The Third Homomorphism Theorem on Trees

Downward & Upward Lead to Divide-and-Conquer

Akimasa Morihata
University of Tokyo
JSPS Research Fellow
morihata@ipl.t.u-tokyo.ac.jp

Kiminori Matsuzaki
University of Tokyo
kmatsu@ipl.t.u-tokyo.ac.jp

Zhenjiang Hu
National Institute of Informatics
hu@nii.ac.jp

Masato Takeichi
University of Tokyo
takeichi@mist.i.u-tokyo.ac.jp

Abstract

Parallel programs on lists have been intensively studied. It is well known that associativity provides a good characterization for divide-and-conquer parallel programs. In particular, *the third homomorphism theorem* is not only useful for systematic development of parallel programs on lists, but it is also suitable for automatic parallelization. The theorem states that if two sequential programs iterate the same list leftward and rightward, respectively, and compute the same value, then there exists a divide-and-conquer parallel program that computes the same value as the sequential programs.

While there have been many studies on lists, few have been done for characterizing and developing of parallel programs on trees. Naive divide-and-conquer programs, which divide a tree at the root and compute independent subtrees in parallel, take time that is proportional to the height of the input tree and have poor scalability with respect to the number of processors when the input tree is ill-balanced.

In this paper, we develop a method for systematically constructing scalable divide-and-conquer parallel programs on trees, in which two sequential programs lead to a scalable divide-and-conquer parallel program. We focus on paths instead of trees so as to utilize rich results on lists and demonstrate that associativity provides good characterization for scalable divide-and-conquer parallel programs on trees. Moreover, we generalize the third homomorphism theorem from lists to trees. We demonstrate the effectiveness of our method with various examples. Our results, being generalizations of known results for lists, are generic in the sense that they work well for all polynomial data structures.

Categories and Subject Descriptors D.1.3 [*Concurrent Programming*]: Parallel programming; I.2.2 [*Automatic Programming*]: Program transformation; D.1.1 [*Applicative (Functional) Programming*]

General Terms Algorithm, Theory

Keywords Divide-and-conquer, Huet's zippers, Polynomial data structures, The third homomorphism theorem

POPL'09, January 18–24, 2009, Savannah, Georgia, USA.
Copyright © 2009 ACM 978-1-60558-379-2/09/01…$5.00

1. Introduction

What are little boys made of?
Snips and snails, and puppy-dogs' tails
That's what little boys are made of!
What are little girls made of?
Sugar and spice and all things nice
That's what little girls are made of!
(an old nursery rhyme)

What are parallel programs on lists made of? Consider summing up the elements in a list $[a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8]$ as an example. It is easy to derive sequential algorithms; both the rightward summation

$$((((((a_1 + a_2) + a_3) + a_4) + a_5) + a_6) + a_7) + a_8$$

and the leftward summation

$$a_1 + (a_2 + (a_3 + (a_4 + (a_5 + (a_6 + (a_7 + a_8))))))$$

are sequential algorithms. Can we derive a parallel algorithm for summation? Yes, the *divide-and-conquer* summation

$$((a_1 + a_2) + (a_3 + a_4)) + ((a_5 + a_6) + (a_7 + a_8))$$

is a parallel algorithm, which divides the list at its center and computes each part in parallel. What is the key to such divide-and-conquer parallel algorithms? Compare the sequential algorithms with the divide-and-conquer algorithm. The only difference is the structure of the parentheses, and the *associative law* of $+$, namely $a + (b + c) = (a + b) + c$, enables us to rearrange the parentheses.

This observation, i.e., an associative operator provides a divide-and-conquer parallel computation, is formalized as the notion of *list homomorphisms* (Bird 1987). Function h is a list homomorphism if there exists associative operator $\odot$ such that

$$h\,(x \mathbin{+\!\!+} y) \quad = \quad h\,x \odot h\,y$$

holds, where operator $\mathbin{+\!\!+}$ denotes the concatenation of two lists. What is nice about list homomorphisms is the scalability[1]: given a list of length n, list homomorphisms yield linear speedup up to $\mathrm{O}(n/\log n)$ processors. Scalability is one of the most important properties in parallel programming. If a parallel program has good scalability, it shows great speedup that other optimization techniques can barely achieve. If scalability is poor, parallelization is nothing but anti-optimization, because parallelization requires overheads such as communication and synchronization.

Then, what are list homomorphisms made of? It may be surprising that parallel programming on lists is made of writing two

[1] In this paper, we consider scalability with respect to numbers of processors (**?**).

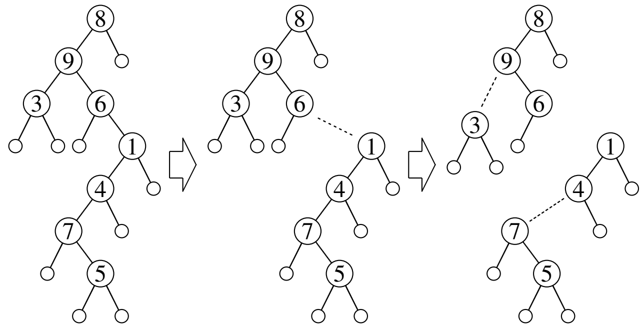

Figure 1. Aggressively dividing binary tree

sequential programs, which proved as *the third homomorphism theorem* (Gibbons 1996). The theorem states that if two sequential programs iterate the same list leftward and rightward, respectively, and compute the same value, then there exists a list homomorphism that computes the same value as these sequential programs. In other words, two sequential programs lead to a scalable divide-and-conquer parallel program. The third homomorphism theorem is useful for developing parallel programs, and automatic parallelization methods have been proposed based on the theorem (Geser and Gorlatch 1999; Morita et al. 2007).

In summary, scalable parallel programs on lists are made of "sugar and spice and all things nice": associative operators and list homomorphisms, which are obtained from two sequential programs.

Then, what are parallel programs on trees made of? For example, let us consider summing up all values in a tree. Someone may think of a divide-and-conquer parallel algorithm raised by subtree structures, i.e., computing independent subtrees in parallel. However, such a naive parallel algorithm is generally not scalable. Its speedup is limited by the height of the input tree, and thus, it has miserable scalability if the input tree is ill-balanced. To obtain better scalability, we need to introduce more aggressive divisions, as outlined in Figure 1. In this case, aggressive divisions yield a scalable parallel algorithm, which computes the summation in logarithmic time on the size of the tree with a sufficient number of processors.

Although we have successfully constructed a scalable divide-and-conquer parallel summation algorithm based on aggressive divisions, it is nontrivial to generalize the algorithm. What algebraic properties will enable us to compute each part in parallel? How can we obtain a parallel program from sequential ones? The nonlinear structure of trees makes it hard to develop parallel programs. Therefore, it has been considered that scalable parallel programs on trees were made of "snips and snails, and puppy-dogs' tails".

In this paper, we explain that scalable parallel programs on trees are in fact made of "sugar and spice and all things nice"—even though they are a bit spicier than those on lists. We focus on the similarity between lists and paths, formalize scalable parallel programs on trees as path-based computations, and demonstrate that associative computations on paths lead to scalable parallel programs. Moreover, we prove that the following proposition holds, which is a tree version of "the third homomorphism theorem" that enables us to derive a scalable parallel program on trees from two sequential programs.

> *If two sequential programs iterate the same tree downward and upward, respectively, and compute the same value, then there exists a scalable divide-and-conquer parallel program that computes the same value as these sequential programs.*

This paper makes three main contributions:

- *Formalization of path-based computations on trees*: We introduce path-based computations, which include downward and upward computations. To express path-based computations, we borrow the notion of *Huet's zippers* (Huet 1997; McBride 2001). Spotlighting paths is the key to developing parallel programs, because it makes theories on lists applicable to trees.
- *Characterization of scalable parallel computations on trees*: We propose an algebraic characterization of scalable parallel programs on trees. The main idea is to consider divide-and-conquer algorithms on one-hole contexts instead of those on trees. We can evaluate programs that belong to our characterization efficiently in parallel: given a tree of n nodes, they finish in $\mathrm{O}(n/p + \log p)$ steps of primitive operations on an exclusive-read/exclusive-write parallel random access machine with p processors. It is worth noting that this computational complexity implies they have good scalability in the sense that they show linear speedup up to $\mathrm{O}(n/\log n)$ processors.
- *"The third homomorphism theorem" on trees*: We prove "the third homomorphism theorem" on trees. It states that if a function is both downward and upward, then it can be evaluated efficiently in parallel. The theorem is useful for systematically developing parallel programs. We will demonstrate its effectiveness with various examples.

Our results are *generic* in the sense that they work well for all *polynomial data structures*, which can capture a large class of algebraic data structures on functional languages; besides, they are generalizations of known results on lists.

The rest of this paper is organized as follows. After the preliminaries in Section 2, we explain our ideas and results on node-valued binary trees in Sections 3 and 4. We develop our theory in Section 3 and present examples in Section 4. After that, we generalize our results to generic trees, namely polynomial data structures, in Section 6. We discuss related and future works in Section 7.

