Associativity for Parallel Tree Computation

松崎 公紀 森畑 明昌 胡 振江 武市 正人
Kiminori MATSUZAKI Akimasa MORIHATA Zhenjiang HU Masato TAKEICHI
東京大学大学院情報理工学系研究科
Graduate School of Information Science and Technology, University of Tokyo
{kmatsu,morihata}@ipl.t.u-tokyo.ac.jp {hu,takeichi}@mist.i.u-tokyo.ac.jp

Associativity is one of the most important properties in parallel computation, for example, associativity of list concatenation plays an important role in parallel computation for lists. However, few studies have been devoted to the formalization in the context of parallel computation on trees. In this paper, we observe flexible division of binary trees in which a tree can be divided at any node rather than the root, and formalize a new property called *tree associativity*. We apply this tree-version associativity to the parallel implementation of several computational patterns on trees.

1 Introduction

Associativity is one of the most important properties in parallel computation. In parallel computation on lists, the associativity of list concatenation enables us to divide a list into arbitrary sublists, where the divide-and-conquer approach can be naturally applied. Many researchers have devoted themselves to the development of parallel programs based on the associativity [6, 10, 18, 20], and to the derivation of associative operators [5, 8].

In spite of the success of the associativity on lists, few studies have been devoted to formalizing associativity in the context of parallel computation on *trees*. Trees are important data structures often used to represent structured data such as XML trees, but development of efficient parallel programs for manipulating trees is a hard task due to their ill-balanced structures.

Two major approaches to the parallel computation on trees are the simple divide-and-conquer approach and the tree-contraction approach. The simple divide-and-conquer approach of computing each subtree independently is widely used, but it may be inefficient if the tree is ill-balanced. This is caused by insufficient flexibility in the division of trees, that is, a tree can only be divided at the root node. Tree contractions, first proposed by Miller and Reif [15], are efficient parallel algorithms for trees of arbitrary shapes. Though several studies have been done for the tree contraction algorithms [1, 2, 7, 12, 14, 19], parallelism in the tree contraction algorithms has not been clarified in relation to flexibility of tree division.

In this paper, we propose a novel property held on trees, namely *tree associativity*, and apply it to the parallel computation on trees. We first observe flexible division of trees in which a tree can be divided at an arbitrary node rather than only the root node, and then introduce a new concept of ternary-tree representation of trees. Based on this ternary-tree representation, we formalize the tree associativity, a key property in the parallel computation for trees. We develop efficient parallel implementations of several tree manipulations in a divide-and-conquer manner on the ternary-tree representation under the condition of tree associativity.

The paper is organized as follows. In Section 2 we review notational conventions and introduce a general form of tree manipulations called tree homomorphism, and in Section 3 we review the role of associativity in the parallel computation on lists. In Section 4, based on the observation of flexible tree division we propose the ternary-tree representation and the tree associativity. We then discuss a parallel implementation of tree homomorphisms under tree associativity in Section 5. We remark on related work in Section 6, and conclude the paper in Section 7.

2 Preliminaries

2.1 Notations

In this paper, we borrow the notations of Haskell [17]. In the following, we introduce important notations and datatypes.

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 $\circ$. Sometimes we do not care about actual value of a variable, and in such a case we may denote the value as $\_$.

Infix binary operators will be denoted by $\oplus$ and $\otimes$ as well as arithmetic operators. In addition to arithmetic operations, we use infix operator $\uparrow$ to denote the *max* computation that returns the bigger of the two inputs.

Datatypes

A (nonempty join) list is constructed from a single value or by concatenating two lists. The datatype for lists where every element has type α is defined as follows.

$$\begin{aligned}\text{data } \mathsf{JList}\ \alpha = &\ \mathsf{Sing}\ \alpha \\ \mid &\ \mathsf{Concat}\ (\mathsf{JList}\ \alpha)\ (\mathsf{JList}\ \alpha)\end{aligned}$$

We may use abbreviations, $[x]$ for $\mathsf{Sing}\ x$, and $xs +\!\!+ ys$ for $\mathsf{Concat}\ xs\ ys$. Note that $+\!\!+$ is an associative operator.

A *binary tree* is a tree whose internal nodes have exactly two children. The datatype for binary trees where every leaf has type α and every internal node has type β is defined as follows.

$$\begin{aligned}\text{data } \mathsf{BT}\ \alpha\ \beta = &\ \mathsf{BL}\ \alpha \\ \mid &\ \mathsf{BN}\ \beta\ (\mathsf{BT}\ \alpha\ \beta)\ (\mathsf{BT}\ \alpha\ \beta)\end{aligned}$$

Function $root_b$ takes a binary tree and returns its root node.

$$\begin{aligned}root_b\ (\mathsf{BL}\ a) &= a \\ root_b\ (\mathsf{BN}\ l\ b\ r) &= b\end{aligned}$$

2.2 Tree Homomorphisms

Once a (recursive) datatype is specified, a natural recursive functions on it can be defined along with the specification of the datatype. For binary trees, we consider the following recursive function called tree homomorphism [21, 22].

Definition 1 (Tree Homomorphism) Let k_l and k_n be given functions. Function h is called *tree homomorphism*, if it is defined in the following recursive form.

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

We may denote the tree homomorphism above as $h = (\!|k_l, k_n|\!)_b$. □

An example of tree homomorphism is function *height* that computes the height of a tree.

$$\begin{aligned}height\ (\mathsf{BL}\ a) &= 1 \\ height\ (\mathsf{BN}\ l\ b\ r) &= 1 + (height\ l \uparrow height\ r)\end{aligned}$$

This function is indeed a tree homomorphism $height = (\!|height_l, height_n|\!)_b$ where $height_l\ a = 1$ and $height_n\ l\ b\ r = 1 + (l \uparrow r)$.

Computations on trees may return trees rather than basic values, and for such computations we define the following two tree accumulations [22]. In fact these tree accumulations can be formalized as special instances of the tree homomorphisms[1].

Definition 2 Let k_l and k_n be given functions. Function h^u is called *upwards accumulation*, if it is defined in the following recursive form.

$$\begin{aligned}&h^u\ (\mathsf{BL}\ a) = \mathsf{BL}\ (k_l\ a) \\ &h^u\ (\mathsf{BN}\ l\ b\ r) \\ &\quad = \mathbf{let}\ l' = h^u\ l;\ r' = h^u\ r \\ &\qquad \mathbf{in}\ \ \mathsf{BN}\ l'\ (k_n\ (root_b\ l')\ b\ (root_b\ r'))\ r'\end{aligned}$$ □

Definition 3 Let g_l and g_r be given functions. Function h^d is called *downwards accumulation*, if it is defined in the following recursive form with an additional parameter c.

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

[1] The downwards accumulation can be formalized as a higher-order tree homomorphism.

