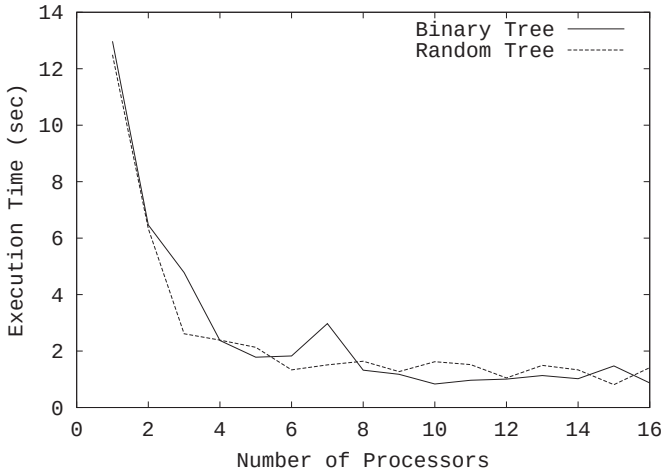

Fig. 13. Experimental Results

binary-tree skeletons and a set of rose-tree skeletons based on Constructive Algorithmics [6, 9, 34, 47]. The implementations of these binary-tree skeletons have been developed [25, 26, 43] based on the tree contraction algorithms [1, 48]. Parallel skeletons for general recursive data structures are also mentioned in [2].

Though several studies have worked on the skeletons for rose trees or general recursive types, formalization and implementation of the skeletons have been insufficient. This paper have tackled these problems and proposed a new set of rose-tree skeletons with their implementation. We believe our rose-tree skeletons are not only theoretically simple but also practically expressive.

Parallel Computation on Rose Trees and Nested Data Structures Parallel tree contraction algorithms are now the bases for efficient parallel computations on trees. Though the original idea proposed by Miller and Reif [48] did not limit the shape of trees to be binary trees, many researchers have developed more efficient tree contraction algorithms based on the assumption of binary trees [1, 19, 46].

According to the efficient tree contraction algorithms on binary trees, several studies developed parallel algorithms on rose trees based on the transformation of rose trees into binary trees. Cole and Vishkin [19], Diks and Hagerup [23], Skillicorn [59] and our previous paper [42] adopted binary-tree representations, in which some dummy nodes are inserted to expand internal nodes. Though these binary-tree representations suit for representing bottom-up computations and top-down computations, they are poor at representing computations among siblings. In this paper, we adopt another binary-tree representation discussed in [20], and this representation enables us to formalize computations among siblings. There are other representations of rose trees, for example, in the community of functional programming, the leaf-labeled binary trees are often used for representing rose trees. Sasano et al. [55] used this representation for deriving programs for the maximum marking problems, but his representation, however, is irrelevant to the formalization of top-down computations or computations among siblings.

Nested data structures may be regarded as instances of rose trees. The NESL [12] provides computational patterns for nested computations, and Palmer et al. discussed how nested computations can be compiled based on this paradigm [51]. These nested computations, however, may fail in performance when the height of rose trees get large. Kakehi et al. [37] has developed a parallel implementation of rose-tree reductions, which is efficient in regardless to the height of rose trees.

Diffusion Theorems The idea of the diffusion theorem was first proposed by Hu et al. on lists [32], and was generalized to polytypic programming [31, 32]. These ideas are based on the two computational patterns on data structures, namely, the bottom-up computation and the top-down computation. These diffusion transformations were also studied by Ahn et al. and they also developed an analytical method for deriving parallel programs by utilizing the static slicing technique [2].

In [44], we proposed a more powerful diffusion theorem which is specific to binary trees. This paper generalized these even for the rose-tree specific computations, providing formalizations with dependencies among siblings.

Implementation in C++ In this paper, we have implemented our rose-tree skeletons as wrapper functions on existing binary-tree skeleton library. To achieve such implementation, we have utilized two features of C++ language, namely, template mechanism and function objects. These two mechanisms are important features of C++, and they are also used in the standard template library (STL) [35] and the new library in C++ called Boost [38]. Kuchen et al. have discussed the advantages of these features in the skeletal parallel programming from the user's point of view [40, 61]. This paper also discussed the advantages of these features from the developer's point of view.

9 Conclusion

In this paper, we introduced seven parallel rose-tree skeletons. We designed these skeletons as simple as possible based on Constructive Algorithmics, and showed a parallel implementation based on the binary-tree representation. Our rose-tree skeletons are natural extensions of the binary-tree skeletons proposed so far, and we have added two computational patterns to denote computations among siblings.

The expressiveness of our rose-tree skeletons is enhanced with the diffusion theorems. The diffusion theorems bridge the gap between the users' recursive algorithms and the skeletal parallel programs. The diffusion theorems are so powerful that we can successfully derive parallel programs for three nontrivial examples as in Section 6. We have made a prototype implementation of our parallel rose-tree skeletons as wrapper functions of existing binary-tree skeleton library. Regardless of the rapid development, the skeletons have shown good scalability.

It is our hope that our rose-tree skeletons are good foundation for not only designing but also implementing parallel tree programs. Our future work is to extend our system with more involved computational patterns on rose trees. Diks and Hagerup [23] have developed a parallel algorithm for computations in which sorting of the siblings is required. Our skeletons currently cannot represent such patterns.

References

1. K. Abrahamson, N. Dadoun, D. G. Kirkpatrik, and T. Przytycka. A simple parallel tree contraction algorithm. *Journal of Algorithms*, 10(2):287–302, June 1989.

2. J. Ahn and T. Han. An analytical method for parallelization of recursive functions. *Parallel Processing Letters*, 10(1):87–98, 2000.

3. D. A. Bader, S. Sreshta, and N. R. Weisse-Bernstein. Evaluating arithmetic expressions using tree contraction: A fast and scalable parallel implementation for symmetric multiprocessors (SMPs). In *9th International Conference on High Performance Computing (HiPC 2002)*, volume Lecture Notes in Computer Science 2552, pages 63–75, Bangalore, India, Dec 2002.

4. R. P. K. Banerjee, V. Goel, and A. Mukherjee. Efficient parallel evaluation of CSG tree using fixed number of processors. In *ACM Symposium on Solid Modeling Foundations and CAD/CAM Applications*, pages 137–146, Montreal, Canada, May 1993.

5. R. Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall, January 1998.

6. R. S. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, volume 36 of *NATO ASI Series F*, pages 5–42. Springer-Verlag, 1987.

7. R. S. Bird. Constructive functional programming. In *STOP Summer School on Constructive Algorithmics, Abeland*, 9 1989.

8. R. S. Bird. Maximum marking problem. *Journal of Functional Programming*, 11(4):411–424, 7 2001.

9. R. S. Bird and O. de Moor. *Algebras of Programming*. Prentice Hall, 1996.

10. H. Bischof and S. Gorlatch. Double-scan: Introducing and implementing a new data-parallel skeleton. In B. Monien and R. Feldmann, editors, *EuroPar 2002*, volume 2400 of *LNCS*, pages 640–647. Springer, August 2002.

11. G. E. Blelloch. Scans as primitive operations. *IEEE Transactions on Computers*, 38(11):1526–1538, November 1989.

12. G. E. Blelloch, S. Chatterjee, J. C. Hardwick, J. Sipelstein, and M. Zagha. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing*, 21(1), April 1994.

13. W. Chin, A. Takano, and Z. Hu. Parallelization via context preservation. *IEEE Computer Society International Conference on Computer Languages (ICCL'98)*, pages 153–162, May 1998.

14. W. N. Chin, J. Darlington, and Y. Guo. Parallelizing conditional recurrences. In *2nd International EuroPar Conference*, volume LNCS 1123, pages 579–586, Lyon, France, 1996. Springer.

