MATHEMATICAL ENGINEERING
TECHNICAL REPORTS

Domain-Specific Optimization for Skeleton Programs Involving Neighbor Elements

Kento EMOTO, Kiminori MATSUZAKI,
Zhenjiang HU and Masato TAKEICHI

METR 2007–05 February 2007

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

WWW page: http://www.i.u-tokyo.ac.jp/edu/course/mi/index_e.shtml

Domain-Specific Optimization for Skeleton Programs Involving Neighbor Elements

Kento EMOTO, Kiminori MATSUZAKI, Zhenjiang HU and Masato TAKEICHI

Graduate School of Information Science and Technology
The University of Tokyo
{emoto,kmatsu}@ipl.t.u-tokyo.ac.jp
{hu,takeichi}@mist.i.u-tokyo.ac.jp

Abstract. Skeletal parallel programming enables us to develop parallel programs easily by composing ready-made components called *skeletons*. However, a simply-composed skeleton program often lacks efficiency due to overheads of intermediate data structures and communications. Many studies have focused on optimizations by fusing successive skeletons to eliminate the overheads. Existing fusion transformations, however, are too general to achieve adequate efficiency for some classes of problems. Thus, a specific fusion optimization is needed for a specific class. In this paper, we propose a specific optimization of skeleton programs involving neighbor elements, which is often seen in scientific computations. We start with a normal form that abstracts the programs of interest. Then, we develop fusion rules that transform a skeleton program into the normal form. Finally, we make efficient parallel implementation of the normal form.

1 Introduction

Recently, the increasing popularity of parallel machines like PC clusters and multi-core CPUs attracts more and more users. However, development of efficient parallel programs is difficult due to synchronization, interprocessor communications, and data distribution that complicate the parallel programs. This difficulty calls for a methodology of parallel programming with ease, and many studies have addressed themselves to it. As one promising solution, skeletal parallel programming [2, 10] has been proposed.

In skeletal parallel programming users develop parallel programs by composing *skeletons*, which are abstracted basic patterns in parallel programs. Each skeleton is given as a higher order function that takes concrete computations as its parameters, and conceals low-level parallelism from users. Therefore, users can develop parallel programs with the skeletons in a similar way to developing sequential programs.

Efficiency is one of the most important topics in the research of skeletal parallel programming. Since skeleton programs are developed in a compositional style, they often have overheads of many loops and intermediate data. To make skeleton programs efficient, not only each skeleton is implemented efficiently in parallel, but also optimizations over multiple skeletons are necessary.

There have been several studies on the optimizations over multiple skeletons based on *fusion transformations* [5–7, 9, 13], which were studied in depth in the field of functional programming [4, 12]. In particular, general fusion optimizations [5, 7, 9, 13] have achieved good results both in theory and in practice. For example, Hu et al. [7] proposed a set of fusion rules based on a general form of skeletons named accumulate.

Although the general fusion optimizations so far are reasonably powerful, there is still large room for further optimizations. Due to the generality of their fusion transformations, some overheads in skeleton programs are left through the general fusion optimizations. In many cases such overheads can be removed if we provide a program-specific implementation. Thus, some specific optimizations are important for efficiency of skeleton programs.

In this paper, we propose a specific optimization for skeleton programs that involve neighbor elements, which is often seen in scientific computations. The proposed optimization is based on the following strategy.

> First, we formalize a *normal form* that captures the domain-specific computations. Then, we develop *fusion rules* that transform a skeleton program into the normal form. Finally, we provide an efficient *parallel implementation* of the normal form.

We formalized a normal form and fusion rules for the class of skeleton programs and developed a small system for fusing skeleton programs into the normal form implemented efficiently in parallel. The experiment results show effectiveness of the domain-specific optimization.

The rest of this paper is organized as follows. Section 2 defines skeletons and our target skeleton programs in this paper. It also explains a strategy for optimization of our target skeleton programs. Section 3 designs a normal form for the target skeleton programs. Section 4 gives fusion rules for transformation of a skeleton program into the normal form. Section 5 explains parallel implementation of the normal form. Section 6 expands our target programs. Section 7 shows experiment results. Section 8 discusses the applicability of our strategy and related work. Section 9 concludes this paper.

2 Preliminaries

2.1 Notations

Notations in this paper follow that of Haskell [1], a pure functional language.

Function application is denoted by a space and the argument may be written without brackets. Thus $f\ a$ means $f(a)$ in ordinary notation. Functions are curried, i.e. $f\ a\ b$ means $(f\ a)\ b$. The function application binds more strongly than any other operator, so $f\ a \otimes b$ means $(f\ a) \otimes b$, but not $f\ (a \otimes b)$. Function composition is denoted by $\circ$, so $(f \circ g)\ x = f\ (g\ x)$ from its definition. The identity function, which is the identity of function composition, is denoted by id. Binary operators are denoted by $\oplus$, $\otimes$ and so on. Binary operators can be used as functions by sectioning as follows: $a \oplus b = (a \oplus)\ b = (\oplus b)\ a = (\oplus)\ a\ b$. The identity of a binary operator $\oplus$ is denoted by $\imath_{\oplus}$. Pairs are Cartesian products of plural data, written like (x, y). It can be extended to the case of arbitrary number of elements like (x, y, z).

2.2 Lists and Basic Functions

Lists are constructed by three constructors: $[\,]$ makes an empty list, $[\cdot]$ makes a singleton, and $+\!\!+$ concatenates two lists.

$$\begin{aligned} \textbf{data}\ \mathit{List}\ \alpha = {} & \mathit{List}\ \alpha +\!\!+ \mathit{List}\ \alpha \\ \mid {} & [\cdot]\ \alpha \\ \mid {} & [\,] \end{aligned}$$

Here, the concatenation constructor $+\!\!+$ is associative. For simplicity, we denote $\mathit{List}\ \alpha$ by $[\alpha]$. With the above constructors, a list of three elements a_0, a_1 and a_2 is constructed as $[\cdot]\ a_1 +\!\!+ [\cdot]\ a_2 +\!\!+ [\cdot]\ a_3$. For simplicity, we denote a list by lining elements up between '[' and ']' separated by ',' . So, the above list is denoted by $[a_1, a_2, a_3]$. We also use a notation $a : x = [a] +\!\!+ x$ to show the head of a list.

We define basic functions on lists that used in the rest of this paper.

$$
\begin{aligned}
&\text{length } (a : x) = 1 + \text{length } x \\
&\text{length } [\,] \quad\;\; = 0 \\
&\text{init } (x +\!\!+ [a]) = x \\
&\text{init } [\,] \quad\;\; = [\,] \\
&\text{tail } (a : x) = x \\
&\text{tail } [\,] \quad\;\; = [\,] \\
&\text{last } (x +\!\!+ [a]) = a \\
&\text{head } (a : x) = a \\
&\text{rev } (a : x) = \text{rev } x +\!\!+ [a] \\
&\text{rev } [\,] \quad\;\; = [\,] \\
&\text{repeat } a = a : \text{repeat } a \\
&\text{at } i \ (a : x) = \textbf{if } i = 0 \textbf{ then } a \textbf{ else at } (i - 1) \ x
\end{aligned}
$$

Function length calculates the length of a list. Function init removes the last element of a list, while tail removes the head element. Function last returns the last element of a list, while head returns the head element. Function rev reverses a list. Function repeat generates an infinite list of the given element. Function at returns the element at the given index.

2.3 Skeletons on Lists

We introduce skeletons on lists. Formal definitions of those skeletons are shown in Figure 1. In the following, we explain skeletons using intuitive definitions and examples.

Skeleton map applies a given function f to each element of the list.

$$\mathsf{map}\ f\ [a_1, \ldots, a_n] = [f\ a_1, \ldots, f\ a_n]$$

For example, computation that calculates squares of each element is described with map as follows.

$$\mathsf{map}\ sqr\ [1, 2, 3, 4] = [1, 4, 9, 16]$$

Skeleton zip zips two lists of the same length.

$$\mathsf{zip}\ [a_1, \ldots, a_n]\ [b_1, \ldots, b_n] = [(a_1, b_1), \ldots, (a_n, b_n)]$$

Similarly, zip_3 zips three lists. A composition of map and zip is often denoted by $\mathsf{zipWith}\ f = \mathsf{map}\ (\lambda(x, y) \to f\ x\ y) \circ \mathsf{zip}$.