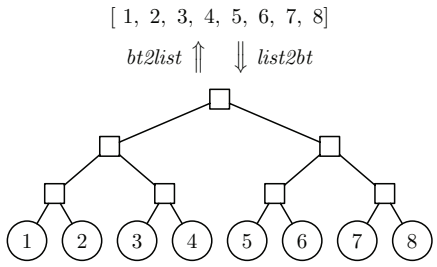

Fig. 1: Binary-tree representation of dividing a list.

The tree homomorphisms and the tree accumulations have been studied as basic computational patterns on trees (called *tree skeletons*). See [13] for examples developed with these functions.

3 Associativity for Parallel Computation on Lists

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

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

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

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

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

Here equation $\mathit{bt2list} \circ \mathit{list2bt} = \mathit{id}$ holds. Figure 1 shows an example of the division of a list.

Computation of divide-and-conquer parallel programs for lists can be performed along the

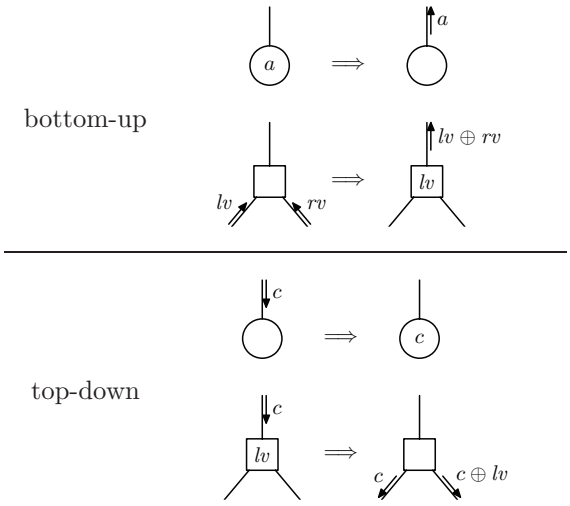

Fig. 2: Illustration of two sweeps for scans.

binary-tree representation of lists. For example, let k be a given function and $\oplus$ be an associative operator, list homomorphism $(\!|k, \oplus|\!)_l$ defined as

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

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

$$\begin{aligned} (\!|k, \oplus|\!)_l = (\!|k_l, k_n|\!)_b \circ \mathit{list2bt} \\ \textbf{where}\ k_l\ a &= k\ a \\ k_n\ l\ \_\ r &= l \oplus r \end{aligned}$$

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

In the following, we will see a parallel implementation of more involved computation for lists called scans or prefix sums. A well-known parallel implementation of scans was developed by Kogge and Stone [11], and it consists of two steps. With the binary-tree representation of lists, we can formalize the implementation of scans as two sweeps on the representation: a bottom-up sweep followed by a top-down sweep. An intuitive definition of the two sweeps are given in Fig. 2. Note that since both sweeps are applied to the two subtrees independently, we can implement a parallel program in a naive divide-and-conquer style on the binary-tree

representation. The parallel algorithm computes scans in logarithmic time to the number of elements of the list for the balanced binary-tree representation with its height being logarithmic to the number of elements.

4 Ternary-Tree Representation and Tree Associativity

One naive way to divide a binary tree is to divide it at the root node into two subtrees, and based on this division we can compute tree homomorphisms in parallel in a divide-and-conquer way. Ill-balanced tree structures, however, may spoil the parallelism, and in the worst case the naive divide-and-conquer programs may run as slow as sequential ones. The problem is due to insufficient flexibility in dividing a tree.

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

4.1 Division of Binary Trees and Ternary-Tree Representation

Consider dividing a binary tree at any node instead of just the root. Let x be a node in a binary tree, we can divide the tree at node x into the following three parts: the left subtree of x, the right subtree of x, and the other nodes including x, as shown in Fig. 3. We first define two keywords *terminal node* and *segment* to discuss the division of binary trees.

Definition 4 (Terminal Node) We call the node at which a binary tree is divided (e.g., x in Fig. 3) as *terminal node*. □

Definition 5 (Segment) We call a set of consecutive nodes as a *segment*. □

Different from a subtree, a segment may not have all the descendants in the original tree. Note that all the segments in this paper form binary trees.

In principle we may divide a binary tree at any internal node, but in practice we should impose some conditions due to the non-linear structure of the tree. As seen in Fig. 4, when a segment has k terminal nodes it has $2k$ child segments, and such a segment with more than two children complicates the handling of the global structure. Consistent handling of the global structure of segments requires the global structure be kept binary through divisions, and therefore we restrict each segment to have at most one terminal node. Under this restriction each segment has zero or two child segments and the global structure of the segments forms a binary tree. Note that we can obtain at least one division satisfying this restriction because dividing a tree at the root node always satisfies the restriction. The flexible division of binary trees yields a lot of ternary-tree representations for a given binary tree. For example, for the binary tree with seven nodes in Fig. 3, there are five possible ternary-tree representations.

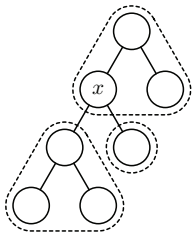

Fig. 3: Dividing a binary tree into three segments at a terminal node x.

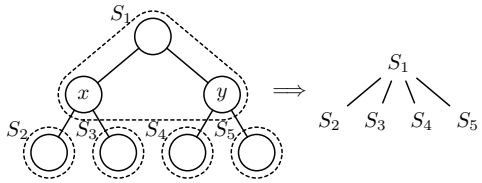

Fig. 4: When segment S_1 has two terminal nodes x and y, it has four child segments.

We divide a binary tree recursively until each segment consists of only one node. Since division of a binary tree yields three segments, we represent the recursive division of a binary tree as a ternary tree. For each division of a segment, we insert a ternary internal node and put the left-child segment to its left child, the parent segment to its center child, and the right-child segment to its right child, re-

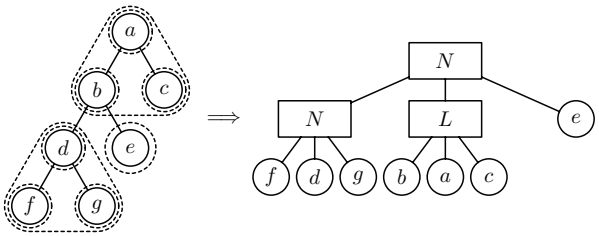

Fig. 5: Example of ternary-tree representation.

spectively. Figure 5 illustrates a ternary-tree representation of a binary tree. A leaf in the ternary-tree representation corresponds to a node in the original binary tree, and a subtree in the ternary-tree representation corresponds to a segment that appears during the recursive division.

The ternary-tree representation should be defined in such a way that the original binary-tree structure can be restored. One naive way to achieve this is to embed a pointer to the terminal node in each internal node. This formalization with pointers, however, makes it hard to discuss the characteristics of the ternary-tree representation. We examine another specification without pointers where we associate one of the following labels into each internal node.