2. The Third Homomorphism Theorem

2.1 Basic definitions

In this paper, we basically borrow notations of functional programming language Haskell (Peyton Jones 2003). The parentheses for function applications may be omitted. Note that applications for functions have higher priority than those for operators, thus $f\ x \oplus y$ means $(f\ x) \oplus y$. Operator $\circ$ denotes a function composition, and its definition is $(f \circ g)(x) = f(g(x))$. Underscore _ is used to express "don't care" pattern. $x :: X$ means the type of x is X.

A list is denoted by brackets split by commas. The list concatenation operator is denoted by $+\!\!+$. Note that $+\!\!+$ is associative. $[A]$ denotes the type of lists whose elements are in A.

The disjoint sum of two sets A and B is denoted by *Either* A B.

$$\mathbf{data}\ \mathit{Either}\ a\ b = \mathit{Left}\ a \mid \mathit{Right}\ b$$

We will write L and R instead of *Left* and *Right* for shorthand.

We will use several standard functions in Haskell, and their definitions are given in Figure 2.

2.2 Right Inverses

Right inverses, which are generalizations of inverse functions, are useful for developing divide-and-conquer parallel programs (Gibbons 1996; Morita et al. 2007).

Definition 2.1 (right inverse)**.** For function $f :: A \to B$, a *right inverse* of f, denoted by f°, is a function satisfying the following equation.

$$f \circ f^\circ \circ f = f \qquad \square$$

$$\begin{array}{lcl}
\mathit{id}\ x & = & x \\
\mathit{fst}\ (a,b) & = & a \\
\mathit{map}\ f\ [] & = & [] \\
\mathit{map}\ f\ ([a] \mathbin{+\!\!+} x) & = & [f\ a] \mathbin{+\!\!+} \mathit{map}\ f\ x \\
\mathit{foldr}\ (\oplus)\ e\ [] & = & e \\
\mathit{foldr}\ (\oplus)\ e\ ([a] \mathbin{+\!\!+} x) & = & a \oplus (\mathit{foldr}\ (\oplus)\ e\ x) \\
\mathit{foldl}\ (\otimes)\ e\ [] & = & e \\
\mathit{foldl}\ (\otimes)\ e\ (x \mathbin{+\!\!+} [a]) & = & (\mathit{foldl}\ (\otimes)\ e\ x) \otimes a
\end{array}$$

Figure 2. Definitions of standard functions

Two things are worth noting. First, as a right inverse exists for any function, it is unnecessary to worry about its existence. Second, a right inverse of a function is generally not unique, and f° denotes one of the right inverses of f.

2.3 List Homomorphisms and The Third Homomorphism Theorem

List homomorphisms are an expressive computation pattern for scalable divide-and-conquer parallel programs on lists.

Definition 2.2 (list homomorphism (Bird 1987))**.** Function $h :: [A] \to B$ is said to be a *list homomorphism* if there exists function $\phi :: A \to B$ and associative operator $(\odot) :: B \to B \to B$ such that

$$\begin{array}{lcl}
h\ [] & = & \iota_\odot \\
h\ [a] & = & \phi\ a \\
h\ (x \mathbin{+\!\!+} y) & = & h\ x \odot h\ y
\end{array}$$

hold, where $\iota_\odot$ is the unit of $\odot$. Here, we write $h = \mathit{hom}\ (\odot)\ \phi$.[2] □

It is worth noting that associative operator $\odot$ characterizes a list homomorphism. The associativity of $\odot$ guarantees that the result of computation is not affected by where to divide the list. List homomorphisms are useful for developing parallel programs on lists (Bird 1987; Cole 1994, 1995; Gibbons 1996; Hu et al. 1997a).

The third homomorphism theorem demonstrates a necessary and sufficient condition of existence of a list homomorphism.

Theorem 2.3 (the third homomorphism theorem (Gibbons 1996))**.** Function h is a list homomorphism if and only if there exist two operators $\oplus$ and $\otimes$ such that the following equations hold.

$$\begin{array}{lcl}
h\ ([a] \mathbin{+\!\!+} x) & = & a \oplus h\ x \\
h\ (x \mathbin{+\!\!+} [a]) & = & h\ x \otimes a
\end{array} \qquad \square$$

The third homomorphism theorem states that if we can compute a function in both leftward and rightward manners, then there exists a divide-and-conquer parallel algorithm to evaluate the function. What the theorem indicates is not only the existence of parallel programs but also a way of systematically developing parallel programs. The following lemma plays a central role in parallelization.

Lemma 2.4 (Gibbons (1996); Morita et al. (2007))**.** Assume that the following equations hold for function h.

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

Then, $h = \mathit{hom}\ (\odot)\ \phi$ holds, where $\odot$ and ϕ are defined as follows.

$$\begin{array}{l}
\phi\ a = h\ [a] \\
a \odot b = h\ (h^\circ\ a \mathbin{+\!\!+} h^\circ\ b)
\end{array} \qquad \square$$

Lemma 2.4 states that we can derive a parallel program from two sequential programs through a right inverse of h.

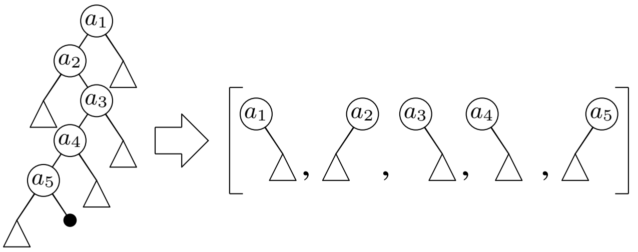

Figure 3. A zipper structure, which expresses a path from the root to the black leaf

3. "The Third Homomorphism Theorem" on Binary Trees

Here, let us consider node-valued binary trees.

$$\begin{array}{lcl}
\mathbf{data}\ \mathit{Tree} & = & \mathit{Node}\ \mathit{Int}\ \mathit{Tree}\ \mathit{Tree} \\
 & | & \mathit{Leaf}
\end{array}$$

The goal is to formalize and prove "the third homomorphism theorem" on node-valued binary trees, which consists of the three notions of downward computations, upward computations, and scalable divide-and-conquer computations on trees. To formalize these notions, we will focus on paths. Given a path, the downward and upward computations are computations that are accomplished while walking the path in downward and upward manners, respectively. Scalable divide-and-conquer computations on trees are computations that split a path in the middle and compute each substructure in parallel. To specify a path on a tree, we will borrow *Huet's zippers* (Huet 1997).

3.1 Zippers on Binary Trees

A zipper is a list whose elements are contexts that are left after a walk. Based on walking downward from the root of a tree, we construct a zipper as follows: when we go down-right from a node, we add its left child to the zipper; when we go down-left, we add the right child to the zipper. For example, Figure 3 shows the correspondence between a zipper and a walk from the root to the black leaf.

The zipper structures for node-valued binary trees can be specified in the following type, where components of the Either type correspond to the left and the right child.

$$\mathbf{type}\ \mathit{Zipper} = [\mathit{Either}\ (\mathit{Int}, \mathit{Tree})\ (\mathit{Int}, \mathit{Tree})]$$

In this paper, a walk usually ends at a leaf. Then, a zipper stores the whole tree. Function $\mathit{z2t}$ below restores a tree from a zipper.

$$\begin{array}{lcl}
\mathit{z2t}\ [] & = & \mathit{Leaf} \\
\mathit{z2t}\ ([\mathsf{L}\ (n,l)] \mathbin{+\!\!+} r) & = & \mathit{Node}\ n\ l\ (\mathit{z2t}\ r) \\
\mathit{z2t}\ ([\mathsf{R}\ (n,r)] \mathbin{+\!\!+} l) & = & \mathit{Node}\ n\ (\mathit{z2t}\ l)\ r
\end{array}$$

When a walk ends at a leaf, a zipper corresponds to a tree. When a walk ends at an internal node, a zipper corresponds not to a tree but to a one-hole context of a tree. Look at Figure 3 again. On one hand, the zipper represents the tree with its path from the root to the black leaf. On the other hand, the zipper also represents the one-hole context in which the black circle represents the hole. We will formalize scalable divide-and-conquer parallel programs on trees based on the second viewpoint, viewing a zipper as a one-hole context. However, we usually regard a zipper as a tree and call the hole the *terminal leaf* because a walk usually ends at a leaf.

We would like to summarize the correspondences among a zipper, a tree (or a one-hole context), and a path. A zipper corresponds to a tree with a terminal leaf (a one-hole context) or a path from the root to the terminal leaf. An initial segment of a zipper corresponds to a one-hole context containing the root or a path from the root to

[2] We usually neglect $\iota_\odot$ because parallel computations on empty lists is useless. If $\iota_\odot$ is necessary, we can prepare it by introducing a special value that behaves as the unit.

a node. A tail segment of a zipper corresponds to a subtree containing the terminal leaf (a one-hole context whose hole is the same as the hole of the original one) or a path from a node to the terminal leaf.