15. W. N. Chin and Z. Hu. Towards a modular program derivation via fusion and tupling. In *The First ACM SIGPLAN Conference on Generators and Components (GCSE/SAIG 2002)*, volume Lecture Notes in Computer Science 2487, pages 140–155, Pittsburgh, PA, USA, Oct 2002. Springer Verlag.

16. M. Cole. *Algorithmic skeletons : A structured approach to the management of parallel computation*. Research Monographs in Parallel and Distributed Computing, Pitman, London, 1989.

17. M. Cole. Parallel programming, list homomorphisms and the maximum segment sum problems. Report CSR-25-93, Department of Computing Science, The University of Edinburgh, May 1993.

18. M. Cole. Parallel programming with list homomorphisms. *Parallel Processing Letters*, 5(2), 1995.

19. R. Cole and U. Vishkin. The accelerated centroid decomposition technique for optimal parallel tree evaluation in logarithmic time. *Algorithmica*, 3:329–346, 1988.

20. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, second edition, 2001.

21. F. Dehne, A. Ferreira, E. Caceres, S. W. Song, and A. Roncato. Efficient parallel graph algorithms for coarse grained multicomputers and bsp. *Algorithmica*, 33(2):183–200, 2002.

22. H. Deldari, J. R. Davy, and P. M. Dew. A skeleton for parallel CSG with a performance model. Technical report, School of computer studies research report series, University of Leeds, 1997.

23. K. Diks and T. Hagerup. More general parallel tree contraction: Register allocation and broadcasting in a tree. *Theoretical Computer Science*, 203(1):3–29, 1998.

24. K. Emoto, Z. Hu, K. Kakehi, and M. Takeichi. A compositional framework for developing parallel programs on two dimensional arrays. Technical Report METR2005-09, Department of Mathematical Informatics, University of Tokyo, 2005.

25. J. Gibbons. Computing downwards accumulations on trees quickly. In G. Gupta, G. Mohay, and R. Topor, editors, *Proceedings of 16th Australian Computer Science Conference*, volume 15 (1), pages 685–691. Australian Computer Science Communications, February 1993.

26. J. Gibbons, W. Cai, and D. B. Skillicorn. Efficient parallel algorithms for tree accumulations. *Science of Computer Programming*, 23(1):1–18, 1994.

27. S. Gorlatch. Systematic efficient parallelization of scan and other list homomorphisms. In *Annual European Conference on Parallel Processing, LNCS 1124*, pages 401–408, LIP, ENS Lyon, France, August 1996. Springer-Verlag.

28. X. He. Efficient parallel algorithms for solving some tree problems. In *24th Allerton Conference on Communication, Control and Computing*, pages 777–786, 1986.

29. Z. Hu, H. Iwasaki, and M. Takeichi. Formal derivation of efficient parallel programs by construction of list homomorphisms. *ACM Transactions on Programming Languages and Systems*, 19(3):444–461, 1997.

30. Z. Hu, H. Iwasaki, M. Takeichi, and A. Takano. Tupling calculation eliminates multiple data traversals. In *ACM SIGPLAN International Conference on Functional Programming*, pages 164–175, Amsterdam, The Netherlands, June 1997. ACM Press.

31. Z. Hu, M. Takeichi, and H. Iwasaki. Towards polytypic parallel programming. Technical Report METR 98-09, University of Tokyo, 1998.

32. Z. Hu, M. Takeichi, and H. Iwasaki. Diffusion: Calculating efficient parallel programs. In *1999 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '99)*, pages 85–94, San Antonio, Texas, January 1999. BRICS Notes Series NS-99-1.

33. H. Iwasaki and Z. Hu. A new parallel skeleton for general accumulative computations. *International Journal of Parallel Programming*, 32(5):389–414, 2004.

34. J. 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.

35. N. M. Josuttis. *The C++ Standard Library : A Tutorial and Reference.* Addison-Wesley, 1999.

36. K. Kakehi, Z. Hu, and M. Takeichi. List homomorphism with accumulation. In W. Dosch and R. Y. Lee, editors, *Proceedings of the ACIS Fourth International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD'03)*, pages 250–259, Lübeck, Germany, October 2003. ACIS.

37. K. Kakehi, K. Matsuzaki, and K. Emoto. Fast tree reductions on distributed environments. Submitted to 20th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2006), 2005.

38. B. Karlsson. *Beyond the C++ Standard Library : An Introduction to Boost.* Addison-Wesley, 2005.

39. G. Kazai, M. Lalmas, N. Fuhr, and N. Gövert. A report on the first year of the INitiative for the Evaluation of XML retrieval (INEX'02). *Journal of the American Society for Information Science and Technology*, 55(6), 2004.

40. H. Kuchen and J. Striegnitz. Higher-order functions and partial applications for a C++ skeleton library. In *Proceedings of the International Symposium on Computing in Object-oriented Parallel Environments (ISCOPE 2002)*, 2002.

41. K. Matsuzaki, K. Emoto, H. Iwasaki, and Z. Hu. A library of constructive skeletons for sequential style of parallel programming. Submitted to ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2006), 2005.

42. K. Matsuzaki, Z. Hu, K. Kakehi, and M. Takeichi. Systematic derivation of tree contraction algorithms. *Parallel Processing Letters*, 15(3):321–336, 2005.

43. K. Matsuzaki, Z. Hu, and M. Takeichi. Implementation of parallel tree skeletons on distributed systems. In *Proceedings of The Third Asian Workshop on Programming Languages And Systems*, pages 258–271, Shanghai, China, 2002.

44. K. Matsuzaki, Z. Hu, and M. Takeichi. Parallelization with tree skeletons. In *Proceedings of the 9th EuroPar Conference (EuroPar 2003), LNCS 2790*, pages 789–798, Klagenfurt, Austria, Aug 2003. Springer-Verlag.

45. E. W. Mayr and R. Werchner. Optimal routing of parentheses on the hypercube. *Journal of Parallel and Distributed Computing*, 26(2):181–192, 1995.

46. E. W. Mayr and R. Werchner. Optimal tree constraction and term matching on the hypercube and related networks. *Algorithmica*, 18(3):445–460, 1997.

47. E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Proc. Conference on Functional Programming Languages and Computer Architecture* (LNCS 523), pages 124–144, Cambridge, Massachuetts, August 1991.

48. G. L. Miller and J. H. Reif. Parallel tree contraction and its application. In *26th Annual Symposium on Foundations of Computer Science*, pages 478–489, Portland, OR, October 1985. IEEE Computer Society Press.

49. G. L. Miller and J. H. Reif. Parallel tree contraction part 2: Further applications. *SIAM Journal on Computing*, 20(6):1128–1147, December 1991.

50. G. L. Miller and S.-H. Teng. Tree-based parallel algorithm design. *Algorithmica*, 19(4):369–389, 1997.

51. D. Palmer, J. Prins, S. Chatterjee, and R. Faith. Piecewise execution of nested data-parallel programs. In *Languages and compilers for parallel computing : 8th International Workshop, Columbus, OH, USA, August 10-12 1995 Proceedings*, volume 1033 of *Lecture Notes in Computer Science*, pages 346–361. Spring-Verlog, 1996.

52. S. Peyton Jones and J. Hughes. Report on the programming language haskell 98: A non-strict, purely functional language. Available from `http://www.haskell.org/`, February 1999.

53. F. Rabhi and S. Gorlatch. *Patterns and Skeletons for Parallel and Distributed Computing.* Springer-Verlag New York Inc., 2002.