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

A terminal node corresponding to a previous division must not be included in the parent segment after dividing the segment, since the parent segment always has a terminal node at which the segment is divided. Because of the restriction that a segment has at most one terminal node, the three labels cover all the cases of the ternary-tree representation.

We can find the terminal node on the ternary-tree representation by using these labels. For a given internal node, traversing the ternary tree from its center child to the leaves by selecting recursively left/right child at node TNL/TNR. For example, in Fig. 5 the global binary tree is divided at node b, which is given on the ternary-tree representation by traversing from the center child of the root node to the left child.

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

$$
\begin{aligned}
\mathrm{data}\,\mathsf{TT}\,\alpha\,\beta = {} & \mathsf{TLL}\ \alpha \\
| \ & \mathsf{TLN}\,\beta \\
| \ & \mathsf{TNN}\,(\mathsf{TT}\,\alpha\,\beta)\,(\mathsf{TT}\,\alpha\,\beta)\,(\mathsf{TT}\,\alpha\,\beta) \\
| \ & \mathsf{TNL}\ (\mathsf{TT}\,\alpha\,\beta)\,(\mathsf{TT}\,\alpha\,\beta)\,(\mathsf{TT}\,\alpha\,\beta) \\
| \ & \mathsf{TNR}\,(\mathsf{TT}\,\alpha\,\beta)\,(\mathsf{TT}\,\alpha\,\beta)\,(\mathsf{TT}\,\alpha\,\beta)
\end{aligned}
$$

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

Not all the ternary trees of the type above represent binary trees. Since the original binary tree has no terminal node before division, the root of a ternary tree should be TNN or TLL. Since a new terminal node is included in the parent segment for each division, and thus the center child of each internal node should be either TNL, TNR or TLN. For an internal node labeled TNN, its left-child and right-child segments do not have any terminal node, and should be either TNN or TLL. For an internal node labeled TNL, its left child segment has a terminal node and thus the left child should be either TNL, TNR or TLN, while the right child should be labeled TNN or TLL. A node labeled TNR is symmetric to the node labeled TNL.

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

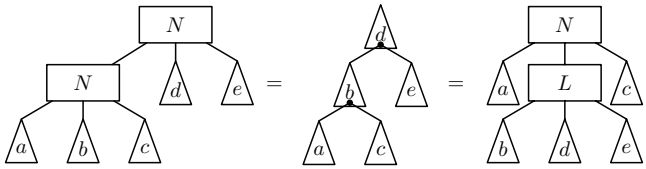

Fig. 6: Illustration of the equation $\mathsf{TNN}\ (\mathsf{TNN}\ a\ b\ c)\ d\ e \equiv_{tt2bt} \mathsf{TNN}\ a\ (\mathsf{TNL}\ b\ d\ e)\ c$.

$$
\begin{aligned}
&tt2bt :: \mathsf{TT}\ \alpha\ \beta \to \mathsf{BT}\ \alpha\ \beta\\
&tt2bt\ t = tt2bt'\ t\ \_\ \_\\
&\\
&tt2bt'\ (\mathsf{TLL}\ a)\ \_\ \_ = \mathsf{BL}\ a\\
&tt2bt'\ (\mathsf{TLN}\ b)\ x\ y\ \ = \mathsf{BN}\ x\ b\ y\\
&tt2bt'\ (\mathsf{TNN}\ l\ n\ r)\ \_\ \_\\
&\quad = tt2bt'\ n\ (tt2bt'\ l\ \_\ \_)\ (tt2bt'\ r\ \_\ \_)\\
&tt2bt'\ (\mathsf{TNL}\ l\ n\ r)\ x\ y\\
&\quad = tt2bt'\ n\ (tt2bt'\ l\ x\ y)\ (tt2bt'\ r\ \_\ \_)\\
&tt2bt'\ (\mathsf{TNR}\ l\ n\ r)\ x\ y\\
&\quad = tt2bt'\ n\ (tt2bt'\ l\ \_\ \_)\ (tt2bt'\ r\ x\ y)
\end{aligned}
$$

In the definition above, the second and the third arguments of the function $tt2bt'$ represent the left and the right subtrees of the terminal node. Since the segments corresponding to TLL and TNN have no terminal node by definition, the arguments are don't-care values for them and in particular for the root node.

We briefly remark on the balancing property of the ternary-tree representation.

Lemma 1 *For any given binary tree of N nodes, there exist a balanced ternary-tree representation of it whose height is* $\mathrm{O}(\log N)$.

Proof Sketch: A balanced ternary-tree representation is given by simulating the tree contraction algorithm proposed by Abrahamson et al. [1] using contracting operations that assign one of the three labels. □

4.2 Tree Associativity

In parallel programming, associativity enables us to change the order of local computations to perform then in parallel. As seen in Section 3, associativity of the list concatenation, $+\!\!+$, plays an important role in parallel programming on lists by providing flexible division of lists. We have introduced the ternary-tree representation for the flexible division of binary trees, and now we formalize the tree-version associativity based on the ternary-tree representation.

On the ternary-tree representation, changing the order of local computation corresponds to swapping an internal node with its parent. As our running example, consider a ternary-tree representation whose root node is TNN and its left child is also TNN (Fig. 6, left). Let a, b, c, d, and e denote subtrees, then we can denote such a tree as follows.

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

Note that two segments corresponding to b and d have a terminal node. This ternary tree represents a binary tree in which the root segment d has two child segments b on the left and e on the right, and the segment b has two child segments a on the left and c on the right (Fig. 6, center). The above ternary-tree representation can be obtained by the division at the terminal node in d followed by the division at the terminal node in b. In fact, we can swap the order of divisions for this binary tree, that is, we divide the tree at the terminal node in b and then divide the parent segment at the terminal node in d, which yields the following ternary-tree representation (Fig. 6, right).

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

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

$$\mathsf{TNN}\ (\mathsf{TNN}\ a\ b\ c)\ d\ e \equiv_{tt2bt} \mathsf{TNN}\ a\ (\mathsf{TNL}\ b\ d\ e)\ c$$

By examining the possible local structures of ternary-tree representations in the same way, we obtain five more equations.