Skeleton $\mathsf{shift}_{\gg}$ shifts elements of a list to the right by one, and inserts the given value e as the leftmost element. The rightmost element is thrown away. Similarly, skeleton $\mathsf{shift}_{\ll}$ shifts elements to the left and inserts the given value as the rightmost element.

$$
\begin{aligned}
&\mathsf{shift}_{\ll}\ e\ [a_1, \ldots, a_n] = [a_2, \ldots, a_n, e] \\
&\mathsf{shift}_{\gg}\ e\ [a_1, \ldots, a_n] = [e, a_1, \ldots, a_{n-1}]
\end{aligned}
$$

Skeleton reduce calculates a reduction of a list by the given associative binary operator $\oplus$. Application of reduce to an empty list results in the identity $\imath_{\oplus}$ of operator $\oplus$.

$$\mathsf{reduce}\ (\oplus)\ [a_1, \ldots, a_n] = a_1 \oplus \cdots \oplus a_n$$

For example, a summation of the elements are calculated by reduce as follows.

$$\mathsf{reduce}\ (+)\ [1, 2, 3, 4] = 10 \quad (= 1 + 2 + 3 + 4)$$

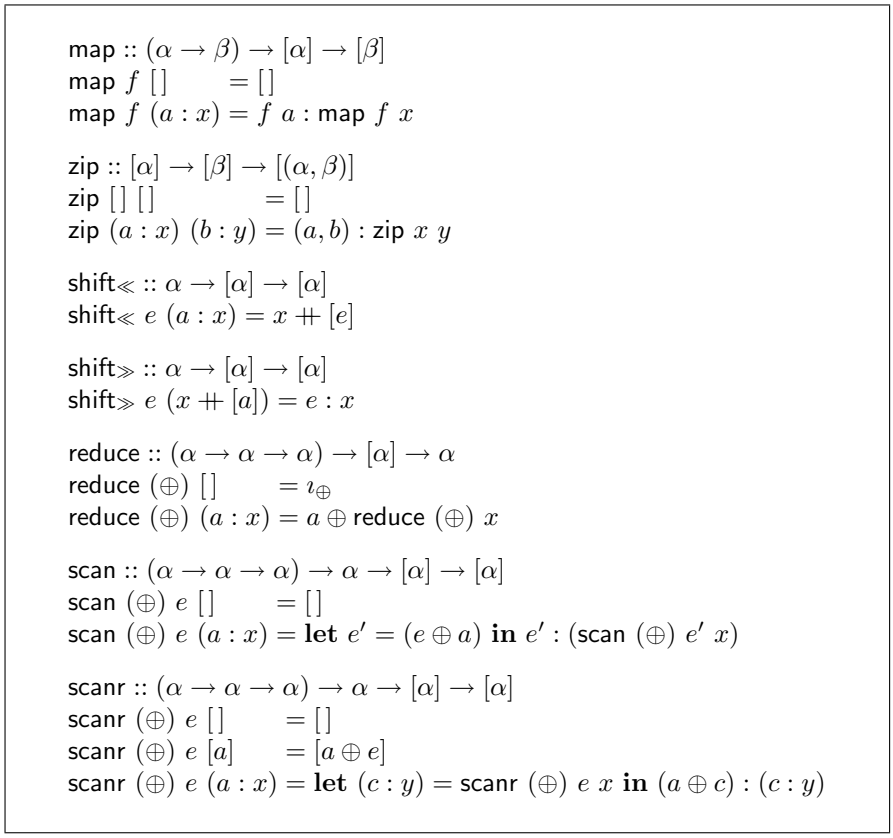


Fig. 1. Definitions of Skeletons on Lists

Skeleton scan accumulate the intermediate results of a reduction.

$$\begin{aligned}\mathsf{scan}\ (\oplus)\ e\ [a_1, \ldots, a_n] = [b_1, \ldots, b_n] \\ \textbf{where}\ b_i = e \oplus a_1 \oplus \cdots \oplus a_i\end{aligned}$$

For example, an accumulation of a summation is described with scan as follows.

$$\mathsf{scan}\ (+)\ 2\ [1, 2, 3, 4] = [3, 5, 8, 12]$$

Skeleton scanr is the reverse of scan.

$$\begin{aligned}\mathsf{scanr}\ (\oplus)\ e\ [a_1, \ldots, a_n] = [c_1, \ldots, c_n] \\ \textbf{where}\ c_i = a_i \oplus \cdots \oplus a_n \oplus e\end{aligned}$$

For example, a reverse accumulation of a summation is described with scanr as follows.

$$\mathsf{scanr}\ (+)\ 2\ [1, 2, 3, 4] = [12, 11, 9, 6]$$

2.4 Target Skeleton Programs

We formalize skeleton programs that are targets of this paper. Our targets are skeleton programs that involve neighbor elements using combination of shift≪, shift≫, zip and map. We also deal with skeleton programs that perform one accumulation or one reduction by scan (scanr) or reduce to the result of the above programs. We define the former program

as *Program*, a program with accumulation as $Programs_S$, and a program with reduction as $Program_R$. A program of $Programs_S$ can have arbitrary number of map after scan or scanr.

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

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

Evaluation of the above defined program is given by $eval_P$, $eval_{PS}$ and $eval_{PR}$. This evaluation is the same as that removes the underlines of skeletons.

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

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

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

In the rest of this paper, we first focus on *Program* α to explain our idea. Then, we discuss $Programs_S\ \alpha$ and $Program_R\ \alpha$ in Section 6 as extensions of the optimization of *Program* α.

2.5 A Running Example

An example of our target programs, which involves neighbor elements, is computation of a numerical solution of differential equations by difference methods. We use a simple program for difference method as our running example.

We consider a wave-equation as a concrete differential equation.

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

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

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

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

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

A skeleton program *next* that computes the values at the next time form the current values of u in parallel is given as follows. Here, we consider simple boundary conditions: $u_{-1}^n = b_{l_0}$, $u_0^n = b_{l_1}$ and $u_{N+1}^n = b_r$ for a fixed N.

$$\begin{aligned} \mathit{next}\ u = \mathbf{let}\ & v'_{-2} = \mathsf{map}\ (c_{-2}\times)\ (\mathsf{shift}{\gg}\ b_{l_0}\ (\mathsf{shift}{\gg}\ b_{l_1}\ u)) \\ & v'_{-1} = \mathsf{map}\ (c_{-1}\times)\ (\mathsf{shift}{\gg}\ b_{l_1}\ u) \\ & v'_0 = \mathsf{map}\ (c_0\times)\ u \\ & v'_1 = \mathsf{map}\ (c_1\times)\ (\mathsf{shift}{\ll}\ b_r\ u) \\ & v_- = \mathsf{map}\ \mathit{add}\ (\mathsf{zip}\ v'_{-2}\ v'_{-1}) \\ & v_+ = \mathsf{map}\ \mathit{add}\ (\mathsf{zip}\ v'_0\ v'_1) \\ \mathbf{in}\ & \mathsf{map}\ \mathit{add}\ (\mathsf{zip}\ v_-\ v_+) \end{aligned}$$

We use intermediate variables v'_{-2}, v'_{-1}, v'_0, and v'_1 for readability, although the definition of target programs *Program* does not have variables. The correspondence of the program *next* and the above recurrence equation is as follows. First, to generate a list corresponding to the first term $c_{-2}u_{i-2}^n$, we apply two $\mathsf{shift}{\gg}$ to shift the elements, and apply map to multiply the coefficient c_{-2}. Similarly, lists corresponding to the second through the fourth terms are generated by using $\mathsf{shift}{\ll}$, $\mathsf{shift}{\gg}$ and map. Then, zipping these four lists by zip and adding elements by $\mathsf{map}\ \mathit{add}$, we obtain the final result. Since each of four lists are shifted by $\mathsf{shift}{\gg}$ and $\mathsf{shift}{\ll}$, ith element of the resulting list is $c_{-2}u_{i-2} + c_{-1}u_{i-1} + c_0 u_i + c_1 u_{i+1}$.

In the rest of this paper, we explain our idea by using this example *next*.

2.6 A Strategy for Optimization

We propose specific optimization for our target skeleton programs according to the following strategy. This strategy is based on the observation that domain-specific skeleton programs are often developed with a fixed set of skeletons composed in a specific manner.

1. Design a normal form that abstracts target computations.
2. Develop fusion rules that transform a skeleton program into the normal form.
3. Implement the normal form efficiently in parallel.

In designing a normal form, we should have the following regards in mind. A normal form is specified to describe any computation of target programs but should not be too general. A normal form specific to the target programs enables us to develop efficient implementation for it. In addition, a normal form should be closed under the fusion rules to maintain the result of optimization in the form. Once we formalize a normal form with fusion rules and efficient implementation, we can perform the optimization easily: we first transform a skeleton program into the normal form with the fusion rules, and then we translate the program in the normal form to an efficient program.

3 Normal Form of Target Skeleton Programs

In this section, we formalize a normal form that describes the computation of *Program* α that involves neighbor elements by using $\mathsf{shift}{\ll}$, $\mathsf{shift}{\gg}$, zip and map. The objective of this normal form is to perform main part of such computation in one loop.

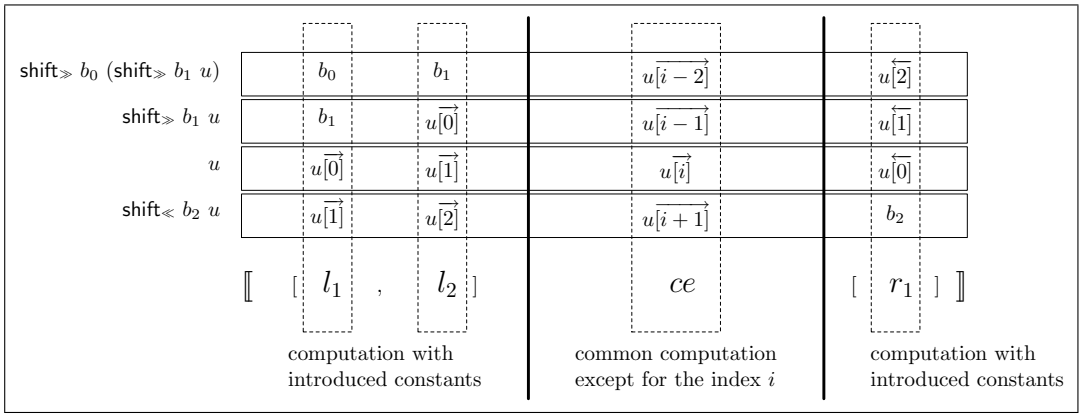


Fig. 2. Regions of elements calculated by triples for *next*. A list of computational trees for elements on the left edge $[l_1, l_2]$. A common computational tree for elements in the center part ce. A list of computational trees for elements on the right edge $[r_1]$. Elements involved in the computation are boxed by dashed lines.