54. M. Reid-Miller, G. L. Miller, and F. Modugno. List ranking and parallel tree contraction. In J. H. Reif, editor, *Synthesis of Parallel Algorithms*, chapter 3, pages 115–194. Morgan Kaufmann Publishers, 1996.

55. I. Sasano, Z. Hu, M. Takeichi, and M. Ogawa. Make it practical: A generic linear time algorithm for solving maximum-weightsum problems. In *The 2000 ACM SIGPLAN International Conference on Functional Programming (ICFP 2000)*, pages 137–149, Montreal, Canada, September 2000. ACM Press.

56. D. B. Skillicorn. The Bird-Meertens formalism as a parallel model. In J. S. Kowalik and L. Grandinetti, editors, *NATO ASI Workshop on Software for Parallel Computation, NATO ARW "Software for Parallel Computation"*, volume 106 of *F*, Cetraro, Italy, June 1992. Springer-Verlag NATO ASI.

57. D. B. Skillicorn. The bird-meertens formalism as a parallel model. In J. S. Kowalik and L. Grandinetti, editors, *Software for Parallel Computation*, volume 106 of *NATO ASI Series F*, pages 120–133. Springer-Verlag, 1993.

58. D. B. Skillicorn. *Foundations of Parallel Programming.* Cambridge University Press, 1994.

59. D. B. Skillicorn. Parallel implementation of tree skeletons. *Journal of Parallel and Distributed Computing*, 39(2):115–125, 1996.

60. D. B. Skillicorn. Structured parallel computation in structured documents. *Journal of Universal Computer Science*, 3(1):42–68, 1997.

61. J. Striegnitz. Making C++ ready for algorithmic skeletons. Technical Report FZJ-ZAM-IB-2000-08, ZAM, Forschungszentrum, Jülich, 2000.

A Proofs of the diffusion theorems

A.1 Proof of Theorem 2

First, we define the specification as mutual recursive functions.

$$
\begin{aligned}
&h\ (RNode\ a\ ts) = k\ a \oplus h'\ ts \\
&h'\ [\,] \qquad\qquad = \iota_{\oplus} \\
&h'\ (t : ts) \qquad = h\ t \otimes h'\ ts
\end{aligned}
$$

Then we calculate the specification and skeletal program for the top case of the rose trees.

$$
\begin{aligned}
Spec &= h\ (RNode\ a\ ts) \\
&= k\ a \oplus h'\ ts \\
\\
Skel &= ((\mathsf{reduce}_r\ (\oplus)\ (\otimes)) \circ (\mathsf{map}_r\ k))\ (RNode\ a\ ts) \\
&= (\mathsf{reduce}_r\ (\oplus)\ (\otimes))\ (RNode\ (k\ a)\ (\mathsf{map}'_r\ k\ ts)) \\
&= k\ a \oplus (\mathsf{reduce}'_r\ (\oplus)\ (\otimes)\ (\mathsf{map}'_r\ k\ ts))
\end{aligned}
$$

Thus we prove the following equation by induction on the structure of rose trees.

$$h'\ ts = \mathsf{reduce}'_r\ (\oplus)\ (\otimes)\ (\mathsf{map}'_r\ k\ ts)$$

Base case: $ts \Rightarrow [\,]$

$$
\begin{aligned}
LHS &= h'\ [\,] = \iota_{\oplus} \\
RHS &= \mathsf{reduce}'_r\ (\oplus)\ (\otimes)\ (\mathsf{map}'_r\ k\ [\,]) \\
&= \mathsf{reduce}'_r\ (\oplus)\ (\otimes)\ [\,] \\
&= \iota_{\otimes}
\end{aligned}
$$

Inductive case: $ts \Rightarrow (RNode\ a\ ts) : ss$

$$
\begin{aligned}
LHS &= h'\ ((RNode\ a\ ts) : ss) \\
&= h\ (RNode\ a\ ts) \otimes (h'\ ss) \\
&= (k\ a \oplus h'\ ts) \otimes (h'\ ss) \\
RHS &= \mathsf{reduce}'_r\ (\oplus)\ (\otimes)\ (\mathsf{map}'_r\ k\ ((RNode\ a\ ts) : ss)) \\
&= \mathsf{reduce}'_r\ (\oplus)\ (\otimes)\ ((RNode\ (k\ a)\ (\mathsf{map}'_r\ k\ ts)) : (\mathsf{map}'_r\ k\ ss)) \\
&= (k\ a \oplus (\mathsf{reduce}'_r\ (\oplus)\ (\otimes)\ (\mathsf{map}'_r\ k\ ts))) \otimes (\mathsf{reduce}'_r\ (\oplus)\ (\otimes)\ (\mathsf{map}'_r\ k\ ss)) \\
&= (k\ a \oplus h'\ ts) \otimes (h'\ ss)
\end{aligned}
$$

It follows from the induction above that the theorem holds.

A.2 Proof of Theorem 3

First, we define the specification as mutual recursive functions.

$$
\begin{aligned}
&f\ c\ (RNode\ a\ ts) = RNode\ (k\ a\ c)\ (f'\ (c \oplus g\ a)\ ts) \\
&f'\ c\ [\,] \qquad\qquad = [\,] \\
&f'\ c\ (t : ts) \qquad = f\ c\ t : f'\ c\ ts
\end{aligned}
$$

Then we calculate the specification and skeletal program for the top case of the rose trees.

$$
\begin{aligned}
Spec &= f\ c\ (RNode\ a\ ts) \\
&= RNode\ (k\ a\ c)\ (f'\ (c \oplus g\ a)\ ts) \\
Skel &= \mathsf{zipwith}_r\ (\lambda a\ d.k\ a\ (c \oplus d))\ (RNode\ a\ ts)\ (\mathsf{dAcc}_r\ (\oplus)\ (\mathsf{map}_r\ g\ (RNode\ a\ ts))) \\
&= \mathsf{zipwith}_r\ (\lambda a\ d.k\ a\ (c \oplus d))\ (RNode\ a\ ts)\ (\mathsf{dAcc}_r\ (\oplus)\ (RNode\ (g\ a)\ (\mathsf{map}'_r\ g\ ts))) \\
&= \mathsf{zipwith}_r\ (\lambda a\ d.k\ a\ (c \oplus d))\ (RNode\ a\ ts)\ (RNode\ \iota_{\oplus}\ (\mathsf{dAcc}''_r\ (\oplus)\ (g\ a)\ (\mathsf{map}'_r\ g\ ts))) \\
&= RNode\ (k\ a\ (c \oplus \iota_{\oplus}))\ (\mathsf{zipwith}'_r\ (\lambda a\ d.k\ a\ (c \oplus d))\ ts\ (\mathsf{dAcc}''_r\ (\oplus)\ (g\ a)\ (\mathsf{map}'_r\ g\ ts))) \\
&= RNode\ (k\ a\ c)\ (\mathsf{zipwith}'_r\ (\lambda a\ d.k\ a\ (c \oplus d))\ ts\ (\mathsf{dAcc}''_r\ (\oplus)\ (g\ a)\ (\mathsf{map}'_r\ g\ ts)))
\end{aligned}
$$

Thus we prove the following equation by induction on the structure of rose trees.

$$f'\ (c \oplus v)\ ts = \mathsf{zipwith}'_r\ (\lambda a\ d.k\ a\ (c \oplus d))\ ts\ (\mathsf{dAcc}''_r\ (\oplus)\ v\ (\mathsf{map}'_r\ g\ ts))$$