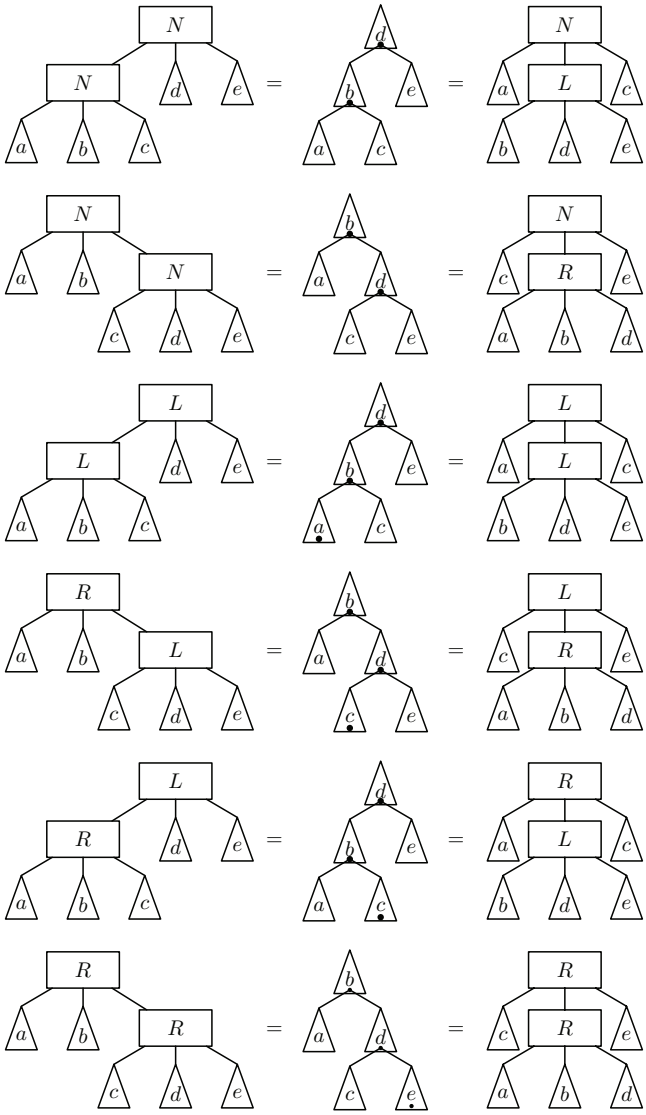

Fig. 7: The six equations of local transformations. A dot in a subtree represents the terminal node.

$$\mathsf{TNN}\ a\ b\ (\mathsf{TNN}\ c\ d\ e) \equiv_{tt2bt} \mathsf{TNN}\ c\ (\mathsf{TNR}\ a\ b\ d)\ e$$
$$\mathsf{TNL}\ (\mathsf{TNL}\ a\ b\ c)\ d\ e \equiv_{tt2bt} \mathsf{TNL}\ a\ (\mathsf{TNL}\ b\ d\ e)\ c$$
$$\mathsf{TNR}\ a\ b\ (\mathsf{TNL}\ c\ d\ e) \equiv_{tt2bt} \mathsf{TNL}\ c\ (\mathsf{TNR}\ a\ b\ d)\ e$$
$$\mathsf{TNL}\ (\mathsf{TNR}\ a\ b\ c)\ d\ e \equiv_{tt2bt} \mathsf{TNR}\ a\ (\mathsf{TNL}\ b\ d\ e)\ c$$
$$\mathsf{TNR}\ a\ b\ (\mathsf{TNR}\ c\ d\ e) \equiv_{tt2bt} \mathsf{TNR}\ c\ (\mathsf{TNR}\ a\ b\ d)\ e$$

We have in total six equations, which are illustrated in Fig. 7. Note that, we do not have equations for two forms, ($\mathsf{TNL}\ a\ b\ (\mathsf{TNN}\ c\ d\ e)$) and ($\mathsf{TNR}\ (\mathsf{TNN}\ a\ b\ c)\ d\ e$), due to the restriction that a segment must not have more than one terminal node.

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

Definition 6 A ternary-tree representation is said to be plain if it consists of only the constructors TLL, TLN, and TNN. □

In other words, a plain ternary-tree representation does not have any constructor TNL or TNR.

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

Lemma 2 *For a given binary tree, there is exactly one plain ternary-tree representation of it.*
Proof: As stated in Section 4.1, the center child of an internal node must be labeled as either TNL, TNR, or TLN, since the corresponding segment must have a terminal node. Therefore, the center child of a plain ternary tree is TLN, which represents the division at the root node. Since the root node is unique in a tree, there is at most one plain ternary tree representing a binary tree.

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

$$\begin{aligned} &\mathit{bt2plain}\ (\mathsf{BL}\ a) = \mathsf{TLL}\ a \\ &\mathit{bt2plain}\ (\mathsf{BN}\ l\ b\ r) \\ &\quad = \mathsf{TNN}\ (\mathit{bt2plain}\ l)\ (\mathsf{TLN}\ b)\ (\mathit{bt2plain}\ r) \end{aligned}$$

This function returns a plain ternary-tree representation since there are only three constructors TLL, TNN, and TLN in the function body. □

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

Lemma 3 *A valid ternary tree can be transformed into a plain ternary tree by the following two equations.*

$$\begin{aligned} &\mathsf{TNN}\ (\mathsf{TNN}\ a\ b\ c)\ d\ e \equiv_{tt2bt} \mathsf{TNN}\ a\ (\mathsf{TNL}\ b\ d\ e)\ c \\ &\mathsf{TNN}\ a\ b\ (\mathsf{TNN}\ c\ d\ e) \equiv_{tt2bt} \mathsf{TNN}\ c\ (\mathsf{TNR}\ a\ b\ d)\ e \end{aligned}$$

Proof: A single top-down algorithm with the following operations achieves the transformation.

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

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

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

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

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

Given two ternary-tree representation for the same binary tree, we can transform one to the other as the following lemma states.

Lemma 4 *Given two ternary trees representing the same binary tree, one tree can be transformed into the other by the two equations in Lemma 3.*
Proof: The lemma follows from Lemmas 2 and 3. □

The Lemmas 3 and 4 also point out that the first two equations suffice for transforming ternary trees. In fact, among the six equations on the ternary-tree representation, the latter four equations can be derived from the former two equations. Generalizing TNN, TNL, and TNR to three functions g_n, g_l, g_r, we obtain the following lemma.

Lemma 5 *Let g_n be a function satisfying the following proposition,*

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

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

$$\begin{aligned} g_n\ (g_n\ a\ b\ c)\ d\ e &= g_n\ a\ (g_l\ b\ d\ e)\ c \\ g_n\ a\ b\ (g_n\ c\ d\ e) &= g_n\ c\ (g_r\ a\ b\ d)\ e \end{aligned}$$

Then, the following four equations hold.

$$\begin{aligned} g_l\ (g_l\ a\ b\ c)\ d\ e &= g_l\ a\ (g_l\ b\ d\ e)\ c \\ g_r\ a\ b\ (g_l\ c\ d\ e) &= g_l\ c\ (g_r\ a\ b\ d)\ e \\ g_l\ (g_r\ a\ b\ c)\ d\ e &= g_r\ a\ (g_l\ b\ d\ e)\ c \\ g_r\ a\ b\ (g_r\ c\ d\ e) &= g_r\ c\ (g_r\ a\ b\ d)\ e \end{aligned}$$

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

$$\begin{aligned} & g_n\ x\ (g_l\ (g_l\ a\ b\ c)\ d\ e)\ y \\ =\ & g_n\ (g_n\ x\ (g_l\ a\ b\ c)\ y)\ d\ e \\ =\ & g_n\ (g_n\ (g_n\ x\ a\ y)\ b\ c)\ d\ e \\ =\ & g_n\ (g_n\ x\ a\ y)\ (g_l\ b\ d\ e)\ c \\ =\ & g_n\ x\ (g_l\ a\ (g_l\ b\ d\ e)\ c)\ y \end{aligned}$$

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

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

also holds by the first proposition.