3.1 Example of Normal Form

Consider calculating the running example *next* in a single loop. By a simple observation, its computation can be divided into three parts according to forms of computations of elements: the computation of the center part elements where only elements of the given list are involved, the computation of the left part where constants introduced by $\mathsf{shift}_{\gg}$ are involved, and the computation of the right part where constants introduced by $\mathsf{shift}_{\ll}$ are involved (shown in Fig. 2).

For the example *next*, each element in the center part is calculated by the following computation. Here, $u[\overrightarrow{i}]$ denotes ith element form the left of u.

$$ce = add(add(c_{-2}{\times}u[\overrightarrow{i-2}], c_{-1}{\times}u[\overrightarrow{i-1}]), add(c_0{\times}u[\overrightarrow{i}], c_1{\times}u[\overrightarrow{i+1}]))$$

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

On the other hand, the leftmost element is calculated by the following.

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

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

$$l_2 = add(add(c_{-2}{\times}b_{l_1}, c_{-1}{\times}u[\overrightarrow{0}]), add(c_0{\times}u[\overrightarrow{1}], c_1{\times}u[\overrightarrow{2}]))$$
$$r_1 = add(add(c_{-2}{\times}u[\overleftarrow{2}], c_{-1}{\times}u[\overleftarrow{1}]), add(c_0{\times}u[\overleftarrow{0}], c_1{\times}b_r))$$

Summarizing these observations, whole computation of *next* can be denoted by the following triple of computations.

- a list of computational trees for elements on the left edge $[l_1, l_2]$
- common computational tree for elements in the center part ce
- a list of computational trees for elements on the right edge $[r_1]$

Figure 3 shows these computational trees. Here, $u_{\ll 1}$ denotes the element $u[\overrightarrow{i-1}]$ on the left of the ith element. Similarly, $u_{\gg 1}$ denotes the element on the right, and $u_{\ll 2}$ denotes the second element on the left. Figure 2 shows a region that each member of the triple

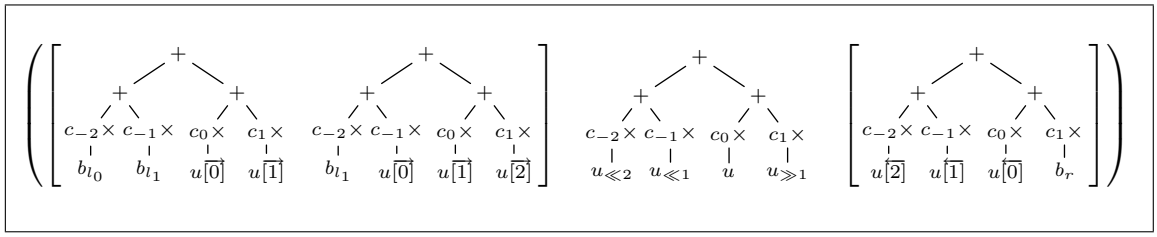


Fig. 3. Triple describing computation of *next*.

calculates. To obtain the result from this triple, we calculate each element on the edges by its computational tree, and calculate elements on the center part by the common computational tree against indices in a single loop.

Generally, such a triple denotes a computation of a target skeletal program. Thus, in the next section, we formalize this triple as a normal form of our target skeletal programs. Transformation of a skeleton program into a normal form is shown in Section 4.

3.2 Definition of Normal Form

A normal form is defined as a triple of two lists of computational trees for elements on the edges and a common computational tree for the center part.

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

We denote this triple by using special brackets like $[\![\, ls, zms, rs \,]\!]$. Here, ls is the list of computational trees for the left edge, zms is the common computational tree for the center part, and rs is the list of computational trees for the right edge. Computational trees in normal forms are defined as follows.

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

Here, types β and γ are bound locally. A node of the tree holds left and right children zipped by zip and a function applied by successive map. There are two kinds of leaves: Leaf_c denotes a constant introduced by $\mathsf{shift}_{\ll}$ or $\mathsf{shift}_{\gg}$, and Leaf_v denotes a list variable and holds a function applied by successive map. Access to a list is formalized by $\mathit{Var}\ \alpha$: *Var* is used is index access of the center part, and *Fix* is used in computations edge elements where the indices are fixed. Thus, *Var* holds the list and the amount of shifting, while *Fix* holds the list, the index and the origin of the index. The origin of the fixed index is specified by *Direction*. A special value *Hole* is used in parallel implementation.

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

$$\mathit{Node}\ \mathit{add}\ (\mathit{Leaf}_c\ (c_{-2}{\times}b_{l_1}))\ (\mathit{Leaf}_v\ (c_{-1}{\times})\ (\mathit{Fix}\ u\ 0\ \mathit{FromL}))$$

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

$$\mathit{Leaf}_v\ (c_{-2}{\times})\ (\mathit{Var}\ u\ (-2))$$

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

3.3 Semantics of Normal Form

We give a semantics of the normal form by sequential evaluation of the normal form. The center part is calculated by a single loop with the common computational tree. Each element on both edges is calculated by its own computational tree.

First, we define the evaluation of computational trees by evaluation function $eval_{T0}$ and $eval_T$ as follows. This evaluation uses an auxiliary function $eval_V$ defined below.

$$
\begin{aligned}
&eval_{T0} :: \mathit{Tree}\ \alpha \to \alpha \\
&eval_{T0}\ x = eval_T\ x\ 0 \\
&eval_T :: \mathit{Tree}\ \alpha \to \mathit{Int} \to \alpha \\
&eval_T\ (\mathit{Node}\ f\ l\ r)\ i = f\ (eval_T\ l\ i, eval_T\ r\ i) \\
&eval_T\ (\mathit{Leaf}_v\ f\ v)\ i\ \ = f\ (eval_V\ v\ i) \\
&eval_T\ (\mathit{Leaf}_c\ c)\ i\ \ \ \ = c
\end{aligned}
$$

The evaluation function $eval_{T0}$ for elements on the left and right edges is defined by $eval_T$ ignoring the index. The evaluation function $eval_T$ performs computation according to the definition of computational trees. The auxiliary function $eval_V$ processes index accessing of input lists. The result of $eval_V$ for *Hole* is not defined.

$$
\begin{aligned}
&eval_V :: \mathit{Var}\ \alpha \to \mathit{Int} \to \alpha \\
&eval_V\ (\mathit{Var}\ u\ s)\ i\ \ \ \ \ \ \ \ = \text{at}\ (i - s)\ u \\
&eval_V\ (\mathit{Fix}\ u\ s\ \mathit{FromL})\ \_ = \text{at}\ s\ u \\
&eval_V\ (\mathit{Fix}\ u\ s\ \mathit{FromR})\ \_ = \text{at}\ s\ (\text{rev}\ u)
\end{aligned}
$$

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

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

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

4 Fusion Rules for Transformation to Normal Form

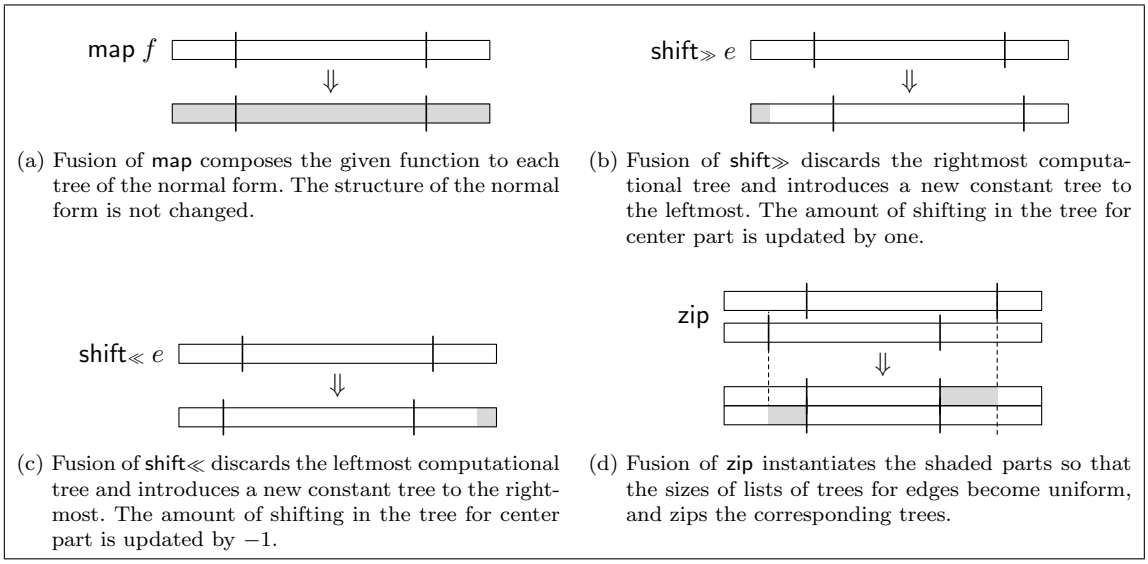
In this section, we define rules to transform a skeleton program *Program* α into a normal form.