Base case: $ts \Rightarrow [\,]$

$$
\begin{aligned}
LHS &= f'\ (c \oplus v)\ [\,]\ = [\,] \\
RHS &= \mathsf{zipwith}'_r\ (\lambda a\ d.k\ a\ (c \oplus d))\ [\,]\ (\mathsf{dAcc}''_r\ (\oplus)\ v\ (\mathsf{map}'_r\ g\ [\,])) \\
&= \mathsf{zipwith}'_r\ (\lambda a\ d.k\ a\ (c \oplus d))\ [\,]\ (\mathsf{dAcc}''_r\ (\oplus)\ v\ [\,]) \\
&= \mathsf{zipwith}'_r\ (\lambda a\ d.k\ a\ (c \oplus d))\ [\,]\ [\,] \\
&= [\,]
\end{aligned}
$$

Inductive case: $ts \Rightarrow (RNode\ a\ ts) : ss$

$$
\begin{aligned}
LHS &= f'\ (c \oplus v)\ ((RNode\ a\ ts) : ss) \\
&= (RNode\ (k\ a\ (c \oplus v))\ (f'\ (c \oplus v \oplus g\ a)\ ts)) : f'\ (c \oplus v)\ ss \\
RHS &= \mathsf{zipwith}'_r\ (\lambda a\ d.k\ a\ (c \oplus d))\ ((RNode\ a\ ts) : ss) \\
&\qquad (\mathsf{dAcc}''_r\ (\oplus)\ v\ (\mathsf{map}'_r\ g\ ((RNode\ a\ ts) : ss))) \\
&= \mathsf{zipwith}'_r\ (\lambda a\ d.k\ a\ (c \oplus d))\ ((RNode\ a\ ts) : ss) \\
&\qquad (\mathsf{dAcc}''_r\ (\oplus)\ v\ ((RNode\ (g\ a)\ (\mathsf{map}'_r\ g\ ts)) : \mathsf{map}'_r\ g\ ss)) \\
&= \mathsf{zipwith}'_r\ (\lambda a\ d.k\ a\ (c \oplus d))\ ((RNode\ a\ ts) : ss) \\
&\qquad (RNode\ v\ (\mathsf{dAcc}''_r\ (\oplus)\ (v \oplus g\ a)\ (\mathsf{map}'_r\ g\ ts))) : \mathsf{dAcc}''_r\ (\oplus)\ v\ (\mathsf{map}'_r\ g\ ss) \\
&= (RNode\ (k\ a\ (c \oplus v))\ (\mathsf{zipwith}'_r\ (\lambda a\ d.k\ a\ (c \oplus d))\ ts\ (\mathsf{dAcc}''_r\ (\oplus)\ (v \oplus g\ a)\ (\mathsf{map}'_r\ g\ ts)))) \\
&\quad : (\mathsf{zipwith}'_r\ (\lambda a\ d.k\ a\ (c \oplus d))\ ss\ (\mathsf{dAcc}''_r\ (\oplus)\ v\ (\mathsf{map}'_r\ g\ ss))) \\
&= (RNode\ (k\ a\ (c \oplus v))\ (f'\ (c \oplus v \oplus g\ a)\ ts)) : f'\ (c \oplus v)\ ss
\end{aligned}
$$

It follows from the induction above that the theorem holds.

A.3 Proof of Theorem 4

First we define the specification as mutual recursive functions.

$$
\begin{aligned}
f\ c\ (RNode\ a\ ts) &= RNode\ (k\ a\ c)\ (f'\ c\ a\ ts) \\
f'\ c\ a\ [\,] &= [\,] \\
f'\ c\ a\ (t : ts) &= f\ (c \oplus g\ a\ (root\ t))\ t : f'\ c\ a\ ts
\end{aligned}
$$

Then we calculate the specification and skeletal program for the top case of the rose trees. Here, we use the sequential definition using $(>>)$.

$$
\begin{aligned}
Spec &= RNode\ (k\ a\ c)\ (f'\ c\ a\ ts) \\
Skel &= \mathbf{let}\ gt = setroot\ \iota_\oplus\ (\mathsf{zipwith}_r\ g\ (\mathsf{dAcc}_r\ (\gg))\ (RNode\ a\ ts))\ (RNode\ a\ ts)) \\
&\qquad dt = \mathsf{dAcc}_r\ (\oplus)\ gt \\
&\quad \mathbf{in}\ \ \mathsf{zipwith3}_r\ (\lambda a\ d\ d'.k\ a\ (c \oplus d \oplus d'))\ (RNode\ a\ ts)\ dt\ gt \\
&= \mathbf{let}\ gt = setroot\ \iota_\oplus\ (\mathsf{zipwith}_r\ g\ (RNode\ \_\ (\mathsf{dAcc}''_r\ (\gg)\ a\ ts))\ (RNode\ a\ ts)) \\
&\qquad dt = \mathsf{dAcc}_r\ (\oplus)\ gt \\
&\quad \mathbf{in}\ \ \mathsf{zipwith3}_r\ (\lambda a\ d\ d'.k\ a\ (c \oplus d \oplus d'))\ (RNode\ a\ ts)\ dt\ gt \\
&= \mathbf{let}\ gt = RNode\ \iota_\oplus\ (\mathsf{zipwith}'_r\ g\ (\mathsf{dAcc}''_r\ (\gg)\ a\ ts)\ ts) \\
&\qquad dt = \mathsf{dAcc}_r\ (\oplus)\ gt \\
&\quad \mathbf{in}\ \ \mathsf{zipwith3}_r\ (\lambda a\ d\ d'.k\ a\ (c \oplus d \oplus d'))\ (RNode\ a\ ts)\ dt\ gt \\
&= \mathbf{let}\ dt = \mathsf{dAcc}_r\ (\oplus)\ (RNode\ \iota_\oplus\ (\mathsf{zipwith}'_r\ g\ (\mathsf{dAcc}''_r\ (\gg)\ a\ ts)\ ts)) \\
&\quad \mathbf{in}\ \ \mathsf{zipwith3}_r\ (\lambda a\ d\ d'.k\ a\ (c \oplus d \oplus d'))\ (RNode\ a\ ts)\ dt \\
&\qquad\qquad (RNode\ \iota_\oplus\ (\mathsf{zipwith}'_r\ g\ (\mathsf{dAcc}''_r\ (\gg)\ a\ ts)\ ts)) \\
&= \mathsf{zipwith3}_r\ (\lambda a\ d\ d'.k\ a\ (c \oplus d \oplus d'))\ (RNode\ a\ ts) \\
&\qquad\qquad (RNode\ \iota_\oplus\ (\mathsf{dAcc}''_r\ (\oplus)\ \iota_\oplus\ (\mathsf{zipwith}'_r\ g\ (\mathsf{dAcc}''_r\ (\gg)\ a\ ts)\ ts))) \\
&\qquad\qquad (RNode\ \iota_\oplus\ (\mathsf{zipwith}'_r\ g\ (\mathsf{dAcc}''_r\ (\gg)\ a\ ts)\ ts)) \\
&= RNode\ (k\ a\ c)\ (\mathsf{zipwith3}'_r\ (\lambda a\ d\ d'.k\ a\ (c \oplus d \oplus d'))\ ts \\
&\qquad\qquad (\mathsf{dAcc}''_r\ (\oplus)\ \iota_\oplus\ (\mathsf{zipwith}'_r\ g\ (\mathsf{dAcc}''_r\ (\gg)\ a\ ts)\ ts)) \\
&\qquad\qquad (\mathsf{zipwith}'_r\ g\ (\mathsf{dAcc}''_r\ (\gg)\ a\ ts)\ ts)) \\
&= \mathbf{let}\ gt = \mathsf{zipwith}'_r\ g\ (\mathsf{dAcc}''_r\ (\gg)\ a\ ts)\ ts \\
&\qquad dt = \mathsf{dAcc}''_r\ (\oplus)\ \iota_\oplus\ gt \\
&\quad \mathbf{in}\ RNode\ (k\ a\ c)\ (\mathsf{zipwith3}'_r\ (\lambda a\ d\ d'.k\ a\ (c \oplus d \oplus d'))\ ts\ dt\ gt)
\end{aligned}
$$