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

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

Definition 7 (Tree Associativity) Functions g_n, g_l, and g_r are *tree associative*, if the following two equations hold for any a, b, c, d, and e.

$$\begin{aligned} & g_n\ (g_n\ a\ b\ c)\ d\ e = g_n\ a\ (g_l\ b\ d\ e)\ c \\ & g_n\ a\ b\ (g_n\ c\ d\ e) = g_n\ c\ (g_r\ a\ b\ d)\ e \end{aligned}$$ □

Lemma 6 *The three constructors of the ternary-tree representation,* TNN*,* TNL *and* TNR*, are tree associative modulo function tt2bt.*

Proof: The constructors satisfy the following two equations of tree associativity if we use $\equiv_{tt2bt}$ instead of $=$.

5 Implementation of Tree Homomorphisms

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

5.1 Conditions for Implementing Tree Homomorphisms

We define a natural computational pattern on the ternary-tree representation.

Definition 8 (Ternary-Tree Homomorphism) Let k'_l and k'_n be given functions, and g'_n, g'_l, and g'_r be tree associative functions. Function h' is called *ternary-tree homomorphism*, if it is defined on ternary trees as follows.

$$\begin{aligned} h'\ (\mathsf{TLL}\ a) &= k'_l\ a \\ h'\ (\mathsf{TLN}\ b) &= k'_n\ b \\ h'\ (\mathsf{TNN}\ l\ n\ r) &= g'_n\ (h'\ l)\ (h'\ n)\ (h'\ r) \\ h'\ (\mathsf{TNL}\ l\ n\ r) &= g'_l\ (h'\ l)\ (h'\ n)\ (h'\ r) \\ h'\ (\mathsf{TNR}\ l\ n\ r) &= g'_r\ (h'\ l)\ (h'\ n)\ (h'\ r) \end{aligned}$$

We may denote a ternary-tree homomorphism as $h' = (\!|k'_l, k'_n, g'_n, g'_l, g'_r|\!)_t$. □

As we have seen in the previous section, the ternary-tree representation provides great flexibility in terms of the order of local computations, and the flexibility supports parallel computation on the ternary-tree representation. However, the flexibility of the ternary-tree representation imposes some conditions on the implementation of tree homomorphisms. In the following, we specify the conditions for implementing tree homomorphism $(\!|k_l, k_n|\!)_b$ by ternary-tree homomorphism $(\!|k'_l, k'_n, g'_n, g'_l, g'_r|\!)_t$.

The first condition is that the ternary-tree homomorphism should simulate the tree homomorphism on the plain ternary trees. We can formalize this condition by induction on the structure of binary trees. For the base case, i.e., $(\mathsf{BL}\ a)$, the results of the tree homomorphism and the ternary-tree homomorphism are given as follows.

$$(\!|k_l, k_n|\!)_b\ (\mathsf{BL}\ a) = k_l\ a$$

$$\begin{aligned} & (\!|k'_l, k'_n, g'_n, g'_l, g'_r|\!)_t\ (bt2plain\ (\mathsf{BL}\ a)) \\ & \quad = (\!|k'_l, k'_n, g'_n, g'_l, g'_r|\!)_t\ (\mathsf{TLL}\ a) \\ & \quad = k'_l\ a \end{aligned}$$

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

$$\begin{aligned} & (\!|k_l, k_n|\!)_b\ (\mathsf{BN}\ l\ b\ r) \\ & \quad = k_n\ ((\!|k_l, k_n|\!)_b\ l)\ b\ ((\!|k_l, k_n|\!)_b\ r) \end{aligned}$$

$$
\begin{aligned}
&(\!|k'_l, k'_n, g'_n, g'_l, g'_r|\!)_t\ (bt2plain\ (\mathsf{BN}\ l\ b\ r))\\
&\quad = (\!|k'_l, k'_n, g'_n, g'_l, g'_r|\!)_t\\
&\qquad\quad (\mathsf{TNN}\ (bt2plain\ l)\ (\mathsf{TLN}\ b)\ (bt2plain\ r))\\
&\quad = g'_n\ ((\!|k'_l, k'_n, g'_n, g'_l, g'_r|\!)_t\ (bt2plain\ l))\ (k'_n\ b)\\
&\qquad\quad ((\!|k'_l, k'_n, g'_n, g'_l, g'_r|\!)_t\ (bt2plain\ r))
\end{aligned}
$$

With the induction hypothesis

$$(\!|k_l, k_n|\!)_b\ x = (\!|k'_l, k'_n, g'_n, g'_l, g'_r|\!)_t\ (bt2plain\ x)$$

for $x = l$ and $x = r$, we have

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

where l' and r' denote the result values of left and right subtrees. Note that l' and r' may have any value in the range of the tree homomorphism. The second condition is that three functions g'_n, g'_l and g'_r should be tree associative by definition.

Note that these two conditions also form a sufficient condition for the implementation of the tree homomorphism by the ternary-tree homomorphism. By Lemma 3, computation on any ternary-tree representation is equivalent to that on the plain ternary tree representing the same binary-tree if the functions are tree associative. The induction guarantees the correctness of the ternary-tree homomorphism on the plain ternary trees.

The following lemma summarizes the discussion.

Lemma 7 *The necessary and sufficient condition for implementing tree homomorphism* $(\!|k_l, k_n|\!)_b$ *by ternary-tree homomorphism* $(\!|k'_l, k'_n, g'_n, g'_l, g'_r|\!)_t$ *is that the functions satisfy the following three conditions:*

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

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

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

Lemma 8 *Tree homomorphism* $(\!|k_l, k_n|\!)_b$ *can be implemented by a ternary-tree homomorphism* $(\!|k'_l, k'_n, g'_n, g'_l, g'_r|\!)_t$ *with the parameter functions defined as follows.*

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

Proof: We can prove this lemma by checking the equations in Lemma 7. The first equation holds by the definition of k'_l. The second equation holds as the following calculation shows.