4.1 Transformation with Fusion Rules

The transformation of a skeleton program into a normal form is performed by the following function *compile* and fusion rules one by one.

$$
\begin{aligned}
&compile :: \mathit{Program}\ \alpha \to \mathit{NForm}\ \alpha \\
&compile\ (\underline{\mathsf{map}}\ f\ x)\ \ = \mathit{fuseMap}\ f\ (compile\ x) \\
&compile\ (\underline{\mathsf{shift}}_{\ll}\ e\ x) = \mathit{fuseShift}_{\ll}\ e\ (compile\ x) \\
&compile\ (\underline{\mathsf{shift}}_{\gg}\ e\ x) = \mathit{fuseShift}_{\gg}\ e\ (compile\ x) \\
&compile\ (\underline{\mathsf{zip}}\ x\ y)\ \ \ \ = \mathit{fuseZip}\ (compile\ x)\ (compile\ y) \\
&compile\ (x)\ \ \ \ \ \ \ \ \ \ = [\![\, [\,], \mathit{Leaf}_v\ \text{id}\ (\mathit{Var}\ x\ 0), [\,] \,]\!]
\end{aligned}
$$

Fusion rules for each skeletons are defined as follows. Figure 4 illustrates the fusion rules.



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

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

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

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

Fig. 4. An image of fusion rules. Rectangles show the resulting lists. The three parts separated by vertical lines correspond to the triple of a normal form. Changed parts are shaded.

Fusion Rule for map Fusion of skeleton map is performed by composing the given function to roots of computational trees.

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

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

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

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

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

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

For the left-shift, the fusion discards the leftmost computational tree and introduces a constant computational tree to the rightmost. Then, it updates the amount of shifting in the common computational tree for the center part. This update is performed by the following *slide*. The right-shift is similar to the left-shift.

$$
\begin{aligned}
&\mathit{slide} :: \mathit{Int} \to \mathit{Tree}\ \alpha \to \mathit{Tree}\ \alpha \\
&\mathit{slide}\ d\ (\mathit{Node}\ f\ l\ r) = \mathit{Node}\ f\ (\mathit{slide}\ d\ l)\ (\mathit{slide}\ d\ r) \\
&\mathit{slide}\ d\ (\mathit{Leaf}_v\ f\ (\mathit{Var}\ x\ s)) = \mathit{Leaf}_v\ f\ (\mathit{Var}\ x\ (s + d)) \\
&\mathit{slide}\ d\ x = x
\end{aligned}
$$

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

$$
\begin{aligned}
&\mathit{fuseZip} :: \mathit{NForm}\ \alpha \to \mathit{NForm}\ \beta \to \mathit{NForm}\ (\alpha, \beta)\\
&\mathit{fuseZip}\ [\![\ \mathit{ls}_1, \mathit{zms}_1, \mathit{rs}_1\ ]\!]\ [\![\ \mathit{ls}_2, \mathit{zms}_2, \mathit{rs}_2\ ]\!]\\
&\quad = \mathbf{let}\ \mathit{zms} = \mathit{Node}\ \mathrm{id}\ \mathit{zms}_1\ \mathit{zms}_2\\
&\qquad\quad \mathit{ls} = \mathit{trim}\ \mathit{FromL}\ \mathit{ls}_1\ \mathit{ls}_2\ \mathit{zms}_1\ \mathit{zms}_2\\
&\qquad\quad \mathit{rs} = \mathit{trim}\ \mathit{FromR}\ (\mathrm{rev}\ \mathit{rs}_1)\ (\mathrm{rev}\ \mathit{rs}_2)\ \mathit{zms}_1\ \mathit{zms}_2\\
&\qquad \mathbf{in}\ [\![\ \mathit{ls}, \mathit{zms}, \mathrm{rev}\ \mathit{rs}\ ]\!]
\end{aligned}
$$

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

$$
\begin{aligned}
&\mathit{trim} :: \mathit{Direction} \to [\mathit{Tree}\ \alpha] \to [\mathit{Tree}\ \beta] \quad \to \mathit{Tree}\ \alpha \to \mathit{Tree}\ \beta \to [\mathit{Tree}\ (\alpha, \beta)]\\
&\mathit{trim}\ d\ \mathit{ts}_1\ \mathit{ts}_2\ \mathit{zms}_1\ \mathit{zms}_2\\
&\quad = \mathbf{let}\ n_1 = \mathrm{length}\ \mathit{ts}_1\ ;\ n_2 = \mathrm{length}\ \mathit{ts}_2\\
&\qquad (\mathit{ts}'_1, \mathit{ts}'_2) = (\mathit{ts}_1 +\!\!+ \mathit{insts}\ d\ \mathit{zms}_1\ n_1\ n_2, \mathit{ts}_2 +\!\!+ \mathit{insts}\ d\ \mathit{zms}_2\ n_2\ n_1)\\
&\quad \mathbf{in}\ \mathsf{zipWith}\ (\mathit{Node}\ \mathrm{id})\ \mathit{ts}'_1\ \mathit{ts}'_2
\end{aligned}
$$

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

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

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

Theorem 1. *Any skeleton program defined by Program can be transformed into the normal form by using the four fusion rules and the base case rule.*

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

Complete proof is shown in Appendix A □

4.2 Example of Transformation

As a brief explanation of the rules, we transform the example *next* into the normal form. For readability, we use a brief notation used in the previous examples instead of the data structures.

Most simplest case is the transformation of the argument list u. A list u needs only the common computational tree $u\overrightarrow{[i]}$ that is just the element of u.

$$u \Rightarrow [\![\ [\,], \mathrm{id}\ u, [\,]\ ]\!]$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

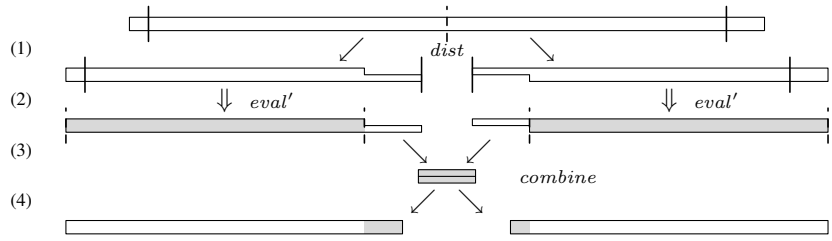
$$[\![\ \ [(c_{-2} \times b_{l_0}, c_{-1} \times b_{l_1}), (c_{-2} \times b_{l_1}, c_{-1} \times u\overrightarrow{[0]})], (c_{-2} \times u_{\gg 2}, c_{-1} \times u_{\gg 1}), [\,]\ \ ]\!]$$

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

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

5 Parallel Implementation of Normal Form

In this section, we give a parallel implementation of the normal form.



(1) Lists in a normal form are divided into two parts to generate two normal forms. Then, these normal forms are distributed to processors.
(2) Each processor calculates elements that can be calculated by local data. Elements on the edges cannot be calculated because they need data on other processors.
(3) To calculate the elements on the edges, communication is performed to complete the data.
(4) Perform calculation of the elements on the edges. Combine the edge elements to the other elements.

Fig. 5. An image of parallel implementation of the normal form (two processors).

5.1 Parallel Implementation of Normal Form

Based on parallel implementation of existing skeletons [8], we consider parallel implementation of the normal form by four steps: (1) distribution of a normal form (input lists), (2) the first local computation, (3) global communication, (4) the second local computation. We explain the idea of parallel implementation using the example *next*. Figure 5 shows an image of parallel implementation of the normal form using two processors.

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

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

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

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

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

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

Since these elements cannot be calculated in this local phase, computational trees for these elements are held until the global computation phase. Here, a hole $\bullet$ is substituted to an element that is not available in this phase.

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

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

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

Note that this tree complement the tree generated from nf_1. Generally, letting the maximum amounts of shifting to the left and the right be l and r, the number of computational trees with holes generated on each edge is $l+r$. Trees on both sides of an edge are complementary to each other.

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

Fourth, in the second local step, each processor calculates the elements on the edges with the completed trees to complete the resulting distributed list.

For the example *next*, nf_1 generates the following trees with holes,

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

while nf_2 generates the following trees.

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

Zipping these trees, we obtain the following complete computational trees for elements on the edge.

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

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

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

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

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

5.2 Formalization of Parallel Implementation of Normal Form

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

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

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

Here, *division* p n calculates the division of input lists, and *distribute zms divs* distributes the input lists according to the division. This process generates distributes a computational tree zms_i that is generated by replacing a list u $(= u_1 +\!\!+ \cdots +\!\!+ u_i +\!\!+ \cdots +\!\!+ u_p)$ in the original tree *zms* with u_i, and zms_i is distributed to ith processor and held in the normal form nf_i (this is generated by zip_3 *lss zmss rss*). The list of computational trees for the left edge is held in the first normal form nf_1, while the list of computational trees for the right edge is held in the last normal form nf_p.