Thus we prove the following equation by induction on the structure of rose trees.

$$
\begin{aligned}
f'\ (c \oplus c')\ a\ ts = \textbf{let}\ & gt = \mathsf{zipwith}'_r\ g\ (\mathsf{dAcc}''_r\ (\gg)\ a\ ts)\ ts \\
& dt = \mathsf{dAcc}''_r\ (\oplus)\ c'\ gt \\
\textbf{in}\ & \mathsf{zipwith3}'_r\ (\lambda a\ d\ d'.k\ a\ (c \oplus d \oplus d'))\ ts\ dt\ gt
\end{aligned}
$$

Base case: $ts \Rightarrow [\,]$

$$
\begin{aligned}
LHS &= f'\ (c \oplus c')\ a\ [\,] \\
&= [\,] \\
RHS &= \textbf{let}\ gt = \mathsf{zipwith}'_r\ g\ (\mathsf{dAcc}''_r\ (\gg)\ a\ [\,])\ [\,] \\
&\qquad dt = \mathsf{dAcc}''_r\ (\oplus)\ c'\ gt \\
&\quad \textbf{in}\ \mathsf{zipwith3}'_r\ (\lambda a\ d\ d'.k\ a\ (c \oplus d \oplus d'))\ [\,]\ dt\ gt \\
&= \textbf{let}\ dt = \mathsf{dAcc}''_r\ (\oplus)\ c'\ [\,] \\
&\quad \textbf{in}\ \mathsf{zipwith3}'_r\ (\lambda a\ d\ d'.k\ a\ (c \oplus d \oplus d'))\ [\,]\ dt\ [\,] \\
&= \mathsf{zipwith3}'_r\ (\lambda a\ d\ d'.k\ a\ (c \oplus d \oplus d'))\ [\,]\ [\,]\ [\,] \\
&= [\,]
\end{aligned}
$$

Inductive case: $ts \Rightarrow (RNode\ a'\ ts) : ss$

$$
\begin{aligned}
LHS &= f'\ (c \oplus c')\ a\ ((RNode\ a'\ ts) : ss) \\
&= f\ ((c \oplus c') \oplus g\ a\ (root\ (RNode\ a'\ ts)))\ (RNode\ a'\ ts) : f'\ (c \oplus c')\ a\ ss \\
&= f\ (c \oplus c' \oplus g\ a\ a')\ (RNode\ a'\ ts) : f'\ (c \oplus c')\ a\ ss \\
&= RNode\ (k\ a'\ (c \oplus c' \oplus g\ a\ a'))\ (f'\ (c \oplus c' \oplus g\ a\ a')\ a'\ ts) : f'\ (c \oplus c')\ a\ ss \\
RHS &= \textbf{let}\ gt\ = \mathsf{zipwith}'_r\ g\ (\mathsf{dAcc}''_r\ (\gg)\ a\ ((RNode\ a'\ ts) : ss))\ ((RNode\ a'\ ts) : ss) \\
&\qquad dt\ = \mathsf{dAcc}''_r\ (\oplus)\ c'\ gt \\
&\quad \textbf{in}\ \mathsf{zipwith3}'_r\ (\lambda a\ d\ d'.k\ a\ (c \oplus d \oplus d'))\ ((RNode\ a'\ ts) : ss)\ dt\ gt \\
&= \textbf{let}\ gt\ = (RNode\ (g\ a\ a')\ (\mathsf{zipwith}'_r\ g\ (\mathsf{dAcc}''_r\ (\gg)\ a'\ ts)\ ts)) : \\
&\qquad\qquad (\mathsf{zipwith}'_r\ g\ (\mathsf{dAcc}''_r\ (\gg)\ a\ ss)\ ss) \\
&\qquad dt\ = \mathsf{dAcc}''_r\ (\oplus)\ c'\ gt \\
&\quad \textbf{in}\ \mathsf{zipwith3}'_r\ (\lambda a\ d\ d'.k\ a\ (c \oplus d \oplus d'))\ ((RNode\ a'\ ts) : ss)\ dt\ gt \\
&= \textbf{let}\ gt'\ = \mathsf{zipwith}'_r\ g\ (\mathsf{dAcc}''_r\ (\gg)\ a'\ ts)\ ts \\
&\qquad gt''\ = \mathsf{zipwith}'_r\ g\ (\mathsf{dAcc}''_r\ (\gg)\ a\ ss)\ ss \\
&\qquad dt\ = \mathsf{dAcc}''_r\ (\oplus)\ c'\ ((RNode\ (g\ a\ a')\ gt') : gt'') \\
&\quad \textbf{in}\ \mathsf{zipwith3}'_r\ (\lambda a\ d\ d'.k\ a\ (c \oplus d \oplus d'))\ ((RNode\ a'\ ts) : ss)\ dt \\
&\qquad\qquad ((RNode\ (g\ a\ a')\ gt') : gt'') \\
&= \textbf{let}\ gt'\ = \mathsf{zipwith}'_r\ g\ (\mathsf{dAcc}''_r\ (\gg)\ a'\ ts)\ ts \\
&\qquad gt''\ = \mathsf{zipwith}'_r\ g\ (\mathsf{dAcc}''_r\ (\gg)\ a\ ss)\ ss \\
&\qquad dt\ = (RNode\ c'\ (\mathsf{dAcc}''_r\ (\oplus)\ (c' \oplus g\ a\ a')\ gt')) : (\mathsf{dAcc}''_r\ (\oplus)\ c'\ gt'') \\
&\quad \textbf{in}\ \mathsf{zipwith3}'_r\ (\lambda a\ d\ d'.k\ a\ (c \oplus d \oplus d'))\ ((RNode\ a'\ ts) : ss)\ dt \\
&\qquad\qquad ((RNode\ (g\ a\ a')\ gt') : gt'') \\
&= \textbf{let}\ gt'\ = \mathsf{zipwith}'_r\ g\ (\mathsf{dAcc}''_r\ (\gg)\ a'\ ts)\ ts \\
&\qquad gt''\ = \mathsf{zipwith}'_r\ g\ (\mathsf{dAcc}''_r\ (\gg)\ a\ ss)\ ss \\
&\qquad dt'\ = \mathsf{dAcc}''_r\ (\oplus)\ (c' \oplus g\ a\ a')\ gt' \\
&\qquad dt''\ = \mathsf{dAcc}''_r\ (\oplus)\ c'\ gt'' \\
&\quad \textbf{in}\ \mathsf{zipwith3}'_r\ (\lambda a\ d\ d'.k\ a\ (c \oplus d \oplus d'))\ ((RNode\ a'\ ts) : ss)\ ((RNode\ c'\ dt') : dt'') \\
&\qquad\qquad ((RNode\ (g\ a\ a')\ gt') : gt'') \\
&= \textbf{let}\ gt'\ = \mathsf{zipwith}'_r\ g\ (\mathsf{dAcc}''_r\ (\gg)\ a'\ ts)\ ts \\
&\qquad gt''\ = \mathsf{zipwith}'_r\ g\ (\mathsf{dAcc}''_r\ (\gg)\ a\ ss)\ ss \\
&\qquad dt'\ = \mathsf{dAcc}''_r\ (\oplus)\ (c' \oplus g\ a\ a')\ gt' \\
&\qquad dt''\ = \mathsf{dAcc}''_r\ (\oplus)\ c'\ gt'' \\
&\quad \textbf{in}\ RNode\ (k\ a'\ (c \oplus c' \oplus g\ a\ a'))\ (\mathsf{zipwith3}'_r\ (\lambda a\ d\ d'.k\ a\ (c \oplus d \oplus d'))\ ts\ dt'\ gt') : \\
&\qquad (\mathsf{zipwith3}'_r\ (\lambda a\ d\ d'.k\ a\ (c \oplus d \oplus d'))\ ss\ dt''\ gt'') \\
&= \textbf{let}\ gt'\ = \mathsf{zipwith}'_r\ g\ (\mathsf{dAcc}''_r\ (\gg)\ a'\ ts)\ ts \\
&\qquad gt''\ = \mathsf{zipwith}'_r\ g\ (\mathsf{dAcc}''_r\ (\gg)\ a\ ss)\ ss \\
&\qquad dt'\ = \mathsf{dAcc}''_r\ (\oplus)\ (c' \oplus g\ a\ a')\ gt' \\
&\qquad dt''\ = \mathsf{dAcc}''_r\ (\oplus)\ c'\ gt'' \\
&\quad \textbf{in}\ RNode\ (k\ a'\ (c \oplus c' \oplus g\ a\ a'))\ (\mathsf{zipwith3}'_r\ (\lambda a\ d\ d'.k\ a\ (c \oplus d \oplus d'))\ ts\ dt'\ gt') : \\
&\qquad (\mathsf{zipwith3}'_r\ (\lambda a\ d\ d'.k\ a\ (c \oplus d \oplus d'))\ ss\ dt''\ gt'') \\
&= RNode\ (k\ a'\ (c \oplus c' \oplus g\ a\ a'))\ (f'\ (c \oplus c' \oplus g\ a\ a')\ a'\ ts)\ (f'\ (c \oplus c')\ a\ ss)
\end{aligned}
$$