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

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

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

$$
\begin{aligned}
\mathrm{RHS} &= g'_n\ a\ (\lambda x\ y.\ f_d\ (f_b\ x\ y)\ e)\ c\\
&= (\lambda x\ y.\ f_d\ (f_b\ x\ y)\ e)\ a\ c\\
&= f_d\ (f_b\ a\ c)\ e
\end{aligned}
$$

We can prove the other equation

$$g'_n\ a\ f_b\ (g'_n\ c\ f_d\ e) = g'_n\ c\ (g'_r\ a\ f_b\ f_d)\ e$$

easily in the same manner, where both sides are reduced into $(f_b\ a\ (f_d\ c\ e))$. □

In general new functions generated by g'_l and g'_r expand in terms of their size and computational cost. For efficient implementation, we attach some requirements on the size of generated functions, and a sufficient but a bit strict requirement is to limit the size of functions to a certain constant. We can find another relaxed requirement named as uniform closure property in the discussion by Miller and Teng [16].

In the following, we demonstrate how to find a set of suitable functions of the ternary-tree homomorphism. A systematic way to derive the set of functions is the *generalization-and-test* approach, which has been studied for the derivation of parallel programs for lists [4, 8]. In this approach, we start at a functional form given by templatization of the function for internal nodes. We then test whether it is closed under generating functions or generalize the functional form until the form is closed.

We now show the derivation of an efficient ternary-tree homomorphism using the tree homomorphism *height* in Section 2.2 as an example. The function *height* is a tree homomorphism $(\![height_l, height_n]\!)_b$ where the function $height_n$ is defined as follows.

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

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

$$f_a = \lambda x\ y.\ a + (x \uparrow y)$$

where a denotes a value introduced by the templatization of the function $height_n$. Then, we simulate the generation of functions by g'_l and g'_r using instances of the form. By substituting instances for the arguments of g'_l, we obtain a new function as follows.

$$\begin{aligned}
&g'_l\ f_l\ f_n\ r\\
&\quad = \lambda x\ y.\ (\lambda x'\ y'.\ l + (x' \uparrow y'))\\
&\qquad\qquad ((\lambda x'\ y'.\ n + (x'' \uparrow y''))\ x\ y)\ r\\
&\quad = \lambda x\ y.\ (\lambda x'\ y'.\ l + (x' \uparrow y'))\ (x \uparrow y)\ r\\
&\quad = \lambda x\ y.\ l + ((x \uparrow y) \uparrow r)\\
&\quad = \lambda x\ y.\ (l + r) \uparrow (l + (x \uparrow y))
\end{aligned}$$

Unfortunately, the above function is not in the original form. Therefore, we again abstract the function and obtain the following form of functions. Note that the new form is a generalized one of the original form.

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

We can prove that the new form is closed under generating functions by g'_l and g'_r, as the following calculations show.

$$\begin{aligned}
&g'_l\ f_{(a_l,b_l)}\ f_{(a_n,b_n)}\ r\\
&\quad = \lambda x\ y.\ f_{(a_n,b_n)}\ (f_{(a_l,b_l)}\ x\ y)\ r\\
&\quad = \lambda x\ y.\ a_n \uparrow (b_n + a_l) \uparrow (b_n + r)\\
&\qquad\qquad \uparrow (b_n + b_l + (x \uparrow y))\\
&\quad = \lambda x\ y.\ f_{(a_n \uparrow (b_n+a_l) \uparrow (b_n+r),\, b_n+b_l)}\ x\ y
\end{aligned}$$

$$\begin{aligned}
&g'_r\ l\ f_{(a_n,b_n)}\ f_{(a_r,b_r)}\\
&\quad = \lambda x\ y.\ f_{(a_n,b_n)}\ l\ (f_{(a_r,b_r)}\ x\ y)\\
&\quad = \lambda x\ y.\ a_n \uparrow (b_n + l) \uparrow (b_n + a_r)\\
&\qquad\qquad \uparrow (b_n + b_r + (x \uparrow y))\\
&\quad = \lambda x\ y.\ f_{(a_n \uparrow (b_n+l) \uparrow (b_n+a_r),\, b_n+b_r)}\ x\ y
\end{aligned}$$

Based on these calculations, we can use the functions g'_l and g'_r for implementing a ternary-tree homomorphism. Noting that the functional form is preserved through the computation of the ternary-tree homomorphism, we can simplify the definition a bit. By substituting pair (a, b) for function $f_{(a,b)}$, and with Lemma 8, we have the following ternary-tree homomorphism $(\![k_l, k_n, g_n, g_l, g_r]\!)_t$ for the tree homomorphism *height*, where the five functions are defined as follows.

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

5.2 Implementation of Tree Accumulations

As seen in Section 3, the parallel implementation of *scan* consists of the bottom-up and the top-down sweeps on the binary-tree representation of lists. In this section, we develop implementations of the two tree accumulations in a similar way on the ternary-tree representation of binary trees.

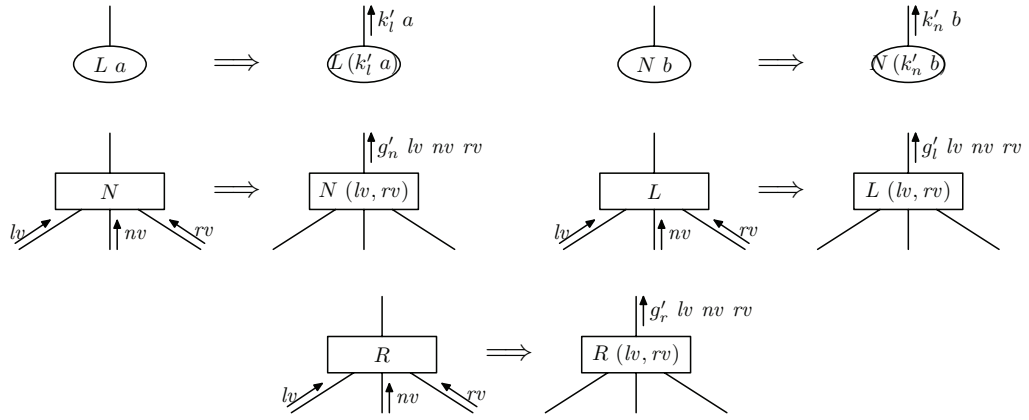

Fig. 8: Illustration of the bottom-up sweep for the upwards accumulation.