3.2 Downward and Upward Computations on Binary Trees

Next, let us formalize downward and upward computations.

Consider function $\mathit{sumTree}$ below as an example, which sums up all values in a tree.

$$\begin{array}{lcl} \mathit{sumTree}\ \mathit{Leaf} & = & 0 \\ \mathit{sumTree}\ (\mathit{Node}\ n\ l\ r) & = & n + \mathit{sumTree}\ l + \mathit{sumTree}\ r \end{array}$$

First, we would like to give its downward version. Since an initial segment of a zipper corresponds to a path from the root to an internal node, function $\mathit{sumTree}_{\downarrow}$ below performs its computation downward from the root to the terminal leaf.

$$\begin{array}{lcl} \mathit{sumTree}_{\downarrow}\ [] & = & 0 \\ \mathit{sumTree}_{\downarrow}\ (x +\!\!+ [\mathsf{L}\ (n,l)]) & = & \mathit{sumTree}_{\downarrow}\ x + n + \mathit{sumTree}\ l \\ \mathit{sumTree}_{\downarrow}\ (x +\!\!+ [\mathsf{R}\ (n,r)]) & = & \mathit{sumTree}_{\downarrow}\ x + n + \mathit{sumTree}\ r \end{array}$$

Note that for computing summations for trees in the zipper, we use the function $\mathit{sumTree}$.

Similarly, we can give its upward version. Since a tail segment of a zipper corresponds to a path from an internal node to the terminal leaf, function $\mathit{sumTree}_{\uparrow}$ below traverses a tree from its terminal leaf to its root.

$$\begin{array}{lcl} \mathit{sumTree}_{\uparrow}\ [] & = & 0 \\ \mathit{sumTree}_{\uparrow}\ ([\mathsf{L}\ (n,l)] +\!\!+ r) & = & n + \mathit{sumTree}\ l + \mathit{sumTree}_{\uparrow}\ r \\ \mathit{sumTree}_{\uparrow}\ ([\mathsf{R}\ (n,r)] +\!\!+ l) & = & n + \mathit{sumTree}_{\uparrow}\ l + \mathit{sumTree}\ r \end{array}$$

Here, $\mathit{sumTree}_{\uparrow}$ and $\mathit{sumTree}_{\downarrow}$ are equivalent to $\mathit{sumTree}$ in the sense that $\mathit{sumTree}_{\uparrow} = \mathit{sumTree}_{\downarrow} = \mathit{sumTree} \circ \mathit{z2t}$ holds. However, computations on a path may require more information than those on trees. To formalize the correspondence between computations on paths and those on trees, we introduce a notion of *path-based computations*.

Definition 3.1 (path-based computation on binary trees)**.** Function $h' :: \mathit{Zipper} \to B$ is said to be a path-based computation of $h :: \mathit{Tree} \to A$ if there exists function $\psi :: B \to A$ such that the following equation holds.

$$\psi \circ h' = h \circ \mathit{z2t} \qquad \square$$

This equation means that h' simulates the computation of h and the result is extracted by ψ. Note that $\mathit{z2t}$ is a path-based computation of any function; however, this is useless in practice because no significant computations are managed on paths. In other words, it is important to specify an appropriate path-based computation.

Now, let us introduce downward and upward computations.

Definition 3.2 (downward computations on binary trees)**.** Function $h' :: \mathit{Zipper} \to B$, which is a path-based computation of $h :: \mathit{Tree} \to A$, is said to be *downward* if there exists operator $(\otimes) :: B \to \mathit{Either}\ (\mathit{Int}, A)\ (\mathit{Int}, A) \to B$ such that the following equations hold.

$$\begin{array}{lcl} h'\ (x +\!\!+ [\mathsf{L}\ (n,t)]) & = & h'\ x \otimes \mathsf{L}\ (n, h\ t) \\ h'\ (x +\!\!+ [\mathsf{R}\ (n,t)]) & = & h'\ x \otimes \mathsf{R}\ (n, h\ t) \end{array} \qquad \square$$

Definition 3.3 (upward computations on binary trees)**.** Function $h' :: \mathit{Zipper} \to B$, which is a path-based computation of $h :: \mathit{Tree} \to A$, is said to be *upward* if there exists operator $(\oplus) :: \mathit{Either}\ (\mathit{Int}, A)\ (\mathit{Int}, A) \to B \to B$ such that the following equations hold.

$$\begin{array}{lcl} h'\ ([\mathsf{L}\ (n,t)] +\!\!+ x) & = & \mathsf{L}\ (n, h\ t) \oplus h'\ x \\ h'\ ([\mathsf{R}\ (n,t)] +\!\!+ x) & = & \mathsf{R}\ (n, h\ t) \oplus h'\ x \end{array} \qquad \square$$

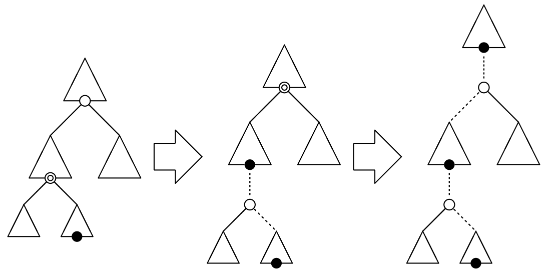

Figure 4. Recursive division on one-hole context: at each step, the one-hole context is divided at the node represented by a concentric circle.

3.3 Parallel Computations on Binary Trees

Next, let us consider scalable divide-and-conquer parallel computations on trees. For scalable divide-and-conquer computations, we would like to divide a tree at an arbitrary place and compute each part in parallel. However, as seen in Figure 1, splitting a tree does not yield two trees of the original type: one is a tree of the original type, but the other is a one-hole context. Therefore, it is difficult to formalize recursive division on trees.

Our idea is to consider recursive division on one-hole contexts, instead of that on trees. We divide a path from the root to the hole, as Figure 4. At each step, we select a node on a path from the root to the hole, divide the one-hole context into three one-hole contexts, i.e., the upper part, the lower part, and a complete subtree with the node. Apparently, we can divide the upper part and the lower part once again; we can divide the complete subtree by taking off an arbitrary leaf and obtaining a one-hole context. Then, we can accomplish recursive division on a one-hole context.

Now, let us characterize scalable divide-and-conquer parallel computations on node-valued binary trees. For parallel computation on trees, we require three operations: an operation (say $\odot$) that merges the results of two one-hole contexts, an operation (say ϕ) that takes the result of a complete tree and yields the result of the complete tree with its parent, and an operation (say ψ) that computes a result of a complete tree from the result of a one-hole context. Since a one-hole context corresponds to a zipper, computation on a one-hole context can be specified by path-based computations.

Definition 3.4 (decomposition on binary trees)**.** A *decomposition* of function $h :: \mathit{Tree} \to A$ is triple $(\phi, \odot, \psi)$ that consists of associative operator $\odot :: B \to B \to B$ and two functions $\phi :: \mathit{Either}\ (\mathit{Int}, A)\ (\mathit{Int}, A) \to B$ and $\psi :: B \to A$ such that

$$\begin{array}{lcl} \multicolumn{3}{l}{\psi \circ h' = h \circ \mathit{z2t}} \\ h'\ [] & = & \iota_{\odot} \\ h'\ [\mathsf{L}\ (n,t)] & = & \phi\ (\mathsf{L}\ (n, h\ t)) \\ h'\ [\mathsf{R}\ (n,t)] & = & \phi\ (\mathsf{R}\ (n, h\ t)) \\ h'\ (x +\!\!+ y) & = & h'\ x \odot h'\ y \end{array}$$

hold, where $\iota_{\odot}$ is the unit of $\odot$. In this case, h is said to be *decomposable*. □

It is worth noting that the associativity of $\odot$ is necessary to guarantee that the result of computation is not affected by where to divide the tree. It is also worth noting that a decomposition can be seen as a list homomorphism on zippers: Define function ϕ' as $\phi'\ (\mathsf{L}\ (n,t)) = \phi\ (\mathsf{L}\ (n, h\ t))$ and $\phi'\ (\mathsf{R}\ (n,t)) = \phi\ (\mathsf{R}\ (n, h\ t))$; then, $h' = \mathit{hom}\ (\odot)\ \phi'$ holds.

Actually decomposable functions can be efficiently evaluated in parallel, which is a consequence of theories of *parallel tree contraction* (Miller and Reif 1985; Cole and Vishkin 1988; Gibbons and Rytter 1989; Abrahamson et al. 1989; Reif 1993).

Theorem 3.5. If $(\phi, \odot, \psi)$ is a decomposition of function h and all of ϕ, $\odot$, and ψ are constant-time computations, then h can be evaluated for a tree of n nodes in $\mathrm{O}(n/p + \log p)$ time on an exclusive-read/exclusive-write parallel random access machine with p processors.