It follows from the calculation above that the theorem holds.

A.4 Proof of Theorem 5

First, we define the specification as mutual recursive functions.

$$
\begin{aligned}
&f\ c\ (RNode\ a\ ts) = k\ a\ c \oplus f'\ (c \odot g\ a)\ ts \\
&f'\ c\ [\,] \qquad\qquad = \iota_{\otimes} \\
&f'\ c\ (t : ts) \qquad = f\ c\ t \otimes f'\ c\ ts
\end{aligned}
$$

Then we calculate the specification and skeletal program for the top case of the rose trees.

$$
\begin{aligned}
Spec &= k\ a\ c \oplus f'\ (c \odot g\ a)\ ts \\
Skel &= \mathbf{let}\ dt = \mathsf{dAcc}_r\ (\odot)\ (\mathsf{map}_r\ g\ (RNode\ a\ ts)) \\
&\quad\ \mathbf{in}\ \ \mathsf{reduce}_r\ (\oplus)\ (\otimes)\ (\mathsf{zipwith}_r\ k\ (RNode\ a\ ts)\ (\mathsf{map}_r\ (c\odot)\ dt)) \\
&= \mathbf{let}\ dt = RNode\ \iota_{\odot}\ (\mathsf{dAcc}''_r\ (\odot)\ (g\ a)\ (\mathsf{map}'_r\ g\ ts)) \\
&\quad\ \mathbf{in}\ \ \mathsf{reduce}_r\ (\oplus)\ (\otimes)\ (\mathsf{zipwith}_r\ k\ (RNode\ a\ ts)\ (\mathsf{map}_r\ (c\odot)\ dt)) \\
&= \mathbf{let}\ dt' = \mathsf{dAcc}''_r\ (\odot)\ (g\ a)\ (\mathsf{map}'_r\ g\ ts) \\
&\quad\ \mathbf{in}\ \ \mathsf{reduce}_r\ (\oplus)\ (\otimes)\ (\mathsf{zipwith}_r\ k\ (RNode\ a\ ts)\ (RNode\ c\ (\mathsf{map}'_r\ (c\odot)\ dt'))) \\
&= \mathbf{let}\ dt' = \mathsf{dAcc}''_r\ (\odot)\ (g\ a)\ (\mathsf{map}'_r\ g\ ts) \\
&\quad\ \mathbf{in}\ \ \mathsf{reduce}_r\ (\oplus)\ (\otimes)\ (RNode\ (k\ a\ c)\ (\mathsf{zipwith}'_r\ ts\ (\mathsf{map}'_r\ (c\odot)\ dt'))) \\
&= \mathbf{let}\ dt' = \mathsf{dAcc}''_r\ (\odot)\ (g\ a)\ (\mathsf{map}'_r\ g\ ts) \\
&\quad\ \mathbf{in}\ \ (k\ a\ c) \oplus (\mathsf{reduce}'_r\ (\oplus)\ (\otimes)\ (\mathsf{zipwith}'_r\ ts\ (\mathsf{map}'_r\ (c\odot)\ dt')))
\end{aligned}
$$

Thus we prove the following equation by induction on the structure of rose trees.

$$
\begin{aligned}
f'\ (c \odot v)\ ts = \mathbf{let}\ &dt = \mathsf{dAcc}''_r\ (\odot)\ v\ (\mathsf{map}'_r\ g\ ts) \\
\mathbf{in}\ \ &\mathsf{reduce}'_r\ (\oplus)\ (\otimes)\ (\mathsf{zipwith}'_r\ k\ ts\ (\mathsf{map}'_r\ (c\odot)\ dt))
\end{aligned}
$$

Base case: $ts \Rightarrow [\,]$

$$
\begin{aligned}
LHS &= f'\ (c \odot v)\ [\,] \\
&= \iota_{\otimes} \\
RHS &= \mathbf{let}\ dt = \mathsf{dAcc}''_r\ (\odot)\ v\ (\mathsf{map}'_r\ g\ [\,]) \\
&\quad\ \mathbf{in}\ \ \mathsf{reduce}'_r\ (\oplus)\ (\otimes)\ (\mathsf{zipwith}'_r\ k\ [\,]\ (\mathsf{map}'_r\ (c\odot)\ dt)) \\
&= \mathsf{reduce}'_r\ (\oplus)\ (\otimes)\ (\mathsf{zipwith}'_r\ k\ [\,]\ (\mathsf{map}'_r\ (c\odot)\ [\,])) \\
&= \mathsf{reduce}'_r\ (\oplus)\ (\otimes)\ [\,] \\
&= \iota_{\otimes}
\end{aligned}
$$