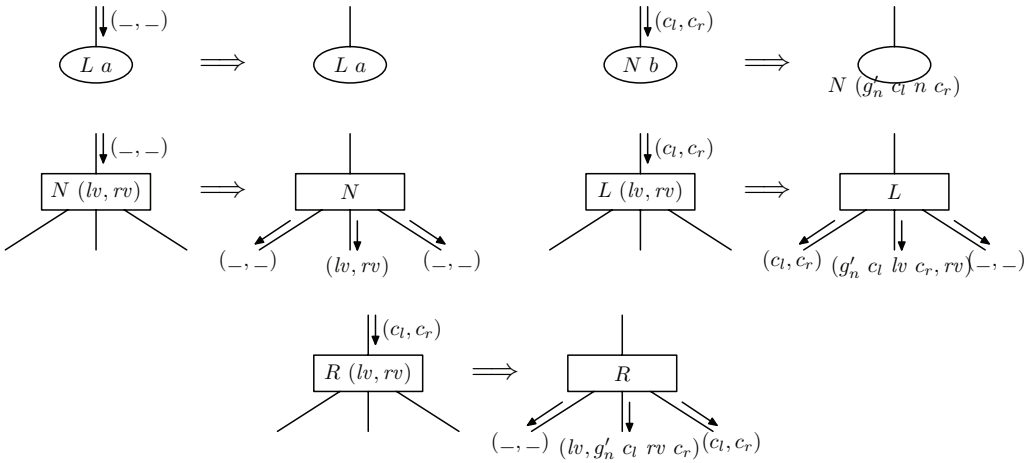

Fig. 9: Illustration of the top-down sweep for the upwards accumulation.

5.2.1 Upwards Accumulation

The upwards accumulation takes two parameter functions k_l and k_n as in Definition 2, which are used in computing tree homomorphism for each subtree. We assume the same condition for the functions, and we define functions k'_l, k'_n, g'_n, g'_l and g'_r in the same way as in Lemma 7.

The computation of the upwards accumulation can be implemented on the ternary-tree representation with a bottom-up sweep (Fig. 8) followed by a top-down sweep (Fig. 9). The bottom-up sweep compute tree homomorphisms along the structure of ternary-tree representation, and at the same time it puts two values from left and right subtrees on each internal node. The top-down sweep computes the values of upwards accumulation by passing a pair of values from the root. The passed values are $-$'s at the beginning. Two values put on each internal node are used in updating the values to the center subtree whose corresponding segment locates above in the original binary tree.

In the bottom-up and top-down sweeps, the computations on the three subtrees are independent and thus we can implement the upwards accumulation in parallel in a divide-and-conquer manner on the ternary-tree representation.

5.2.2 Downwards Accumulation

The downwards accumulation takes two parameter functions g_l and g_r as in Definition 3. For the implementation of the downwards accumulation on the ternary-tree representation, we require some condi-

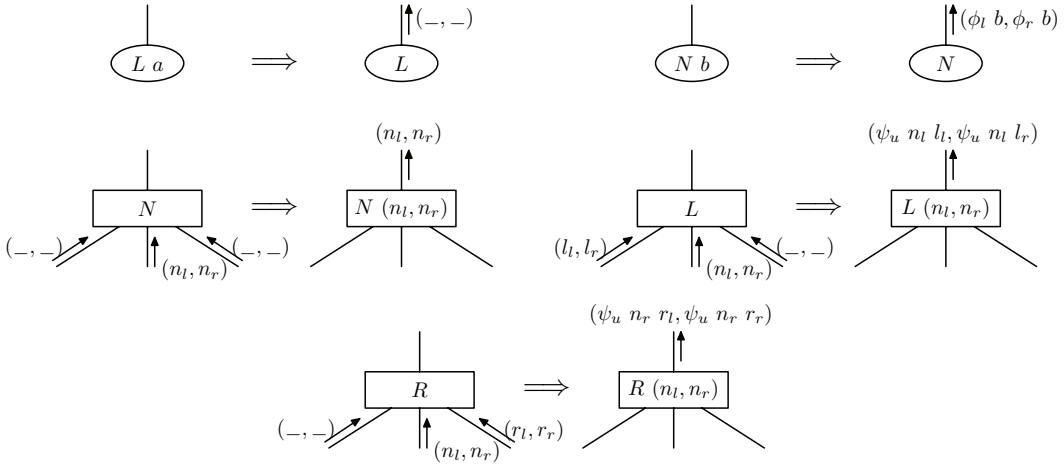

Fig. 10: Illustration of the bottom-up sweep for the downwards accumulation.

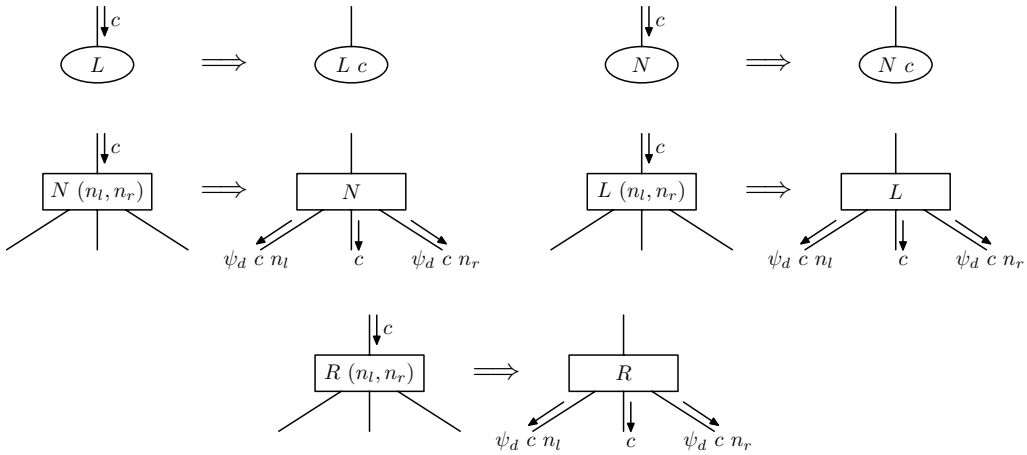

Fig. 11: Illustration of the top-down sweep for the downwards accumulation.

tion on the two parameter functions. Let us assume the existence of four auxiliary functions ϕ'_l, ϕ'_r, ψ_u, and ψ_d satisfying the following three equations.

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

If we do not care about efficiency of the functions there always exist such four auxiliary functions, which are given in a similar way to Lemma 8.

The computation of the downwards accumulation can also be implemented on the ternary-tree representation with a bottom-up sweep (Fig. 10) followed by a top-down sweep (Fig. 11). The bottom-up sweep computes two values using auxiliary functions and stores two values passed from the center subtree whose corresponding segment locates above in the original binary tree. The top-down sweep computes the values of downwards accumulation using the values stored in the bottom-up sweep. The computations on internal nodes TNN, TNL, and TNR are the same in this computation.

Here again, the computations on the three subtrees are independent and thus we can implement the downwards accumulation in parallel on the ternary-tree representation.

Theorem 1 *Tree homomorphism and two tree accumulations can be implemented in parallel on the ternary-tree representation in* $\mathrm{O}(\log N)$ *steps where* N *is the number of nodes in the corresponding binary tree.*

Proof Sketch: Tree homomorphism can be implemented by a bottom-up sweep and both accumulations can be implemented by a bottom-up sweep followed by a top-down sweep. Every sweep can be implemented in parallel based on a divide-and-conquer manner on ternary-tree representations. Since there exists a balanced ternary-tree representation of height $O(\log N)$ as stated in Lemma 1.