Proof. It is not difficult to confirm that h satisfies the premise of Theorem 3.1 in (Abrahamson et al. 1989). □

Complexity implies good scalability of decomposable functions, because we can obtain linear speedup up to $\mathrm{O}(n/\log n)$ processors.

The function $\mathit{sumTree}$ is decomposable as the following equations show.

$$\begin{array}{l} \mathit{sumPara} = \mathit{sumTree} \circ \mathit{z2t} \\ \quad \begin{array}{lcl} \mathit{sumPara}\ [] & = & 0 \\ \mathit{sumPara}\ [\mathsf{L}\ (n, l)] & = & n + \mathit{sumTree}\ l \\ \mathit{sumPara}\ [\mathsf{R}\ (n, r)] & = & n + \mathit{sumTree}\ r \\ \mathit{sumPara}\ (x +\!\!+\ y) & = & \mathit{sumPara}\ x + \mathit{sumPara}\ y \end{array} \end{array}$$

In other words, $\mathit{sumTree}$ has a decomposition $(\phi, +, \mathit{id})$, where ϕ is defined by $\phi\ (\mathsf{L}\ (n, v)) = n + v$ and $\phi\ (\mathsf{R}\ (n, v)) = n + v$. It is not difficult to derive a parallel program for $\mathit{sumTree}$ because of the associativity of $+$. However, it is generally difficult to derive an associative operator that provides a decomposition.

3.4 "The Third Homomorphism Theorem" on Binary Trees

Finally, let us introduce "the third homomorphism theorem" on node-valued binary trees, which demonstrates a necessary and sufficient condition of existence of a decomposition.

Theorem 3.6 ("the third homomorphism theorem" on binary trees)**.** Function h is decomposable if and only if there exists a path-based computation of h that is both downward and upward.

Proof. We will prove a stronger theorem later (Theorem 6.7). □

The statement of Theorem 3.6 is similar to the third homomorphism theorem on lists. The observation underlying the theorem is that associative operators provide parallel programs not only on lists but also on trees. In addition, the following lemma is useful for developing parallel programs.

Lemma 3.7. Assume that function h', which is a path-based computation of h so that $\psi \circ h' = h \circ \mathit{z2t}$, is both downward and upward; then, there exists a decomposition $(\phi, \odot, \psi)$ of h such that the following equations hold.

$$\begin{array}{lcl} \phi\ (\mathsf{L}\ (v, h\ t)) & = & h'\ [\mathsf{L}\ (v, t)] \\ \phi\ (\mathsf{R}\ (v, h\ t)) & = & h'\ [\mathsf{R}\ (v, t)] \\ a \odot b & = & h'\ (h'^{\circ}\ a +\!\!+\ h'^{\circ}\ b) \end{array}$$

Proof. We will prove a stronger lemma later (Lemma 6.6). □

Combining Theorem 3.6 and Lemma 3.7, we can derive decompositions of functions. Recall $\mathit{sumTree}$. A path-based computation of $\mathit{sumTree}$ (say st) is both downward and upward, because of $\mathit{st} = \mathit{sumTree}_{\downarrow} = \mathit{sumTree}_{\uparrow}$. Therefore, Theorem 3.6 proves that there is a decomposition of $\mathit{sumTree}$. Lemma 3.7 shows a way of obtaining a decomposition $(\phi, \odot, \psi)$. Here, $\psi = \mathit{id}$, because $\mathit{sumTree} \circ \mathit{z2t} = \mathit{id} \circ \mathit{st}$ holds. Obtaining function ϕ is easy as the following calculation shows, where C is either L or R.

$$\begin{array}{lcl} \phi\ (C\ (n, \mathit{sumTree}\ t)) & = & \{\ \text{Lemma 3.7}\ \} \\ & & \mathit{st}\ [C\ (n, t)] \\ & = & \{\ \text{definition of } \mathit{st}\ \} \\ & & n + \mathit{sumTree}\ t \end{array}$$

Thus $\phi\ (C\ (n, r)) = n + r$. The last is associative operator $\odot$. Lemma 3.7 states that a right inverse of st enables us to derive the operator. It is not difficult to find a right inverse.

$$\mathit{st}^{\circ}\ s = [\mathsf{L}\ (s, \mathit{Leaf})]$$

Function st° above is certainly a right inverse of st, because $\mathit{st}\ (\mathit{st}^{\circ}\ s) = \mathit{st}\ [\mathsf{L}\ (s, \mathit{Leaf})] = s$ holds. Now we can obtain a definition of $\odot$ as follows.

$$\begin{array}{lcl} a \odot b & = & \{\ \text{Lemma 3.7}\ \} \\ & & \mathit{st}\ (\mathit{st}^{\circ}\ a +\!\!+\ \mathit{st}^{\circ}\ b) \\ & = & \{\ \text{definition of } \mathit{st}^{\circ}\ \} \\ & & \mathit{st}\ [\mathsf{L}\ (a, \mathit{Leaf}), \mathsf{L}\ (b, \mathit{Leaf})] \\ & = & \{\ \text{definition of } \mathit{st}\ \} \\ & & a + b \end{array}$$

We have obtained a decomposition of $\mathit{sumTree}$, which is exactly the same as the one we previously showed.

4. Examples

In this section, we demonstrate how to develop scalable divide-and-conquer parallel programs. Our development consists of two steps. First, we seek an appropriate path-based computation that is both downward and upward. After that, we obtain a decomposition that brings scalable parallelism.

4.1 Maximum Path Weight

Let us consider a small optimization problem as an initial example to compute the maximum weight of paths from the root to a leaf. For simplicity, we will assume that the value of each node is non-negative. The following sequential program solves the problem.

$$\begin{array}{lcl} \mathit{maxPath}\ \mathit{Leaf} & = & 0 \\ \mathit{maxPath}\ (\mathit{Node}\ n\ l\ r) & = & n + \mathit{max}\ (\mathit{maxPath}\ l)\ (\mathit{maxPath}\ r) \end{array}$$

Our objective here is to develop a parallel program to solve the problem. First, we try to obtain a downward definition and think of the following program.

$$\begin{array}{l} \mathit{maxPath}_{\downarrow}\ [] \\ \quad = 0 \\ \mathit{maxPath}_{\downarrow}\ (x +\!\!+\ [\mathsf{L}\ (n, l)]) \\ \quad = \mathit{max}\ (\mathit{maxPath}_{\downarrow}\ x)\ (\mathit{pathWeight}\ x + n + \mathit{maxPath}\ l) \\ \mathit{maxPath}_{\downarrow}\ (x +\!\!+\ [\mathsf{R}\ (n, r)]) \\ \quad = \mathit{max}\ (\mathit{maxPath}_{\downarrow}\ x)\ (\mathit{pathWeight}\ x + n + \mathit{maxPath}\ r) \end{array}$$

$$\begin{array}{lcl} \mathit{pathWeight}\ [] & = & 0 \\ \mathit{pathWeight}\ (x +\!\!+\ [\mathsf{L}\ (n, l)]) & = & \mathit{pathWeight}\ x + n \\ \mathit{pathWeight}\ (x +\!\!+\ [\mathsf{R}\ (n, r)]) & = & \mathit{pathWeight}\ x + n \end{array}$$

Note that $\mathit{maxPath}_{\downarrow}$ is not downward, because it uses auxiliary function $\mathit{pathWeight}$ that computes the weight of the path from the root to the terminal leaf. The tupling transformation (Fokkinga 1989; Chin 1993; Hu et al. 1997b) is helpful in dealing with such situations. Consider function $\mathit{maxPath}'_{\downarrow}$, which computes the weight of the path to the terminal leaf together with the maximum path weight of the tree, namely $\mathit{maxPath}'_{\downarrow}\ x = (\mathit{maxPath}_{\downarrow}\ x, \mathit{pathWeight}\ x)$. Apparently $\mathit{maxPath}'_{\downarrow}$ is a path-based computation of $\mathit{maxPath}$; in addition, it is downward.

$$\begin{array}{l} \mathit{maxPath}'_{\downarrow}\ []\ =\ (0, 0) \\ \mathit{maxPath}'_{\downarrow}\ (x +\!\!+\ [\mathsf{L}\ (n, l)]) \\ \qquad\qquad = \textbf{let}\ (m, w) = \mathit{maxPath}'_{\downarrow}\ x \\ \qquad\qquad\quad\ \textbf{in}\ (\mathit{max}\ m\ (w + n + \mathit{maxPath}\ l), w + n) \\ \mathit{maxPath}'_{\downarrow}\ (x +\!\!+\ [\mathsf{R}\ (n, r)]) \\ \qquad\qquad = \textbf{let}\ (m, w) = \mathit{maxPath}'_{\downarrow}\ x \\ \qquad\qquad\quad\ \textbf{in}\ (\mathit{max}\ m\ (w + n + \mathit{maxPath}\ r), w + n) \end{array}$$