Next, we define the triple of the partial result *PResult*.

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

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

Next, we define an auxiliary function *maxShift* that calculates the maximum amount of shifting.

$$
\begin{aligned}
&\mathit{maxShift} :: \mathit{Tree}\ \alpha \to (\mathit{Int}, \mathit{Int}) \\
&\mathit{maxShift}\ (\mathit{Node}\ \_\ l\ r) = \mathbf{let}\ (l_1, r_1) = \mathit{maxShift}\ l \\
&\qquad\qquad\qquad\qquad\qquad\quad (l_2, r_2) = \mathit{maxShift}\ r \\
&\qquad\qquad\qquad\qquad\quad \mathbf{in}\ (\max\ l_1\ l_2, \max\ r_1\ r_2) \\
&\mathit{maxShift}\ (\mathit{Leaf}_v\ \_\ (\mathit{Var}\ \_\ s)) = \mathbf{if}\ s < 0\ \mathbf{then}\ (-s, 0)\ \mathbf{else}\ (0, s) \\
&\mathit{maxShift}\ \_ \qquad\qquad\qquad\quad = (0, 0)
\end{aligned}
$$

We define a general function gEval' to perform the local computation.

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

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

$$
\begin{aligned}
&\mathit{instP} :: \mathit{Direction} \to \mathit{Tree}\ \alpha \to \mathit{Int} \to \mathit{Tree}\ \alpha \\
&\mathit{instP}\ d\ (\mathit{Node}\ f\ l\ r)\ i = \mathit{Node}\ f\ (\mathit{instP}\ d\ l\ i)\ (\mathit{instP}\ d\ r\ i) \\
&\mathit{instP}\ d\ (\mathit{Leaf}_v\ f\ (\mathit{Var}\ x\ s))\ i = \mathbf{let}\ s' = \mathbf{case}\ d\ \mathbf{of}\ \mathit{FromL} \to s;\ \mathit{FromR} \to -s \\
&\qquad\qquad\qquad\qquad\qquad\quad i' = (-s' + i) \\
&\qquad\qquad\qquad\qquad\quad \mathbf{in\ if}\ (i' \le (\mathrm{length}\ x)) || (i' < 0)\ \mathbf{then}\ \mathit{Leaf}_v\ f\ \mathit{Hole} \\
&\qquad\qquad\qquad\qquad\qquad\quad \mathbf{else}\ \mathit{Leaf}_v\ f\ (\mathit{Fix}\ x\ i'\ d) \\
&\mathit{instP}\ d\ x\ i = x
\end{aligned}
$$

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

Using $gEval'$, we define the function $eval'$ that performs local computation of the normal form as follows.

$$
\begin{aligned}
&eval' :: NForm\alpha \to PResult\alpha[\alpha] \\
&eval' = gEval'\ fcMap \\
&\quad \textbf{where}\ fcMap\ [\![\, ls, zms, rs \,]\!]\ idces \\
&\qquad\qquad = \mathsf{map}\ eval_{T0}\ ls +\!\!+ \mathsf{map}\ (eval_T\ zms)\ idces +\!\!+ \mathsf{map}\ eval_{T0}\ rs
\end{aligned}
$$

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

Next, we define the process of completion of a computational tree from two trees complementary to each other.

$$
\begin{aligned}
&combine :: Tree\ \alpha \to Tree\ \alpha \to Tree\ \alpha \\
&combine\ (Node\ f_1\ l_1\ r_1)\ (Node\ f_2\ l_2\ r_2) = Node\ f_1\ (combine\ l_1\ l_2)\ (combine\ r_1\ r_2) \\
&combine\ (Leaf_v\ f_1\ Hole)\ (Leaf_v\ f_2\ v) \quad = Leaf_v\ f_2\ v \\
&combine\ (Leaf_v\ f_1\ v)\ (Leaf_v\ f_2\ Hole)) \quad = Leaf_v\ f_1\ v \\
&combine\ (Leaf_c\ c)\ (Leaf_c\ c') \quad = Leaf_c\ c
\end{aligned}
$$

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

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

This operator takes an associative operator and a function, and calculates the reduction of the elements on edges. This reduction is done by the following general function.

$$
\begin{aligned}
&\mathsf{cata} :: (\alpha \to \alpha \to \alpha) \to (\beta \to \alpha) \to [\beta] \to \alpha \\
&\mathsf{cata}\ (\oplus)\ f\ [\,] \quad = \imath_\oplus \\
&\mathsf{cata}\ (\oplus)\ f\ (a : x) = f\ a \oplus \mathsf{cata}\ (\oplus)\ f\ x
\end{aligned}
$$

Since the computation of the normal form generates a list, we use $\boxplus_{(+\!\!+,[\cdot])}$ that performs reduction by $+\!\!+$ and $[\cdot]$.

Using the functions defined above, we define the parallel implementation of normal form *parEval* as follows.

$$
\begin{aligned}
&parEval :: Int \to NForm\ \alpha \to [\alpha] \\
&parEval\ p{=}extract \circ \mathsf{reduce}\ (\boxplus_{(+\!\!+,[\cdot])}) \circ \mathsf{map}\ eval' \circ dist\ p
\end{aligned}
$$

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

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

6 Expansion of Target Programs

Based on the results for *Program*, we expand our target programs to $Program_S$, which includes accumulation by scan or scanr, and $Program_R$, which includes reduction by reduce (see Section 2).

6.1 Target Programs with Accumulation

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

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

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

A skeleton program that solves this tridiagonal linear system of equations is given as follows. This program is based on the LU decomposition of the coefficient matrix $A = LU$ [11].

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

Here, *us* corresponds to the upper triangular matrix U, *ms* corresponds to the lower triangular matrix L, *ys* is the solution of the linear equation $Lys = bs$, and *xs* the solution of the linear equation $Axs = bs$. The operators $\otimes_1$, $\otimes_2$ and $\otimes_3$ are multiplication of 2×2 matrices, although $\otimes_2$ and $\otimes_3$ omit the half of the elements. This program consists of three parts: the first part performs the LU decomposition with scan to obtain *us* and *ms*, the second part performs the forward substitution by scan to obtain *ys*, and the third part performs the backward substitution by scanr to obtain *xs*.

Normal Form with Accumulation We extend the normal form of *Program* to hold the operator, the direction and the element of the accumulation, and the functions applied by the last map. Thus, the extended normal form *NFormS* is defined as follows.

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

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

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

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

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

Fusion Rules with Accumulation We extend the fusion rules to handle scan, scanr and map after accumulation. The transformation of a skeleton program into an extended normal form is performed by the following function *compileS* and fusion rules one by one.

$$\begin{aligned}
&\mathit{compileS} :: \mathit{Program}_S\ \alpha \to \mathit{NFormS}\ \alpha \\
&\mathit{compileS}\ (\underline{\mathsf{scan}}\ (\oplus)\ e\ x) \ = \mathit{fuseScan}\ (\oplus)\ e\ (\mathit{compile}\ x) \\
&\mathit{compileS}\ (\underline{\mathsf{scanr}}\ (\oplus)\ e\ x) = \mathit{fuseScanr}\ (\oplus)\ e\ (\mathit{compile}\ x) \\
&\mathit{compileS}\ (\underline{\mathsf{map}}\ f\ x) \qquad = \mathit{fuseMapS}\ f\ (\mathit{compileS}\ x)
\end{aligned}$$

Transformation of the computation before the accumulation is done by *compile* defined in Section 4.

Fusion rules for accumulations are as follows.

$$\begin{aligned}
&\mathit{fuseScanr} :: (\alpha \to \alpha \to \alpha) \to \alpha \to \mathit{NForm}\ \alpha \to \mathit{NFormS}\ \alpha \\
&\mathit{fuseScanr}\ (\oplus)\ e\ \llbracket\, ls, zms, rs\, \rrbracket = (\llbracket\, ls, zms, rs\, \rrbracket, (\mathit{FromR}, (\oplus), e, \mathrm{id}))
\end{aligned}$$

$$\begin{aligned}
&\mathit{fuseScan} :: (\alpha \to \alpha \to \alpha) \to \alpha \to \mathit{NForm}\ \alpha \to \mathit{NFormS}\ \alpha \\
&\mathit{fuseScan}\ (\oplus)\ e\ \llbracket\, ls, zms, rs\, \rrbracket = (\llbracket\, ls, zms, rs\, \rrbracket, (\mathit{FromL}, (\oplus), e, \mathrm{id}))
\end{aligned}$$

Each rule appends the operator $\oplus$ and the initial element e to the normal form, and marks the direction of the accumulation by *FromL* or *FromR*. The last element of the extended normal form is initialized by the identity function id.

The fusion rule for map after accumulation is as follows.

$$\begin{aligned}
&\mathit{fuseMapS} :: (\beta \to \alpha) \to \mathit{NFormS}\ \beta \to \mathit{NFormS}\ \alpha \\
&\mathit{fuseMapS}\ f\ = (nf, (d, (\oplus), e, g)) = (nf, (d, (\oplus), e, f \circ g))
\end{aligned}$$

This rule merely composes the given function f to the function g in the extended normal form.