The correctness of the implementation can be proved by showing the following two facts. First, the implementation is correct on the plain ternary-tree representation, which can be shown by induction. Secondly, the implementation is tree associative modulo function *tt2bt* by showing two equations of tree associativity. □

6 Related Work

The basic idea to represent a (binary) tree with a balanced tree structure has been studied so far. One of such representations is a balanced decomposition tree, where a decomposition tree is generated by recursive removal of an edge from a tree. There are many applications on this decomposition tree, especially in computational geometry [3]. There are also studies for deriving such a decomposition tree in parallel [24, 25]. The decomposition tree losses structural information of the original binary tree, only a limited class of computations are applicable to the decomposition tree. The ternary-tree representation in this paper keeps structural information of the original binary tree and thus any computation can be mapped onto it if we do not matter efficiency.

The tree contraction algorithms, whose idea was first introduced by Miller and Reif [15], are very important parallel algorithms for implementing tree manipulations, and have been studied by many researches for many parallel computing models [1, 2, 7, 14, 15]. Gibbons et al. [9] and Skillicorn [21, 22] have discussed the implementation of tree homomorphisms and tree accumulations based on the tree contraction algorithms.

We showed conditions for parallel implementation of tree homomorphisms and tree accumulations in Section 5. The conditions are equivalent in terms of their expressiveness to those given by Abrahamson et al. [1]. The difference is that our conditions are given as closure properties of functions while they formalized based on indexed sets of functions.

One sufficient condition proposed in Section 5 was existence of closed functions, and the same idea was also studied on deriving associative operators for parallel computation on lists. The idea of closed functions was formalized for unary functions as the context preservation theorem [5], and there are systems [8, 23] for automatic parallelization of list programs based on this idea.

7 Conclusion

In this paper, we have proposed a new concept of tree associativity and applied it to development of parallel algorithms on trees. The contributions are summarized as follows.

First, we have observed flexible division of binary trees and proposed the *ternary-tree representation* for parallel computation on trees. We have furthermore formalized associativity on this ternary-tree representation. This tree associativity plays an important role in parallel computation on trees.

Secondly, we have shown the tree homomorphisms and the tree accumulations can be computed in parallel on the ternary-tree representation. We have given a condition for implementing tree homomorphisms by providing a definition of functions, which is equivalent to that of the tree contraction algorithms.

The six equations between ternary-tree representations shown in Fig. 7 suggest possibility of local balancing with low cost. Developing algorithms for dynamic balancing of ternary-trees after inserting or deleting nodes is another interesting and important future work that brings the concept of ternary-tree representation for parallel computing on trees into practice.

Acknowledgments This work was partially supported by Japan Society for the Promotion of Science, Grant-in-Aid for Scientific Research (B) No. 17300005, and the Ministry of Education, Culture, Sports, Science and Technology, Grant-in-Aid for Young Scientists (B) No. 18700021.

References

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

[2] 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) (extended abstract). In *High Performance Computing — HiPC 2002, 9th International Conference, Proceedings*, Vol. 2552 of *Lecture Notes in Computer Science*, pp. 63–78. Springer, 2002.

[3] B. Chazelle. A theorem on polygon cutting with applications. In *23rd Annual Symposium on Foundations of Computer Science*, pp. 339–349. IEEE Press, 1982.

[4] W.-N. Chin, S.-C. Khoo, Z. Hu, and M. Takeichi. Deriving parallel codes via invariants. In *Static Analysis, 7th International Symposium, SAS 2000, Proceedings*, Vol. 1824 of *Lecture Notes in Computer Science*, pp. 75–94. Springer, 2000.

[5] W.-N. Chin, A. Takano, and Z. Hu. Parallelization via context preservation. In *Proceedings of the 1998 International Conference on Computer Languages, ICCL '98*, pp. 153–162. IEEE Computer Society, 1998.

[6] M. Cole. Parallel programming with list homomorphisms. *Parallel Processing Letters*, 5(2):191–203, 1995.

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

[8] A. L. Fisher and A. M. Ghuloum. Parallelizing complex scans and reductions. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation (PLDI)*, Vol. 29 of *SIGPLAN Notices*, pp. 135–146. ACM Press, 1994.

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

[10] 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, May 1997.

[11] P. M. Kogge and H. S. Stone. A parallel algorithm for the efficient solution of a general class of recurrence equations. *IEEE Transactions on Computers*, 22(8):786–793, 1973.

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

[13] K. Matsuzaki, Z. Hu, and M. Takeichi. Parallelization with tree skeletons. In *Euro-Par 2003, Parallel Processing, 9th International Euro-Par Conference, Proceedings*, Vol. 2790 of *Lecture Notes in Computer Science*, pp. 789–798. Springer, 2003.

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

[15] G. L. Miller and J. H. Reif. Parallel tree contraction and its application. In *26th Annual Symposium on Foundations of Computer Science*, pp. 478–489. IEEE Computer Society, 1985.

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

[17] 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/`, 1999.

[18] W. M. Pottenger. The role of associativity and commutativity in the detection and transformation of loop-level parallelism. In *ICS '98, Proceedings of the 1998 International Conference on Supercomputing*, pp. 188–195. ACM Press, 1988.

[19] J. H. Reif and S. R. Tate. Dynamic parallel tree contraction (extended abstract). In *SPAA '94: Proceedings of the 6th Annual ACM Symposium on Parallel Algorithms and Architectures*, pp. 114–121. ACM Press, 1994.

[20] D. B. Skillicorn. The Bird-Meertens formalism as a parallel model. In *Software for Parallel Computation*, Vol. 106 of *NATO ASI Series F*, pp. 120–133. Springer, 1993.

[21] D. B. Skillicorn. *Foundations of Parallel Programming*, Vol. 6 of *Cambridge International Series on Parallel Computation*. Cambridge University Press, 1994.

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

[23] D. N. Xu, S.-C. Khoo, and Z. Hu. PType system: A featherweight parallelizability detector. In *Programming Languages and Systems: Second Asian Symposium, APLAS 2004, Proceedings*, Vol. 3302 of *Lecture Notes in Computer Science*, pp. 197–212. Springer, 2004.

[24] 陳, 中野, 増澤, 辻野, 都倉. 単純多角形内の最短経路を求める最適並列アルゴリズム. 電子情報通信学会論文誌 (D-I), J75-D-I(12):814–825, 1991.

[25] 藤原, 陳, 増澤, 都倉. 2 分木の平衡分解木を求めるコスト最適な並列アルゴリズム. 電子情報通信学会論文誌 (D-I), J83-D-I(1):90–98, 2000.