Inductive case: $ts \Rightarrow (RNode\ a\ ts) : ss$

$$
\begin{aligned}
LHS &= f'\ (c \odot v)\ ((RNode\ a\ ts) : ss) \\
&= (k\ a\ (c \odot v) \oplus (f'\ (c \odot v \odot g\ a)\ ts)) \otimes (f'\ (c \odot v)\ ss) \\
RHS &= \mathbf{let}\ dt = \mathsf{dAcc}''_r\ (\odot)\ v\ (\mathsf{map}'_r\ g\ ((RNode\ a\ ts) : ss)) \\
&\quad\ \mathbf{in}\ \ \mathsf{reduce}'_r\ (\oplus)\ (\otimes)\ (\mathsf{zipwith}'_r\ k\ ((RNode\ a\ ts) : ss)\ (\mathsf{map}'_r\ (c\odot)\ dt)) \\
&= \mathbf{let}\ dt = RNode\ v\ (\mathsf{dAcc}''_r\ (\odot)\ (v \odot g\ a)\ (\mathsf{map}'_r\ g\ ts)) : \mathsf{dAcc}''_r\ (\odot)\ v\ (\mathsf{map}'_r\ g\ ss) \\
&\quad\ \mathbf{in}\ \ \mathsf{reduce}'_r\ (\oplus)\ (\otimes)\ (\mathsf{zipwith}'_r\ k\ ((RNode\ a\ ts) : ss)\ (\mathsf{map}'_r\ (c\odot)\ dt)) \\
&= \mathbf{let}\ dt' = \mathsf{dAcc}''_r\ (\odot)\ (v \odot g\ a)\ (\mathsf{map}'_r\ g\ ts) \\
&\qquad dt'' = \mathsf{dAcc}''_r\ (\odot)\ v\ (\mathsf{map}'_r\ g\ ss) \\
&\quad\ \mathbf{in}\ \ \mathsf{reduce}'_r\ (\oplus)\ (\otimes)\ (\mathsf{zipwith}'_r\ k\ ((RNode\ a\ ts) : ss) \\
&\qquad\qquad\qquad\qquad ((RNode\ (c \odot v)\ (\mathsf{map}'_r\ (c\odot)\ dt')) : (\mathsf{map}'_r\ (c\odot)\ dt'))) \\
&= \mathbf{let}\ dt' = \mathsf{dAcc}''_r\ (\odot)\ (v \odot g\ a)\ (\mathsf{map}'_r\ g\ ts) \\
&\qquad dt'' = \mathsf{dAcc}''_r\ (\odot)\ v\ (\mathsf{map}'_r\ g\ ss) \\
&\quad\ \mathbf{in}\ \ \mathsf{reduce}'_r\ (\oplus)\ (\otimes)\ ((RNode\ (k\ a\ (c \odot v))\ (\mathsf{zipwith}'_r\ k\ ts\ (\mathsf{map}'_r\ (c\odot)\ dt'))) : \\
&\qquad\qquad\qquad\qquad (\mathsf{zipwith}'_r\ k\ ss\ (\mathsf{map}'_r\ (c\odot)\ dt'))) \\
&= \mathbf{let}\ dt' = \mathsf{dAcc}''_r\ (\odot)\ (v \odot g\ a)\ (\mathsf{map}'_r\ g\ ts) \\
&\qquad dt'' = \mathsf{dAcc}''_r\ (\odot)\ v\ (\mathsf{map}'_r\ g\ ss) \\
&\quad\ \mathbf{in}\ \ ((k\ a\ (c \odot v)) \oplus (\mathsf{reduce}'_r\ (\oplus)\ (\otimes)\ (\mathsf{zipwith}'_r\ k\ ts\ (\mathsf{map}'_r\ (c\odot)\ dt')))) \otimes \\
&\qquad (\mathsf{reduce}'_r\ (\oplus)\ (\otimes)\ (\mathsf{zipwith}'_r\ k\ ss\ (\mathsf{map}'_r\ (c\odot)\ dt''))) \\
&= (k\ a\ (c \odot v) \oplus (f'\ (c \odot v \odot g\ a)\ ts)) \otimes (f'\ (c \odot v)\ ss)
\end{aligned}
$$

It follows from the induction above that the theorem holds.

A.5 Proof of Theorem 6

First, we define the specification as mutual recursive functions.

$$
\begin{aligned}
&f\ c\ (RNode\ a\ ts) = RNode\ (k\ a\ c)\ (f'\ (c \odot g\ a\ (h\ (RNode\ a\ ts)))\ ts)\\
&f'\ c\ [\,] = [\,]\\
&f'\ c\ (t : ts) = f\ c\ t : f'\ c\ ts
\end{aligned}
$$

Then we calculate the specification and skeletal program for the top case of the rose trees.

$$
\begin{aligned}
Spec &= RNode\ (k\ a\ c)\ (f'\ (c \odot g\ a\ (h\ (RNode\ a\ ts)))\ ts)\\
Skel &= \mathbf{let}\ t' \ = \mathsf{uAcc}_r\ (\oplus')\ (\otimes')\ (\mathsf{map}_r\ k'\ (RNode\ a\ ts))\\
&\qquad\ \ dt\ = \mathsf{dAcc}_r\ (\odot)\ (\mathsf{zipwith}_r\ g\ (RNode\ a\ ts)\ t')\\
&\quad\ \mathbf{in}\ \ \mathsf{zipwith}_r\ (\lambda a\ d.k\ a\ (c \odot d))\ (RNode\ a\ ts)\ dt\\
&= \mathbf{let}\ t' \ = RNode\ (h\ (RNode\ a\ ts))\ (\mathsf{uAcc}'_r\ (\oplus')\ (\otimes')\ (\mathsf{map}'_r\ k'\ ts))\\
&\qquad\ \ dt\ = \mathsf{dAcc}_r\ (\odot)\ (\mathsf{zipwith}_r\ g\ (RNode\ a\ ts)\ t')\\
&\quad\ \mathbf{in}\ \ \mathsf{zipwith}_r\ (\lambda a\ d.k\ a\ (c \odot d))\ (RNode\ a\ ts)\ dt\\
&= \mathbf{let}\ dt\ = \mathsf{dAcc}_r\ (\odot)\ (RNode\ (g\ a\ (h\ (RNode\ a\ ts)))\\
&\qquad\qquad\qquad\qquad (\mathsf{zipwith}'_r\ g\ ts\ (\mathsf{uAcc}'_r\ (\oplus')\ (\otimes')\ (\mathsf{map}'_r\ k'\ ts))))\\
&\quad\ \mathbf{in}\ \ \mathsf{zipwith}_r\ (\lambda a\ d.k\ a\ (c \odot d))\ (RNode\ a\ ts)\ dt\\
&= \mathbf{let}\ dt\ = RNode\ \iota_{\odot}\ (\mathsf{dAcc}''_r\ (\odot)\ (g\ a\ (h\ (RNode\ a\ ts)))\\
&\qquad\qquad\qquad\qquad (\mathsf{zipwith}'_r\ g\ ts\ (\mathsf{uAcc}'_r\ (\oplus')\ (\otimes')\ (\mathsf{map}'_r\ k'\ ts))))\\
&\quad\ \mathbf{in}\ \ \mathsf{zipwith}_r\ (\lambda a\ d.k\ a\ (c \odot d))\ (RNode\ a\ ts)\ dt\\
&= \mathbf{let}\ dt' = \mathsf{dAcc}''_r\ (\odot)\ (g\ a\ (h\ (RNode\ a\ ts)))\ (\mathsf{zipwith}'_r\ g\ ts\ (\mathsf{uAcc}'_r\ (\oplus')\ (\otimes')\ (\mathsf{map}'_r\ k'\ ts)))\\
&\quad\ \mathbf{in}\ \ RNode\ (k\ a\ c)\ (\mathsf{zipwith}'_r\ (\lambda a\ d.k\ a\ (c \odot d))\ ts\ dt')
\end{aligned}
$$

In the calculation we used the following equation which can be proved immediately by definition.

$$
\begin{aligned}
h\ (RNode\ a\ ts) &= \mathsf{reduce}_r\ (\oplus')\ (\otimes')\ (\mathsf{map}_r\ k'\ (RNode\ a\ ts))\\
&= root\ (\mathsf{uAcc}_r\ (\oplus')\ (\otimes')\ (\mathsf{map}_r\ k'\ (RNode\ a\ ts)))
\end{aligned}
$$

Thus we prove the following equation by induction on the structure of rose trees.

$$
\begin{aligned}
f'\ (c \odot v)\ ts = \mathbf{let}\ &dt = \mathsf{dAcc}''_r\ (\odot)\ v\ (\mathsf{zipwith}'_r\ g\ ts\ (\mathsf{uAcc}'_r\ (\oplus')\ (\otimes')\ (\mathsf{map}'_r\ k'\ ts)))\\
\mathbf{in}\ &\mathsf{zipwith}'_r\ (\lambda a\ d.k\ a\ (c \odot d))\ ts\ dt
\end{aligned}
$$