Therefore, $\mathit{maxPath}'_{\downarrow}$ seems an appropriate path-based computation for $\mathit{maxPath}$, and we would like to derive its upward defini-

$$\mathit{mp}^\circ\ (m,w) =$$

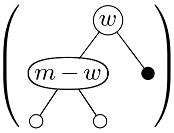

Figure 5. Outline of mp°: closed circle represents terminal leaf.

tion. Function $\mathit{maxPath}'_\uparrow$ below is the upward one.

$$\begin{array}{l}
\mathit{maxPath}'_\uparrow\ [\,] \ = \ (0,0) \\
\mathit{maxPath}'_\uparrow\ ([\mathsf{L}\ (n,l)] +\!\!+\ x) \\
\quad = \ \mathbf{let}\ (m,w) = \mathit{maxPath}'_\uparrow\ x \\
\quad\ \ \mathbf{in}\ (n + \mathit{max}\ m\ (\mathit{maxPath}\ l), n + w) \\
\mathit{maxPath}'_\uparrow\ ([\mathsf{R}\ (n,r)] +\!\!+\ x) \\
\quad = \ \mathbf{let}\ (m,w) = \mathit{maxPath}'_\uparrow\ x \\
\quad\ \ \mathbf{in}\ (n + \mathit{max}\ m\ (\mathit{maxPath}\ r), n + w)
\end{array}$$

Now that we have confirmed that $\mathit{maxPath}'_\downarrow = \mathit{maxPath}'_\uparrow$ (say mp) is both downward and upward, Theorem 3.6 proves the existence of its parallel program. We derive this based on Lemma 3.7.

Obtaining ϕ is straightforward: $\phi\ (C\ (n,m)) = (n+m, n)$, where C is either L or R. To obtain associative operator $\odot$, we would like to find a right inverse of mp. This is not difficult, and function mp° below is a right inverse of mp, which is outlined in Figure 5.

$$\mathit{mp}^\circ\ (m,w) = [\mathsf{L}\ (w, \mathit{Node}\ (m-w)\ \mathit{Leaf}\ \mathit{Leaf})]$$

Note that $\mathit{mp}\ t = (m,w)$ implies $m \geq w$. Therefore, mp° is certainly a right inverse of mp, because given tree $[\mathsf{L}\ (w, \mathit{Node}\ (m - w)\ \mathit{Leaf}\ \mathit{Leaf})]$, where there exists tree t such that $\mathit{mp}\ t = (m,w)$, the maximum path weight of the tree is m and the path weight to the terminal leaf is w. Then, $\odot$ is derived as follows.

$$\begin{array}{l}
(m_1,w_1) \odot (m_2,w_2) \\
\quad = \ \{\ \text{Lemma 3.7}\ \} \\
\quad\ \ \mathit{mp}\ (\mathit{mp}^\circ\ (m_1,w_1) +\!\!+\ \mathit{mp}^\circ\ (m_2,w_2)) \\
\quad = \ \{\ \text{Definition of } \mathit{mp}^\circ\ \} \\
\quad\ \ \mathit{mp}\ [\mathsf{L}\ (w_1, \mathit{Node}\ (m_1 - w_1)\ \mathit{Leaf}\ \mathit{Leaf}), \\
\quad\ \ \phantom{\mathit{mp}\ [}\mathsf{L}\ (w_2, \mathit{Node}\ (m_2 - w_2)\ \mathit{Leaf}\ \mathit{Leaf})] \\
\quad = \ \{\ \text{Definition of } \mathit{mp}\ \} \\
\quad\ \ (\mathit{max}\ m_1\ (w_1 + m_2), w_1 + w_2)
\end{array}$$

Lemma 3.7 guarantees the associativity of operator $\odot$. In summary, we obtain the following parallel program.

$$\begin{array}{lcl}
\multicolumn{3}{l}{\mathit{maxPath} \circ \mathit{z2t} = \mathit{fst} \circ \mathit{mp}} \\
\ \mathit{mp}\ [\,] & = & (0,0) \\
\ \mathit{mp}\ [\mathsf{L}\ (n,t)] & = & (n + \mathit{maxPath}\ t, n) \\
\ \mathit{mp}\ [\mathsf{R}\ (n,t)] & = & (n + \mathit{maxPath}\ t, n) \\
\ \mathit{mp}\ (z_1 +\!\!+\ z_2) & = & \mathbf{let}\ (m_1,w_1) = \mathit{mp}\ z_1 \\
 & & \phantom{\mathbf{let}}\ (m_2,w_2) = \mathit{mp}\ z_2 \\
 & & \mathbf{in}\ (\mathit{max}\ m_1\ (w_1 + m_2), w_1 + w_2)
\end{array}$$

The parallel program we obtained above computes two values, while the sequential program only computes one value; thus, the parallel program is about two times slower than the sequential one on single-processor machines. Since the parallel program is scalable, it will run faster than the sequential one if several processors are available; besides, we can reduce the overhead by using the sequential program, as reported in (Matsuzaki and Hu 2006).

4.2 Leftmost Odd Number

The next example is a small query to find the leftmost odd number in a tree. In fact, this problem can be solved by flattening the tree into a list and considering a divide-and-conquer algorithm on the list. Here we will derive a parallel program without such clever observation.

Function $\mathit{leftOdd}$ below returns the the leftmost odd number in the input tree if one exists; otherwise, it returns special value $*$ that stands for emptiness.

$$\begin{array}{lcl}
\mathit{leftOdd}\ \mathit{Leaf} & = & * \\
\mathit{leftOdd}\ (\mathit{Node}\ n\ l\ r) & = & \mathbf{case}\ \mathit{leftOdd}\ l\ \mathbf{of} \\
 & & \quad * \rightarrow \mathbf{if}\ \mathit{odd}\ n\ \mathbf{then}\ n \\
 & & \quad\ \ \mathbf{else}\ \mathit{leftOdd}\ r \\
 & & \quad v \rightarrow v
\end{array}$$

In the downward computation of $\mathit{leftOdd}$, we need to determine whether a nearly-leaf odd number is leftmost or not. For this purpose, we add additional information to the result: $\mathsf{L}\ v$ and $\mathsf{R}\ v$ correspond to an odd number v at the left and the right of the terminal leaf.

$$\begin{array}{l}
\mathit{leftOdd}_\downarrow\ [\,] \ = \ * \\
\mathit{leftOdd}_\downarrow\ (x +\!\!+\ [\mathsf{L}\ (n,l)]) \\
\qquad = \ \mathbf{case}\ \mathit{leftOdd}_\downarrow\ x\ \mathbf{of} \\
\qquad\qquad \mathsf{L}\ v \rightarrow \mathsf{L}\ v \\
\qquad\qquad a \rightarrow \mathbf{case}\ \mathit{leftOdd}\ l\ \mathbf{of} \\
\qquad\qquad\qquad * \rightarrow \mathbf{if}\ \mathit{odd}\ n\ \mathbf{then}\ \mathsf{L}\ n\ \mathbf{else}\ a \\
\qquad\qquad\qquad v \rightarrow \mathsf{L}\ v \\
\mathit{leftOdd}_\downarrow\ (x +\!\!+\ [\mathsf{R}\ (n,r)]) \\
\qquad = \ \mathbf{case}\ \mathit{leftOdd}_\downarrow\ x\ \mathbf{of} \\
\qquad\qquad \mathsf{L}\ v \rightarrow \mathsf{L}\ v \\
\qquad\qquad a \rightarrow \mathbf{if}\ \mathit{odd}\ n\ \mathbf{then}\ \mathsf{R}\ n \\
\qquad\qquad\qquad \mathbf{else\ case}\ \mathit{leftOdd}\ r\ \mathbf{of}\ * \rightarrow a \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad v \rightarrow \mathsf{R}\ v
\end{array}$$

Function $\mathit{leftOdd}_\downarrow$ is a path-based computation of $\mathit{leftOdd}$: define ψ by $\psi\ * = *$ and $\psi\ (C\ v) = v$ where C is either L or R; then $\psi \circ \mathit{leftOdd}_\downarrow = \mathit{leftOdd} \circ \mathit{z2t}$ holds. Next, we would like to derive its upward definition.

$$\begin{array}{l}
\mathit{leftOdd}_\uparrow\ [\,] \ = \ * \\
\mathit{leftOdd}_\uparrow\ ([\mathsf{L}\ (n,l)] +\!\!+\ x) \\
\qquad = \ \mathbf{case}\ \mathit{leftOdd}\ l\ \mathbf{of} \\
\qquad\qquad * \rightarrow \mathbf{if}\ \mathit{odd}\ n\ \mathbf{then}\ \mathsf{L}\ n\ \mathbf{else}\ \mathit{leftOdd}_\uparrow\ x \\
\qquad\qquad v \rightarrow \mathsf{L}\ v \\
\mathit{leftOdd}_\uparrow\ ([\mathsf{R}\ (n,r)] +\!\!+\ x) \\
\qquad = \ \mathbf{case}\ \mathit{leftOdd}_\uparrow\ x\ \mathbf{of} \\
\qquad\qquad * \rightarrow \mathbf{if}\ \mathit{odd}\ n\ \mathbf{then}\ \mathsf{R}\ n \\
\qquad\qquad\qquad \mathbf{else\ case}\ \mathit{leftOdd}\ r\ \mathbf{of}\ * \rightarrow * \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad v \rightarrow \mathsf{R}\ v \\
\qquad\qquad a \rightarrow a
\end{array}$$

Now that $\mathit{leftOdd}_\downarrow = \mathit{leftOdd}_\uparrow$ (say lo) is both downward and upward, Theorem 3.6 proves that it is decomposable.