It is obvious that any target program can be transformed into an extended normal form by the above fusion rules and the fusion rules defined in Section 4.

Parallel Implementation of Normal form with Accumulation Based on parallel implementation of skeleton scan, we design parallel implementation of the normal form. Parallel implementation of skeleton scan is given as follows.

$$\begin{aligned}\mathsf{scan}\ (\oplus)\ e = &\mathsf{reduce}\ (+\!\!+) \circ \mathit{zipWithP}\ (\mathit{postScan}\ (\oplus))\\ &\circ ((\mathit{prescan}\ (\oplus)\ e \circ \mathsf{map}\ \mathrm{last}) \triangle \mathrm{id})\\ &\circ \mathsf{map}\ (\mathsf{scan}\ (\oplus)\ \imath_{\oplus}) \circ \mathit{dist}\ p\\ &\textbf{where}\ \mathit{postScan}\ (\oplus)\ a\ x = \mathsf{map}\ (a\oplus)\ x\end{aligned}$$

Here, $\mathit{zipWithP}\ f\ (x, y) = \mathsf{zipWith}\ f\ x\ y$ and $(f \triangle g)\ x = (f\ x, g\ x)$. First, this implementation performs local accumulation in parallel by scan. Then, it performs global accumulation by *prescan* defined below. Finally, *postScan* adds the accumulated value calculated by *prescan* to each element of the result of the local accumulation.

$$\begin{aligned}&\mathit{prescan} :: (\alpha \to \alpha \to \alpha) \to \alpha \to [\alpha] \to [\alpha]\\ &\mathit{prescan}\ (\oplus)\ e\ [\,] \quad\ = [\,]\\ &\mathit{prescan}\ (\oplus)\ e\ (a : x) = e : \mathit{prescan}\ (\oplus)\ (e \oplus a)\ x\end{aligned}$$

Based on the parallel implementation of scan, we consider the following parallel implementation of a normal form with accumulation. We only show the implementation of the normal form with accumulation by scan. The implementation for accumulation by scanr is similar to that by scan.

$$\begin{aligned}&\mathit{parEvalScan} :: \mathit{Int} \to \mathit{NFormS}\ \alpha \to [\alpha]\\ &\mathit{parEvalScan}\ p\ (nf, (\mathit{FromL}, \oplus, e, f)) = \mathsf{reduce}\ (+\!\!+) \circ \mathit{zipWithP}\ (\mathit{evalPostScan}\ (\oplus)\ f)\\ &\qquad\qquad \circ ((\mathit{prescan}\ (\boxplus_{(\oplus,\mathrm{id})})\ ([\,], e, [\,]) \circ \mathsf{map}\ \mathit{takeLast}) \triangle \mathrm{id})\\ &\qquad\qquad \circ \mathsf{map}\ (\mathit{evalScan}'\ (\oplus)\ \imath_{\oplus}) \circ \mathit{dist}\ p\ \$\ nf\end{aligned}$$

Here, $\mathit{takeLast}\ \langle\!\langle\ pls, cs, prs\ \rangle\!\rangle = \langle\!\langle\ pls, \mathrm{last}\ cs, prs\ \rangle\!\rangle$. Basic structure is the same as the implementation of scan. Main difference is that local and global computation deal with triples of partial results defined. The local computation $\mathit{evalScan}'$ that generates a partial result is defined as follows. The generated partial result holds the result of local accumulation in the center of the triple.

$$\begin{aligned}&\mathit{evalScan}' :: (\alpha \to \alpha \to \alpha) \to \alpha \to \mathit{NForm}\ \alpha \to \mathit{PResult}\ \alpha\ [\alpha]\\ &\mathit{evalScan}'\ (\oplus)\ e = \mathit{gEval}'\ \mathit{fcScan}\\ &\quad \textbf{where}\ \mathit{fcScan}\ [\![\ ls, zms, rs\ ]\!]\ idces = sls +\!\!+ scs +\!\!+ srs\\ &\qquad\quad \textbf{where}\ sls = \mathit{scata}\ (\oplus)\ e\ \mathit{eval}_{T0}\ ls\\ &\qquad\qquad\quad e' = \mathrm{last}\ (e : sls)\\ &\qquad\qquad\quad scs = \mathit{scata}\ (\oplus)\ e'\ (\mathit{eval}_{T}\ zms)\ idces\\ &\qquad\qquad\quad e'' = \mathrm{last}\ (e' : scs)\\ &\qquad\qquad\quad srs = \mathit{scata}\ (\oplus)\ e''\ \mathit{eval}_{T0}\ rs\end{aligned}$$

Here, *scata* defined below performs accumulation and evaluation of the computational tree at the same time.

$$\begin{aligned}&\mathit{scata} :: (\alpha \to \alpha \to \alpha) \to \alpha \to (\beta \to \alpha) \to [\beta] \to [\alpha]\\ &\mathit{scata}\ (\oplus)\ e\ f\ [\,] \quad\ = [\,]\\ &\mathit{scata}\ (\oplus)\ e\ f\ (a : x) = \textbf{let}\ e' = (e \oplus f\ a)\\ &\qquad\qquad\qquad\qquad\ \ \textbf{in}\ e' : \mathit{scata}\ (\oplus)\ e'\ f\ x\end{aligned}$$

The final local computation *evalPostScan* $(\oplus)$ calculates accumulation of elements on the edges and adds the accumulated value to the result of the local computation.

$$
\begin{aligned}
&\mathit{evalPostScan} :: (\alpha \to \alpha \to \alpha) \to (\alpha \to \beta)\mathit{PResult}\ \alpha\ \alpha \to \mathit{PResult}\ \alpha\ [\alpha] \to [\beta] \\
&\mathit{evalPostScan}\ (\oplus)\ f\ \langle\!\langle\ \mathit{epls}, e, \mathit{eprs}\ \rangle\!\rangle\ \langle\!\langle\ \mathit{pls}, \mathit{cs}, \mathit{prs}\ \rangle\!\rangle = \mathit{res} \\
&\quad \textbf{where}\, \mathit{fe}\ (\mathit{epr}, \mathit{pl}) = \mathit{eval}_{T0}\ (\mathit{combine}\ \mathit{epr}\ \mathit{pl})) \\
&\qquad \mathit{sls} = \mathit{scata}\ (\oplus)\ e\ \mathit{fe}\ (\mathsf{zip}\ \mathit{eprs}\ \mathit{pls}) \\
&\qquad e' = \mathrm{last}\ (e : \mathit{sls}) \\
&\qquad \mathit{scs} = \mathsf{map}\ (f \circ e' \oplus)\ \mathit{cs} \\
&\qquad \mathit{res} = \mathsf{map}\ f\ \mathit{sls} +\!\!+ \mathit{scs}
\end{aligned}
$$

6.2 Target Programs with Reduction

Next, we expand our target programs to $\mathit{Program}_R$.

Normal Form with Reduction We extend the normal form of *Program* to hold the operator of the reduction. Thus, the extended normal form *NFormR* is defined as follows.

$$\textbf{type}\ \mathit{NFormR}\ \alpha = (\mathit{NForm}\ \beta, \beta \to \beta \to \beta)$$

In the extended normal form $(\llbracket\ ls, zms, rs\ \rrbracket, \oplus)$, $\llbracket\ ls, zms, rs\ \rrbracket$ specifies the computation before the accumulation, and $\oplus$ is the associative binary operator used in the reduction.

Fusion Rules with Reduction We extend the fusion rules to handle reduce. The transformation of a skeleton program into an extended normal form is performed by the following function *compileR* and fusion rules one by one.

$$
\begin{aligned}
&\mathit{compileS} :: \mathit{Program}_S\ \alpha \to \mathit{NFormS}\ \alpha \\
&\mathit{compileR}\ (\underline{\mathsf{reduce}}\ (\oplus)\ x) = \mathit{fuseReduce}\ (\oplus)\ (\mathit{compile}\ x)
\end{aligned}
$$

Transformation of the computation before the reduction is done by *compile* defined in Section 4.

The fusion rule for reduction is as follows.

$$
\begin{aligned}
&\mathit{fuseReduce} :: (\alpha \to \alpha \to \alpha) \to \mathit{NForm}\ \alpha \to \mathit{NFormR}\ \alpha \\
&\mathit{fuseReduce}\ (\oplus)\ \llbracket\ ls, zms, rs\ \rrbracket = (\llbracket\ ls, zms, rs\ \rrbracket, (\oplus))
\end{aligned}
$$

The rule appends the operator $\oplus$ to the normal form.

It is obvious that any target program can be transformed into an extended normal form by the above fusion rule and the fusion rules defined in Section 4.

Parallel Implementation of Normal Form with Reduction In this section, we give parallel implementation for $\mathit{Program}_R$ that perform reduction by reduce to the result of a normal form.

These programs can be executed in parallel by performing reduction, instead of generation of a list, in the implementation of a normal form. That is, parallel implementation is obtained by giving the reduction operator to gEval' and $\boxplus_{(\cdot,\cdot)}$.