Base case: $ts \Rightarrow [\,]$

$$
\begin{aligned}
LHS &= f'\ (c \odot v)\ [\,]\ [\,] = [\,]\\
RHS &= \mathbf{let}\ dt = \mathsf{dAcc}''_r\ (\odot)\ v\ (\mathsf{zipwith}'_r\ g\ [\,]\ (\mathsf{uAcc}'_r\ (\oplus')\ (\otimes')\ (\mathsf{map}'_r\ k'\ [\,])))\\
&\quad\ \mathbf{in}\ \ \mathsf{zipwith}'_r\ (\lambda a\ d.k\ a\ (c \odot d))\ [\,]\ dt\\
&= \mathbf{let}\ dt = \mathsf{dAcc}''_r\ (\odot)\ v\ (\mathsf{zipwith}'_r\ g\ [\,]\ [\,])\\
&\quad\ \mathbf{in}\ \ \mathsf{zipwith}'_r\ (\lambda a\ d.k\ a\ (c \odot d))\ [\,]\ dt\\
&= \mathsf{zipwith}'_r\ (\lambda a\ d.k\ a\ (c \odot d))\ [\,]\ [\,]\\
&= [\,]
\end{aligned}
$$

Inductive case: $ts \Rightarrow (RNode\ a\ ts) : ss$

$$
\begin{aligned}
LHS &= f'\ (c \odot v)\ ((RNode\ a\ ts) : ss)\\
&= (RNode\ (k\ a\ (c \odot v))\ (f'\ (c \odot v \odot g\ a\ (h\ (RNode\ a\ ts)))\ ts)) : f'\ (c \odot v)\ ss\\
RHS &= \mathbf{let}\ dt = \mathsf{dAcc}''_r\ (\odot)\ v\ (\mathsf{zipwith}'_r\ g\ ((RNode\ a\ ts) : ss)\\
&\qquad\qquad\qquad\qquad (\mathsf{uAcc}'_r\ (\oplus')\ (\otimes')\ (\mathsf{map}'_r\ k'\ ((RNode\ a\ ts) : ss))))\\
&\quad\ \mathbf{in}\ \ \mathsf{zipwith}'_r\ (\lambda a\ d.k\ a\ (c \odot d))\ ((RNode\ a\ ts) : ss)\ dt\\
&= \mathbf{let}\ dt = \mathsf{dAcc}''_r\ (\odot)\ v\ (\mathsf{zipwith}'_r\ g\ ((RNode\ a\ ts) : ss)\\
&\qquad\qquad\qquad\qquad (\mathsf{uAcc}'_r\ (\oplus')\ (\otimes')\ ((RNode\ (k'\ a)\ (\mathsf{map}'_r\ k'\ ts)) : (\mathsf{map}'_r\ k'\ ss))))\\
&\quad\ \mathbf{in}\ \ \mathsf{zipwith}'_r\ (\lambda a\ d.k\ a\ (c \odot d))\ ((RNode\ a\ ts) : ss)\ dt\\
&= \mathbf{let}\ dt = \mathsf{dAcc}''_r\ (\odot)\ v\ (\mathsf{zipwith}'_r\ g\ ((RNode\ a\ ts) : ss)\\
&\qquad\qquad (\mathsf{uAcc}_r\ (\oplus')\ (\otimes')\ (RNode\ (k'\ a)\ (\mathsf{map}'_r\ k'\ ts)) : \mathsf{uAcc}'_r\ (\oplus')\ (\otimes')\ (\mathsf{map}'_r\ k'\ ss)))\\
&\quad\ \mathbf{in}\ \ \mathsf{zipwith}'_r\ (\lambda a\ d.k\ a\ (c \odot d))\ ((RNode\ a\ ts) : ss)\ dt
\end{aligned}
$$

$$
\begin{aligned}
&= \textbf{let } rv = h\ (RNode\ a\ ts) \\
&\qquad dt = \mathsf{dAcc}''_r\ (\odot)\ v\ (\mathsf{zipwith}'_r\ g\ ((RNode\ a\ ts) : ss) \\
&\qquad\qquad (RNode\ rv\ (\mathsf{uAcc}'_r\ (\oplus')\ (\otimes')\ (\mathsf{map}'_r\ k'\ ts)) : \mathsf{uAcc}'_r\ (\oplus')\ (\otimes')\ (\mathsf{map}'_r\ k'\ ss))) \\
&\quad \textbf{in } \mathsf{zipwith}'_r\ (\lambda a\ d.k\ a\ (c \odot d))\ ((RNode\ a\ ts) : ss)\ dt \\
&= \textbf{let } rv = h\ (RNode\ a\ ts) \\
&\qquad dt = \mathsf{dAcc}''_r\ (\odot)\ v\ (RNode\ (g\ a\ rv)\ (\mathsf{zipwith}'_r\ g\ ts\ (\mathsf{uAcc}'_r\ (\oplus')\ (\otimes')\ (\mathsf{map}'_r\ k'\ ts))) : \\
&\qquad\qquad\qquad (\mathsf{zipwith}'_r\ ss\ \mathsf{uAcc}'_r\ (\oplus')\ (\otimes')\ (\mathsf{map}'_r\ k'\ ss))) \\
&\quad \textbf{in } \mathsf{zipwith}'_r\ (\lambda a\ d.k\ a\ (c \odot d))\ ((RNode\ a\ ts) : ss)\ dt \\
&= \textbf{let } rv = h\ (RNode\ a\ ts) \\
&\qquad dt' = \mathsf{dAcc}''_r\ (\odot)\ (v \odot g\ a\ rv)\ (\mathsf{zipwith}'_r\ g\ ts\ (\mathsf{uAcc}'_r\ (\oplus')\ (\otimes')\ (\mathsf{map}'_r\ k'\ ts))) \\
&\qquad dt'' = \mathsf{dAcc}''_r\ (\odot)\ v\ (\mathsf{zipwith}'_r\ ss\ \mathsf{uAcc}'_r\ (\oplus')\ (\otimes')\ (\mathsf{map}'_r\ k'\ ss)) \\
&\quad \textbf{in } \mathsf{zipwith}'_r\ (\lambda a\ d.k\ a\ (c \odot d))\ ((RNode\ a\ ts) : ss)\ ((RNode\ v\ dt') : dt'') \\
&= \textbf{let } rv = h\ (RNode\ a\ ts) \\
&\qquad dt' = \mathsf{dAcc}''_r\ (\odot)\ (v \odot g\ a\ rv)\ (\mathsf{zipwith}'_r\ g\ ts\ (\mathsf{uAcc}'_r\ (\oplus')\ (\otimes')\ (\mathsf{map}'_r\ k'\ ts))) \\
&\qquad dt'' = \mathsf{dAcc}''_r\ (\odot)\ v\ (\mathsf{zipwith}'_r\ ss\ \mathsf{uAcc}'_r\ (\oplus')\ (\otimes')\ (\mathsf{map}'_r\ k'\ ss)) \\
&\quad \textbf{in } RNode\ (k\ a\ (c \odot v))\ (\mathsf{zipwith}'_r\ (\lambda a\ d.k\ a\ (c \odot d))\ ts\ dt') : \\
&\qquad (\mathsf{zipwith}'_r\ (\lambda a\ d.k\ a\ (c \odot d))\ ss\ dt'') \\
&= (RNode\ (k\ a\ (c \odot v))\ (f'\ (c \odot v \odot g\ a\ (h\ (RNode\ a\ ts)))\ ts)) : (f'\ (c \odot v)\ ss)
\end{aligned}
$$

It follows from the induction above that the theorem holds.