Let us derive a parallel program. It is easy to find a right inverse of lo, and function lo° below is a right inverse.

$$\begin{array}{lcl}
\mathit{lo}^\circ\ * & = & [\,] \\
\mathit{lo}^\circ\ (\mathsf{L}\ v) & = & [\mathsf{L}\ (v, \mathit{Leaf})] \\
\mathit{lo}^\circ\ (\mathsf{R}\ v) & = & [\mathsf{R}\ (v, \mathit{Leaf})]
\end{array}$$

Then Lemma 3.7 gives a decomposition of $\mathit{leftOdd}$ after a small amount of calculation. We have omitted the calculation, because it is slightly boring though straightforward. The parallel program is shown in Figure 6. The key is distinguishing two kinds of odd number: those that are to the left of the terminal leaf and those that are to the right. Writing downward/upward programs is helpful for noticing such case analyses necessary for parallel computations.

4.3 Height

The final example is computing the height of a tree.

$$\begin{array}{lcl}
\mathit{height}\ \mathit{Leaf} & = & 1 \\
\mathit{height}\ (\mathit{Node}\ \_\ l\ r) & = & 1 + \mathit{max}\ (\mathit{height}\ l)\ (\mathit{height}\ r)
\end{array}$$

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathit{leftOdd} \circ \mathit{z2t} = \psi \circ \mathit{lo}} \\
\quad \psi\ * & = & * \\
\quad \psi\ (\mathsf{L}\ v) & = & v \\
\quad \psi\ (\mathsf{R}\ v) & = & v \\
\quad \mathit{lo}\ [] & = & * \\
\quad \mathit{lo}\ [\mathsf{L}\ (n,t)] & = & \mathbf{case}\ \mathit{leftOdd}\ t\ \mathbf{of} \\
 & & \quad * \rightarrow \mathbf{if}\ \mathit{odd}\ n\ \mathbf{then}\ \mathsf{L}\ n\ \mathbf{else}\ * \\
 & & \quad v \rightarrow \mathsf{L}\ v \\
\quad \mathit{lo}\ [\mathsf{R}\ (n,t)] & = & \mathbf{if}\ \mathit{odd}\ n\ \mathbf{then}\ \mathsf{R}\ n \\
 & & \mathbf{else\ case}\ \mathit{leftOdd}\ t\ \mathbf{of}\ * \rightarrow * \\
 & & \qquad\qquad\qquad\qquad\quad v \rightarrow \mathsf{R}\ v \\
\quad \mathit{lo}\ (z_1 +\!\!+\ z_2) & = & \mathbf{case}\ (\mathit{lo}\ z_1, \mathit{lo}\ z_2)\ \mathbf{of} \\
 & & \quad (\mathsf{L}\ v, \_) \rightarrow \mathsf{L}\ v \\
 & & \quad (\mathsf{R}\ v, *) \rightarrow \mathsf{R}\ v \\
 & & \quad (\_, r) \rightarrow r
\end{array}
$$

Figure 6. Divide-and-conquer parallel program for *leftOdd*

This problem is similar to the maximum-path-weight problem, and we can specify downward and upward definitions in a similar way.

$$
\begin{array}{lll}
\mathit{height}_{\downarrow}\ [] & = & (1,1) \\
\mathit{height}_{\downarrow}\ (x +\!\!+ [\mathsf{L}\ \_\ l]) & = & \mathbf{let}\ (h,d) = \mathit{height}_{\downarrow}\ x \\
 & & \mathbf{in}\ (\mathit{max}\ h\ (d + \mathit{height}\ l), d+1) \\
\mathit{height}_{\downarrow}\ (x +\!\!+ [\mathsf{R}\ \_\ r]) & = & \mathbf{let}\ (h,d) = \mathit{height}_{\downarrow}\ x \\
 & & \mathbf{in}\ (\mathit{max}\ h\ (d + \mathit{height}\ r), d+1) \\
\mathit{height}_{\uparrow}\ [] & = & (1,1) \\
\mathit{height}_{\uparrow}\ ([\mathsf{L}\ \_\ l] +\!\!+ x) & = & \mathbf{let}\ \ (h,d) = \mathit{height}_{\uparrow}\ x \\
 & & \mathbf{in}\ (1 + \mathit{max}\ h\ (\mathit{height}\ l), d+1) \\
\mathit{height}_{\uparrow}\ ([\mathsf{R}\ \_\ r] +\!\!+ x) & = & \mathbf{let}\ \ (h,d) = \mathit{height}_{\uparrow}\ x \\
 & & \mathbf{in}\ (1 + \mathit{max}\ h\ (\mathit{height}\ r), d+1)
\end{array}
$$

Function $\mathit{height}_{\downarrow} = \mathit{height}_{\uparrow}$ (say ht) computes the height of a tree in its first result, and its second result retains the depth of the terminal leaf. Function ht is a path-based computation of height, because $\mathit{fst} \circ \mathit{ht} = \mathit{height} \circ \mathit{z2t}$ holds.

Let us parallelize it. The only nontrivial part is that to obtain an associative operator from its right inverse. In this case, different from the previous examples, we should define right inverse ht° in a recursive manner because $\mathit{ht}^{\circ}\ (h,d)$ yields a tree of height h. Therefore, it first seems difficult to simplify the definition of $\odot$, even though the naive definition $a \odot b = \mathit{ht}\ (\mathit{ht}^{\circ}\ a +\!\!+ \mathit{ht}^{\circ}\ b)$ is inefficient. Actually, that simplification is not too difficult. Let us look at Figure 7, which outlines the tree $\mathit{ht}^{\circ}\ (h_1,d_1) +\!\!+ \mathit{ht}^{\circ}\ (h_2,d_2)$. The left and right trees correspond to $\mathit{ht}^{\circ}\ (h_1,d_1)$ and $\mathit{ht}^{\circ}\ (h_2,d_2)$, and the curved arrow corresponds to the concatenation operation on zippers. Now it is easy to see that the height of this tree is $\mathit{max}\ h_1\ (d_1 + h_2)$ and the depth of the terminal leaf is $d_1 + d_2$. In short, the following gives a definition of $\odot$.

$$(h_1,d_1) \odot (h_2,d_2) = (\mathit{max}\ h_1\ (d_1 + h_2), d_1 + d_2)$$

Then, we obtain the following parallel program for height.

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathit{height} \circ \mathit{z2t} = \mathit{fst} \circ \mathit{ht}} \\
\quad \mathit{ht}\ [] & = & (1,1) \\
\quad \mathit{ht}\ [\mathsf{L}\ (n,t)] & = & (1 + \mathit{height}\ t, 2) \\
\quad \mathit{ht}\ [\mathsf{R}\ (n,t)] & = & (1 + \mathit{height}\ t, 2) \\
\quad \mathit{ht}\ (z_1 +\!\!+\ z_2) & = & \mathbf{let}\ (h_1,d_1) = \mathit{ht}\ z_1 \\
 & & \qquad (h_2,d_2) = \mathit{ht}\ z_2 \\
 & & \mathbf{in}\ (\mathit{max}\ h_1\ (d_1 + h_2), d_1 + d_2)
\end{array}
$$

We have considered how to merge the results of substructures with the abstraction in Figure 7. The most significant thing is that Theorem 3.6 guarantees the correctness of the merging operation obtained from the abstraction. Theorem 3.6 proves that the results of ht, namely the height of the tree and the depth of the terminal leaf, are sufficient for merging the results of two parts; thus, we can derive correct merging operation no matter what shape the trees are we image for ht°.

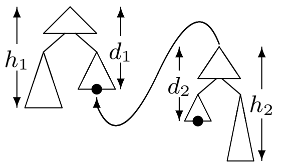

Figure 7. Outline of $\mathit{ht}^{\circ}\ (h_1,d_1) +\!\!+ \mathit{ht}^{\circ}\ (h_2,d_2)$: curved arrowed line denotes plugging operation, which corresponds to concatenation of two zippers.

5. "The Third Homomorphism Theorem" on Polynomial Data Structures

This section discusses our generalization of the method so that it can deal with all *polynomial data structures*, which can capture a large class of algebraic data structures on functional languages.