The function $\mathit{evalReduce}'$ that performs local reduction is defined as follows.

$$
\begin{aligned}
&\mathit{evalReduce}' :: (\alpha \to \alpha \to \alpha) \to \mathit{NForm}\ \alpha \to \mathit{PResult}\ \alpha\ \alpha \\
&\mathit{evalReduce}'\ (\oplus) = \mathit{gEval}'\ \mathit{fcReduce} \\
&\quad \textbf{where}\ \mathit{fcReduce}\ \llbracket\ ls, zms, rs\ \rrbracket\ \mathit{idces} \\
&\qquad\qquad = \mathsf{cata}\ (\oplus)\ \mathit{eval}_{T0}\ ls \oplus \mathsf{cata}\ (\oplus)\ (\mathit{eval}_T\ zms)\ \mathit{idces} \oplus \mathsf{cata}\ (\oplus)\ \mathit{eval}_{T0}\ rs
\end{aligned}
$$

Table 1. Running times and speedups of parallel programs against the number of processors. A speedup is one with respect to a sequential program.

#processors		1	2	4	8	16	24	32	48	64
next	time (s)	210.25	100.84	48.12	24.41	13.31	8.86	6.52	4.70	3.50
	speedup	0.09	0.20	0.41	0.81	1.49	2.24	3.04	4.23	5.67
next_opt	time (s)	19.86	9.64	4.93	2.44	1.26	0.87	0.70	0.54	0.47
	speedup	1.00	2.06	4.03	8.14	15.79	22.76	28.26	36.73	42.22

```
r[0] = c_2*b0 + c_1*b1 + c0*u[0] + c1*u[1];
r[1] = c_2*b1 + c_1*u[0] + c0*u[1] + c1*u[2];
for(int i = 2; i < n-1; i++) {
  r[i] = c_2*u[i-2] + c_1*u[i-1] + c0*u[i] + c1*u[i+1];
}
r[n-1] = c_2*u[n-3] + c_1*u[n-2] + c0*u[n-1] + c1*b2;
```

Fig. 6. A sequential program of *next*

This function performs reduction and evaluation of computational trees at the same time by cata $(\oplus)$ $eval_{T0}$ and cata $(\oplus)$ $(eval_T\ zms)$. Using this function, parallel implementation of a normal form with reduction is defined as follows.

$$parEvalReduce :: Int \to NFormR\ \alpha \to \alpha$$
$$parEvalReduce\ p\ (nf, \oplus) = extract \circ \mathsf{reduce}\ (\boxplus_{(\oplus,\mathrm{id})}) \circ \mathsf{map}\ (evalReduce'\ (\oplus)) \circ dist\ p\ \$\ nf$$

7 Experiment Result

We implemented a small domain-specific optimizer for the target programs. The system reads a skeleton program written with our skeleton library SkeTo [8], and generates an optimized C++ code of the program. For the examples *next* and *solveTS*, we measured running times of skeleton programs and optimized programs. We used a PC cluster where each of the nodes connected with Gigabit Ethernet has a CPU of Intel® Xeon®2.80GHz and 2GB memory, with Linux 2.4.21, GCC 4.1.1, and mpich 1.2.7.

7.1 Result of *next*

Table 1 shows measured running times and speedups. Running time is of applying *next* 100 times to an input list of 10,000,000 elements. A speedup is a ratio of running time of a parallel program to running time of a sequential program (shown in Figure 6).

The optimized program (*next_opt*) achieves ten times faster running time than the original skeleton program, and the same running time as a sequential program on one processor. This improvement was gained by elimination of redundant intermediate data and communications. Also, the optimized program achieves good speedups against the number of processors. These results show effectiveness of the proposed optimization.

7.2 Result of *solveTS*

Table 2 shows measured running times and speedups. Running time is of applying *solveTS* 10 times to an input list of 1,000,000 elements. A speedup is a ratio of running time of a parallel program to running time of *solveTS* on one processor.

The optimized program (*solveTS_opt*) achieves about 20% faster running time than the original skeleton program. This improvement was gained by elimination of redundant intermediate data and communications. Also, the optimized program achieves good speedups against the number of processors. These results show effectiveness of the proposed optimization.

Table 2. Running times and speedups of parallel programs against the number of processors. A speedup is one with respect to *solveTS* on one processor.

#processors		1	2	4	8	16	32	64
solveTS	time (s)	117.03	69.92	40.54	20.21	10.14	4.97	2.79
	speedup^p	1.00	1.67	2.89	5.79	11.54	23.56	41.92
solveTS_opt	time (s)	62.20	60.81	30.47	15.26	7.66	3.91	2.18
	speedup^p	1.88	1.92	3.84	7.67	15.28	29.92	53.61

8 Discussion and Related Work

One of the simplest fusion optimizations so far uses a general form called cataJ [9]. This cataJ can describe the skeleton map and the reduction skeleton reduce, and consists of two things: a function applied to each element of the input list, and an associative binary operator used to reduce those elements. The general form cataJ can describe any computation written as composition of any number of map and at most one reduce at the last. Thus, cataJ is thought to be a normal form of skeleton programs described with such compositions.

Hu et al. [7] proposed a more general fusion optimization using a general form called accumulate with some fusion rules. This accumulate can describe skeleton scan, which calculates an accumulation of the input list with an associative binary operator, as well as map and reduce. The general form accumulate essentially consists of two pairs of a function and an associative binary operator, and it is thought to be a normal form of skeleton programs described with compositions of the skeletons. This accumulate is powerful so that it can describe shift$_{\gg}$ and shift$_{\ll}$ too. However, describing shift$_{\gg}$ and shift$_{\ll}$ with accumulate cause some overheads due to lack of special consideration of elements on the edges. The overheads are extensions of elements to treat all elements uniformly, and logarithmic steps of interprocessor communications for general implementation of accumulation. Thus, we need to consider a specific fusion optimization, i.e. a normal form, fusion rules and efficient implementation.

Our normal form extends these fusion optimizations with special consideration of elements on the edges, which are introduced by shift$_{\gg}$ and shift$_{\ll}$. The normal form separates computation of elements on the edges from that of center elements, so that it does not introduce the overheads accumulate needed.

Grelck et al. [6] proposed an optimization based on fusion of a general skeleton called WITH-loop. Since their WITH-loop are based on index accessing, their method can deal with shifting operation such as shift$_{\gg}$ and shift$_{\ll}$. Also, their fusion can handle nesting use of skeletons. However, skeletons that perform accumulation or reduction, in which a region of elements required in computation of an element varies, cannot be described with WITH-loop.

Generalized scan (gen_scan) proposed by Fischer et al. [3] can perform accumulation that involves neighbor elements. We can describe shifting operations such as shift$_{\gg}$ and shift$_{\ll}$ with their gen_scan. We think programs that perform accumulation after combinations of map, zip, shift$_{\gg}$ and shift$_{\ll}$ can be described by gen_scan. However, since the condition for parallel implementation of gen_scan is very complicated, formalization of fusion rules of gen_scan is very difficult.

9 Conclusion

In this paper, we proposed an optimization technique for skeletons programs that involve neighbor elements, which cannot gain adequate efficiency with existing fusion techniques. We give a normal form that abstracts computations that involve neighbor elements, fusion

rules of skeletons into a normal form, and an efficient parallel implementation of a normal form. The optimization is performed by transforming a skeleton program with fusion rules into a normal form that has efficient implementation. Experimental results show the optimized program can be executed efficiently in parallel, and show effectiveness of proposed optimization. We consider extension of this technique to multidimensional data structures and nesting use of skeletons as our future work.

References

1. R. S. Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall Series in Computer Science. Prentice Hall, 2nd edition, April 1998.
2. M. Cole. *Algorithmic Skeletons: Structural Management of Parallel Computation*. Research Monographs in Parallel and Distributed Computing. MIT Press, 1989.
3. P. F. Fischer, F. P. Preparata, and J. E. Savage. Generalized scans and tridiagonal systems. *Theoretical Computer Science*, 255(1–2):423–436, 2001.
4. A. J. Gill, J. Launchbury, and S. L. P. Jones. A short cut to deforestation. In *FPCA '93 Conference on Functional Programming Languages and Computer Architecture. Copenhagen, Denmark, 9–11 June 1993*, pages 223–232. ACM Press, 1993.
5. S. Gorlatch, C. Wedler, and C. Lengauer. Optimization rules for programming with collective operations. In *13th International Parallel Processing Symposium / 10th Symposium on Parallel and Distributed Processing (IPPS / SPDP '99), 12-16 April 1999, San Juan, Puerto Rico, Proceedings*, pages 492–499. IEEE Computer Society, 1999.
6. C. Grelck and S.-B. Scholz. Merging compositions of array skeletons in SaC. *Parallel Computing*, 32(7–8):507–522, 2006.
7. Z. Hu, H. Iwasaki, and M. Takeichi. An accumulative parallel skeleton for all. In D. L. Métayer, editor, *Programming Languages and Systems, 11th European Symposium on Programming, ESOP 2002, held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings*, volume 2305 of *Lecture Notes in Computer Science*, pages 83–97. Springer, 2002.
8. K. Matsuzaki, H. Iwasaki, K. Emoto, and Z. Hu. A library of constructive skeletons for sequential style of parallel programming. In *InfoScale '06: Proceedings of the 1st international conference on Scalable information systems*, volume 152 of *ACM International Conference Proceeding Series*, page 13. ACM Press, 2006.
9. K. Matsuzaki, K. Kakehi, H. Iwasaki, Z. Hu, and Y. Akashi. A fusion-embedded skeleton library. In M. Danelutto, M. Vanneschi, and D. Laforenza, editors, *Euro-Par 2004 Parallel Processing, 10th International Euro-Par Conference, Pisa, Italy, August 31 – September 3, 2004, Proceedings*, volume 3149 of *Lecture Notes in Computer Science*, pages 644–653. Springer, 2004.
10. F. A. Rabhi and S. Gorlatch, editors. *Patterns and Skeletons for Parallel and Distributed Computing*. Springer-Verlag, 2002.
11. H. S. Stone. An efficient parallel algorithm for the solution of a tridiagonal linear system of equations. *Journal of the ACM*, 20(1):27–38, 1973.
12. P. Wadler. Deforestation: Transforming programs to eliminate trees. In H. Ganzinger, editor, *ESOP '88, 2nd European Symposium on Programming, Nancy, France, March 21-24, 1988, Proceedings*, volume 300 of *Lecture Notes in Computer Science*, pages 344–358. Springer, 1988.
13. C. Wedler and C. Lengauer. On linear list recursion in parallel. *Acta Informatica*, 35(10):875–909, October 1998.