5.1 Polynomial Functors

To describe our results, we use a few notions from category theory. A category consists of a set of objects and a set of morphisms. A functor (say F) is a morphism of categories, and it respects both identity and composition.

$$
\begin{array}{lll}
\mathsf{F}\mathit{id} & = & \mathit{id} \\
\mathsf{F}(f \circ g) & = & \mathsf{F}f \circ \mathsf{F}g
\end{array}
$$

We will consider the category $\mathbf{Set}$, where an object is a set and morphisms from object A to object B are all total functions from A to B. A functor gives a mapping between functions together with sets.

A functor is said to be polynomial if it is made up from identity functor I, constant functors such as $!A$ where A is an object, and bifunctors $+$ and $\times$. Let two operators for morphisms $\times$ and $+$ be $(f \times g)(x,y) = (f\ x, g\ y)$, $(f+g)(1,x) = (1, f\ x)$, and $(f + g)(2,y) = (2, g\ y)$. Then, the definitions of polynomial functors are given as follows, where F and G are polynomial functors, A and B are objects, and f is a morphism.

$$
\begin{array}{l}
(!A)B = A \\
(!A)f = \mathit{id} \\
\mathsf{I}A = A \\
\mathsf{I}f = f \\
(\mathsf{F} \times \mathsf{G})A = \mathsf{F}A \times \mathsf{G}A \\
(\mathsf{F} \times \mathsf{G})f = \mathsf{F}f \times \mathsf{G}f \\
(\mathsf{F} + \mathsf{G})A = (\{1\} \times \mathsf{F}A) \cup (\{2\} \times \mathsf{G}A) \\
(\mathsf{F} + \mathsf{G})f = \mathsf{F}f + \mathsf{G}f
\end{array}
$$

The least fixed point of functor F, denoted by $\mu\mathsf{F}$, is the smallest set that satisfies $\mathsf{F}(\mu\mathsf{F}) = \mu\mathsf{F}$. The least fixed point exists for each polynomial functor. It is well known that the least fixed points of polynomial functors provide a good characterization of a large class of algebraic data structures (Backhouse et al. 1998), and these are called *polynomial data structures*. For example, lists having elements of type A can be recognized as the least fixed point of functor L below, where 1 denotes a singleton set.

$$\mathsf{L} = !1 + !A \times \mathsf{I}$$

Node-valued trees having values of type A can also be recognized as the least fixed point of functor T below.

$$\mathsf{T} = !1 + !A \times \mathsf{I} \times \mathsf{I}$$

5.2 Zippers for Polynomial Data Structures

Next, let us define zippers for polynomial data structures. We will follow McBride (2001) who showed a systematic derivation of zippers based on derivatives of functors. The derivative of polynomial functor F, denoted by $\partial\mathsf{F}$, is a polynomial functor defined as follows, where $\bullet$ denotes a distinguishable element.

$$\begin{array}{lcl}
\partial(!A) & = & !\emptyset \\
\partial(\mathsf{I}) & = & !\{\bullet\} \\
\partial(\mathsf{F} \times \mathsf{G}) & = & \mathsf{F} \times \partial\mathsf{G} + \partial\mathsf{F} \times \mathsf{G} \\
\partial(\mathsf{F} + \mathsf{G}) & = & \partial\mathsf{F} + \partial\mathsf{G}
\end{array}$$

Functor $\partial\mathsf{F}$ corresponds to a one-hole context structure of F. The zipper structure of $\mu\mathsf{F}$, denoted by Z_F, is recognized as $Z_\mathsf{F} = [\partial\mathsf{F}(\mu\mathsf{F})]$.

To convert zippers to trees, we use an operator $(\triangleleft_\mathsf{F}) :: Z_\mathsf{F} \to \mu\mathsf{F} \to \mu\mathsf{F}$, which is defined by

$$z \triangleleft_\mathsf{F} t = \mathit{foldr}\ (\lessdot_\mathsf{F})\ t\ z$$

where $(\lessdot_\mathsf{F}) :: \partial\mathsf{F}(\mu\mathsf{F}) \to \mu\mathsf{F} \to \mu\mathsf{F}$ is the plugging-in operator (McBride 2001) defined as follows.

$$\begin{array}{lcl}
a \lessdot_{!A} t & = & a \\
\bullet \lessdot_\mathsf{I} t & = & t \\
(1, (a, b)) \lessdot_{\mathsf{F}\times\mathsf{G}} t & = & (a, b \lessdot_\mathsf{G} t) \\
(2, (a, b)) \lessdot_{\mathsf{F}\times\mathsf{G}} t & = & (a \lessdot_\mathsf{F} t, b) \\
(1, a) \lessdot_{\mathsf{F}+\mathsf{G}} t & = & (1, a \lessdot_\mathsf{F} t) \\
(2, b) \lessdot_{\mathsf{F}+\mathsf{G}} t & = & (2, b \lessdot_\mathsf{G} t)
\end{array}$$

We will omit the subscript of $\triangleleft_\mathsf{F}$ when it is apparent from its context.

Note that operator $\triangleleft$ takes an additional tree to convert a zipper to a tree, because a zipper corresponds to a one-hole context. As we want to minimize the differences between zippers and trees, we will force the difference between zippers and trees to be leaves. A set of leaves of $\mu\mathsf{F}$, denoted by $\mathit{leaves}_{\mu\mathsf{F}}$, is formalized as follows.

$$\begin{array}{l}
\mathit{leaves}_{\mu\mathsf{F}} = [\![\mathsf{F}]\!]_{\mathit{leaves}} \\
\quad \begin{array}{lcl}
[\![!A]\!]_{\mathit{leaves}} & = & A \\
[\![\mathsf{I}]\!]_{\mathit{leaves}} & = & \emptyset \\
[\![\mathsf{F} \times \mathsf{G}]\!]_{\mathit{leaves}} & = & [\![\mathsf{F}]\!]_{\mathit{leaves}} \times [\![\mathsf{G}]\!]_{\mathit{leaves}} \\
[\![\mathsf{F} + \mathsf{G}]\!]_{\mathit{leaves}} & = & (\{1\} \times [\![\mathsf{F}]\!]_{\mathit{leaves}}) \cup (\{2\} \times [\![\mathsf{G}]\!]_{\mathit{leaves}})
\end{array}
\end{array}$$

5.3 Sequential and Parallel Computations on Zippers

Now that the notion of zippers has been clarified, it is not difficult to provide definitions for downward and upward computations.

Definition 5.1 (path-based computation)**.** Function $h' :: Z_\mathsf{F} \to B$ is said to be a *path-based computation* of function $h :: \mu\mathsf{F} \to A$ if there exists operator $\ominus :: B \to \mu\mathsf{F} \to A$ such that the following equation holds for all $t \in \mathit{leaves}_{\mu\mathsf{F}}$.

$$h\ (z \triangleleft t) \quad = \quad h'\ z \ominus t \qquad \Box$$

Definition 5.2 (downward computation)**.** Function $h' :: Z_\mathsf{F} \to B$, which is a path-based computation of function $h :: \mu\mathsf{F} \to A$, is said to be *downward* if there exist operator $(\otimes) :: B \to \partial\mathsf{F}A \to B$ and value $e :: B$ such that the following equation holds.

$$h'\ z \quad = \quad \mathit{foldl}\ (\otimes)\ e\ (\mathit{map}\ (\partial\mathsf{F}h)\ z) \qquad \Box$$

Definition 5.3 (upward computation)**.** Function $h' :: Z_\mathsf{F} \to B$, which is a path-based computation of function $h :: \mu\mathsf{F} \to A$, is said to be *upward* if there exist operator $(\oplus) :: \partial\mathsf{F}A \to B \to B$ and value $e :: B$ such that the following equations holds.

$$h'\ z \quad = \quad \mathit{foldr}\ (\oplus)\ e\ (\mathit{map}\ (\partial\mathsf{F}h)\ z) \qquad \Box$$

We characterize scalable divide-and-conquer parallel programs by recursive division on one-hole contexts as the same as the case of node-valued binary trees.

Definition 5.4 (decomposition)**.** A *decomposition* of function $h :: \mu\mathsf{F} \to A$ is tuple $(\phi, \odot, \ominus)$ that consists of function $\phi :: \partial\mathsf{F}A \to B$, associative operator $\odot :: B \to B \to B$, and operator $\ominus :: B \to \mu\mathsf{F} \to A$ such that the following equation holds for all $t \in \mathit{leaves}_{\mu\mathsf{F}}$.

$$h\ (z \triangleleft t) = \mathit{hom}\ (\odot)\ (\phi \circ \partial\mathsf{F}h)\ z \ominus t \qquad \Box$$

One of the most difficult issues is to find a scalable divide-and-conquer parallel algorithm to evaluate such functions. We (Morihata and Matsuzaki 2008) devised an algorithm by generalizing the tree contraction algorithm on binary trees given by Abrahamson et al. (1989). We have not discussed this in detail here, because it is beyond the scope of this paper.

Theorem 5.5. If $(\phi, \odot, \ominus)$ is a decomposition of function h and all of ϕ, $\odot$, and $\ominus$ are constant-time computations, then h can be evaluated for a tree of n nodes in $\mathrm{O}(n/p + \log p)$ time on an exclusive-read/exclusive-write parallel random access machine with p processors.

Proof. It is not difficult to confirm that h satisfies the premise of Theorem 9 in (Morihata and Matsuzaki 2008). □

5.4 "The Third Homomorphism Theorem" on Polynomial Data Structures

We have shown that list homomorphisms characterize scalable divide-and-conquer parallel programs on polynomial data structures. Therefore, we can utilize parallelization methods on lists for free. For instance, "the third homomorphism theorem" on trees is a direct consequence of that on lists.

Lemma 5.6. Assume that function h', which is a path-based computation of h so that $h\ (z \triangleleft t) = h'\ z \ominus t$ holds for all $t \in \mathit{leaves}_{\mu\mathsf{F}}$, is both downward and upward; then, there exists a decomposition $(\phi, \odot, \ominus)$ of h such that $\phi\ (\partial\mathsf{F}h\ a) = h'\ [a]$ and $a \odot b = h'\ (h'^{\circ}\ a +\!\!+\ h'^{\circ}\ b)$ hold. □

Proof. Since h' is both leftward and rightward, $\odot$ is associative and $h' = \mathit{hom}\ (\odot)\ (\phi \circ \partial\mathsf{F}h)$ holds from Lemma 2.4. Therefore, $(\phi, \odot, \ominus)$ forms a decomposition of h. □

Theorem 5.7 ("the third homomorphism theorem" on polynomial data structures)**.** Function $h :: \mu\mathsf{F} \to A$ is decomposable if and only if there exist operators $(\oplus) :: \partial\mathsf{F}A \to B \to B$, $(\otimes) :: B \to \partial\mathsf{F}A \to B$, and $(\ominus) :: B \to \mu\mathsf{F} \to A$, such that the following equations hold for all $t \in \mathit{leaves}_{\mu\mathsf{F}}$.

$$\begin{array}{lcl}
h\ (z \triangleleft t) & = & \mathit{foldr}\ (\oplus)\ e\ (\mathit{map}\ (\partial\mathsf{F}h)\ z) \ominus t \\
& = & \mathit{foldl}\ (\otimes)\ e\ (\mathit{map}\ (\partial\mathsf{F}h)\ z) \ominus t
\end{array}$$

Proof. The "if" part is a direct consequence of Lemma 6.6, and the "only if" part is straightforward. □

6. Discussion

We have developed a method for systematically constructing scalable divide-and-conquer parallel programs on polynomial data structures. We have focused on paths and expressed paths by zippers to utilize the known theories on lists. We have proposed a characterization of scalable divide-and-conquer parallel programs on polynomial data structures, proved "the third homomorphism theorem" on polynomial data structures, and demonstrated the effectiveness of our method with examples.

Our motivation was to derive "the third homomorphism theorem" on trees. The theorem, which was first introduced by Gibbons (1996), is useful for developing parallel programs, and automatic parallelization methods have been proposed (Geser and Gorlatch 1999; Morita et al. 2007) based on the theorem. Our theorem is an exact generalization of the original third homomorphism theorem, and our results are built on list homomorphisms. Therefore, existing automatic parallelization methods should be applicable.

One of the aims of this paper is to explain ideas for constructing scalable parallel programs on trees through the development of our theory. Parallel tree contraction, which was first proposed by Miller and Reif (1985), is known to be a useful framework for developing scalable parallel programs on trees, and many computations have been implemented on it (Cole and Vishkin 1988; Gibbons and Rytter 1989; Abrahamson et al. 1989; Skillicorn 1996; Gibbons et al. 1994; **?**; Matsuzaki 2007); however, parallel tree contraction is hard to use, because it requires a set of operations that satisfy a certain condition. In fact, the requirements for decomposable functions is equivalent to the sufficient condition for parallel tree contraction discussed in (**?**), and "the third homomorphism theorem" on trees brings sets of operations that satisfy the condition. Moreover, our results can deal with polynomial structures, even though most of the existing studies have only considered binary trees.

The third homomorphism theorem requires two sequential programs, while conventional parallelization methods generate a parallel program from a sequential program. Even though this requirement may have its shortcomings, it is arguably true. There is generally little hope of obtaining a parallel program from a sequential program, because parallel programs are more complicated than sequential ones. In other words, extra information is necessary to develop parallel programs from sequential ones. What the third homomorphism theorem provides is a systematic way of revealing such extra information.

We have developed our methods on polynomial data structures, which correspond to trees of bounded degree. Regular data structures are a generalization of polynomial data structures, and include trees of unbounded degree. Since McBride (2001) demonstrated a systematic derivation of zipper structures for regular data structures, there are no problems with formalizing path-based computations, downward computations, or upward computations on regular data structures. However, it is difficult to achieve scalability on regular data structures. We have required the operation to merge the results of siblings in $\mathrm{O}(1)$ time. As this requirement is not realistic on regular data structures, it is necessary to parallelize this merging operation. In summary, it would be interesting to investigate "the third homomorphism theorem" on regular data structures.

Acknowledgments

The authors are grateful to Nao Hirokawa, who proposed using Huet's zippers for formalizing computations on paths. The authors are also grateful to anonymous referees of POPL 2009 for their helpful comments. The first author is supported by Grant-in-Aid for JSPS research fellows $20 \cdot 2411$.

References

Karl R. Abrahamson, N. Dadoun, David G. Kirkpatrick, and Teresa M. Przytycka. A simple parallel tree contraction algorithm. *Journal of Algorithms*, 10(2):287–302, 1989.

Roland Carl Backhouse, Patrik Jansson, Johan Jeuring, and Lambert G. L. T. Meertens. Generic programming: An introduction. In *Advanced Functional Programming*, pages 28–115, 1998.

Richard S. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, pages 3–42. Springer-Verlag, 1987. NATO ASI Series F Volume 36. Also available as Technical Monograph PRG-56, from the Programming Research Group, Oxford University.

Wei-Ngan Chin. Towards an automated tupling strategy. In *Proceedings of the ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation, PEPM'93*, pages 119–132, New York, NY, USA, 1993. ACM Press.

Murray Cole. Parallel programming, list homomorphisms and the maximum segment sum problem. In Gerhard R. Joubert, Denis Trystram, Frans J. Peters, and David J. Evans, editors, *Parallel Computing: Trends and Applications, PARCO 1993, Grenoble, France*, pages 489–492. Elsevier, 1994.

Murray Cole. Parallel programming with list homomorphisms. *Parallel Processing Letters*, 5:191–203, 1995.

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

Maarten M. Fokkinga. Tupling and mutumorphisms. In *Squiggolist*, volume 1(4), 1989.

Alfons Geser and Sergei Gorlatch. Parallelizing functional programs by generalization. *Journal of Functional Programming*, 9(6):649–673, 1999.

Alan Gibbons and Wojciech Rytter. Optimal parallel algorithm for dynamic expression evaluation and context-free recognition. *Information and Computation*, 81(1):32–45, 1989.

Jeremy Gibbons. The third homomorphism theorem. *Journal of Functional Programming*, 6(4):657–665, 1996.

Jeremy Gibbons, Wentong Cai, and David B. Skillicorn. Efficient parallel algorithms for tree accumulations. *Science of Computer Programming*, 23(1):1–18, 1994.

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

Zhenjiang Hu, Hideya Iwasaki, Masato Takeichi, and Akihiko Takano. Tupling calculation eliminates multiple data traversals. In *Proceedings of the 2nd ACM SIGPLAN International Conference on Functional Programming, ICFP'97, Amsterdam, The Netherlands*, pages 164–175. ACM Press, 1997b.

Gérard P. Huet. The zipper. *Journal of Functional Programming*, 7(5): 549–554, 1997.

Kiminori Matsuzaki. *Parallel Programming with Tree Skeletons*. PhD thesis, Graduate School of Information Science and Technology, The University of Tokyo, 2007.

Kiminori Matsuzaki and Zhenjiang Hu. Implementation of tree skeletons on distributed-memory parallel computers. *Technical Report METR* 2006-65, Department of Mathematical Informatics, University of Tokyo, December 2006.

Conor McBride. The derivative of a regular type is its type of one-hole contexts. Unpublished manuscript, 2001.

Gary L. Miller and John H. Reif. Parallel tree contraction and its application. In *26th Annual Symposium on Foundations of Computer Science, 21-23 October 1985, Portland, Oregon, USA*, pages 478–489. IEEE, 1985.

Akimasa Morihata and Kiminori Matsuzaki. A tree contraction algorithm on non-binary trees. *Technical Report METR* 2008-27, Department of Mathematical Informatics, University of Tokyo, June 2008.

Kazutaka Morita, Akimasa Morihata, Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi. Automatic inversion generates divide-and-conquer parallel programs. In Jeanne Ferrante and Kathryn S. McKinley, editors, *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, pages 146–155, 2007.

Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

John H. Reif, editor. *Synthesis of Parallel Algorithms*. Morgan Kaufmann Publishers, 1993.

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