A Proof of Theorem 1

In the following, we use an abbreviation of the index accessing for readability: $\langle x \rangle_i = \text{at } i\ x$. We assume that the length of an input list is n. The goal of this proof is to show the equation $\mathit{eval}_P\ \mathit{prog} = \mathit{eval}\ (\mathit{compile}\ \mathit{prog})$ for any prog.

A.1 Base Case

What we have to prove is the following equation for a list x.

$$\mathit{eval}_P\ x = \mathit{eval}\ [\![\, [], \mathit{Leaf}_c\ \mathrm{id}\ (\mathit{Var}\ x\ 0), [] \,]\!] \tag{1}$$

The ith element of the left-hand sides is as follows.

$$\begin{aligned} \langle \mathit{eval}_P\ x \rangle_i = \quad & \{\ \text{definition of } \mathit{eval}_P\ \} \\ & \langle \mathit{eval}_P\ x \rangle_i \end{aligned}$$

The ith element of the right-hand sides is as follows.

$$\begin{aligned} \langle \mathit{eval}\ [\![\, [], \mathit{Leaf}_c\ \mathrm{id}\ (\mathit{Var}\ x\ 0), [] \,]\!] \rangle_i = \quad & \{\ \text{definition of } \mathit{eval}\ \} \\ & \langle \mathsf{map}\ \mathit{eval}_T\ (\mathit{Leaf}_c\ \mathrm{id}\ (\mathit{Var}\ x\ 0))\ [0..n-1] \rangle_i \\ = \quad & \{\ i\text{th element}\ \} \\ & \mathit{eval}_T\ (\mathit{Leaf}_c\ \mathrm{id}\ (\mathit{Var}\ x\ 0))\ i \\ = \quad & \{\ \text{definition of } \mathit{eval}_T \text{ and the identity function}\ \} \\ & \mathit{eval}_V\ (\mathit{Var}\ x\ 0)\ i \\ = \quad & \{\ \text{definition of } \mathit{eval}_V \text{ and } i\text{th element}\ \} \\ & \langle x \rangle_i \end{aligned}$$

Thus, the following equation holds.

$$\langle \mathit{eval}_P\ x \rangle_i = \langle \mathit{eval}\ [\![\, [], \mathit{Leaf}_c\ \mathrm{id}\ (\mathit{Var}\ x\ 0), [] \,]\!] \rangle_i$$

Since this equation holds for $i \in [0..n-1]$, the equation (1) holds.

A.2 Inductive Case for map

What we have to prove is the following equation for a function f and a program prog. Here, $\mathit{compile}\ \mathit{prog} = [\![\, \mathit{ls}, \mathit{zms}, \mathit{rs} \,]\!]$.

$$\mathit{eval}_P\ (\underline{\mathsf{map}}\ f\ \mathit{prog}) = \mathit{eval}\ [\![\, \mathsf{map}\ (\mathit{comp}\ f)\ \mathit{ls}, \mathit{comp}\ f\ \mathit{zms}, \mathsf{map}\ (\mathit{comp}\ f)\ \mathit{rs} \,]\!] \tag{2}$$

To prove this equation, we first show a lemma.

Lemma 1. *Let f be a function, t be a computational tree, and i be an index. Then, the following equation holds.*

$$\mathit{eval}_T\ (\mathit{comp}\ f\ t)\ i = f\ (\mathit{eval}_T\ t\ i)$$

Proof. This is shown by induction on *Tree*.

The *Node* case:

$$\begin{aligned} \mathit{eval}_T\ (\mathit{comp}\ f\ (\mathit{Node}\ g\ l\ r))\ i = \quad & \{\ \text{definition of } \mathit{comp}\ \} \\ & \mathit{eval}_T\ (\mathit{Node}\ (f \circ g)\ l\ r)\ i \\ = \quad & \{\ \text{definition of } \mathit{eval}_T\ \} \\ & (f \circ g)\ (\mathit{eval}_T\ l\ i, \mathit{eval}_T\ r\ i) \\ = \quad & \{\ \text{function composition}\ \} \\ & f\ (g\ (\mathit{eval}_T\ l\ i, \mathit{eval}_T\ r\ i)) \\ = \quad & \{\ \text{definition of } \mathit{eval}_T\ \} \\ & f\ (\mathit{eval}_T\ (\mathit{Node}\ g\ l\ r)\ i) \end{aligned}$$

The $Leaf_v$ case:

$$\begin{aligned}
eval_T\ (comp\ f\ (Leaf_v\ g\ v))\ i = &\quad \{\ \text{definition of } comp\ \} \\
&\ eval_T\ (Leaf_v\ (f \circ g)\ v)\ i \\
= &\quad \{\ \text{definition of } eval_T\ \} \\
&\ (f \circ g)\ (eval_V\ v\ i) \\
= &\quad \{\ \text{function composition}\ \} \\
&\ f\ (g\ (eval_V\ v\ i)) \\
= &\quad \{\ \text{definition of } eval_T\ \} \\
&\ f\ (eval_T\ (Leaf_v\ g\ v)\ i)
\end{aligned}$$

The $Leaf_c$ case:

$$\begin{aligned}
eval_T\ (comp\ f\ (Leaf_c\ c))\ i = &\quad \{\ \text{definition of } comp\ \} \\
&\ eval_T\ (Leaf_c\ (f\ c))\ i \\
= &\quad \{\ \text{definition of } eval_T\ \} \\
&\ f\ c \\
= &\quad \{\ \text{definition of } eval_T\ \} \\
&\ f\ (eval_T\ (Leaf_c\ c)\ i)
\end{aligned}$$

□

Now, we show the equation (2). In the following, l = length ls, r = length rs, and $idces = [l..(n-r-1)]$.

$$\begin{aligned}
&eval_P\ (\underline{\mathsf{map}}\ f\ prog) \\
= &\quad \{\ \text{definition of } eval_P\ \} \\
&\mathsf{map}\ f\ (eval_P\ prog) \\
= &\quad \{\ \text{induction hypothesis}\ \} \\
&\mathsf{map}\ f\ (eval\ [\![ls, zms, rs]\!]) \\
= &\quad \{\ \text{definition of } eval\ \} \\
&\mathsf{map}\ f\ (\mathsf{map}\ eval_{T0}\ ls +\!\!+ \mathsf{map}\ (eval_T\ zms)\ idces +\!\!+ \mathsf{map}\ eval_{T0}\ rs) \\
= &\quad \{\ \text{definition of map and its distributivity: } \mathsf{map}\ h \circ \mathsf{map}\ g = \mathsf{map}\ (h \circ g)\ \} \\
&\mathsf{map}\ (f \circ eval_{T0})\ ls +\!\!+ \mathsf{map}\ (f \circ eval_T\ zms)\ idces +\!\!+ \mathsf{map}\ (f \circ eval_{T0})\ rs \\
= &\quad \{\ \text{Lemma 1 and definition of } eval_{T0}\ \} \\
&\mathsf{map}\ (eval_{T0} \circ comp\ f)\ ls +\!\!+ \mathsf{map}\ (eval_T(comp\ f\ zms))\ idces +\!\!+ \mathsf{map}\ (eval_{T0} \circ comp\ f)\ rs \\
= &\quad \{\ \text{distributivity of map and definition of } eval\ \} \\
&eval\ [\![\mathsf{map}\ (comp\ f)\ ls, comp\ f\ zms, \mathsf{map}\ (comp\ f)\ rs]\!]
\end{aligned}$$

Thus, the equation (2) holds.

A.3 Inductive Cases for $\underline{\mathsf{shift}}_{\ll}$ and $\underline{\mathsf{shift}}_{\gg}$

What we have to prove is the following equations for an element e and a program $prog$. Here, $compile\ prog = [\![ls, zms, rs]\!]$.

$$eval_P\ (\underline{\mathsf{shift}}_{\ll}\ e\ prog) = eval\ [\![\mathrm{tail}\ ls, slide\ (-1)\ zms, rs +\!\!+ [Leaf_c\ e]]\!] \tag{3}$$

$$eval_P\ (\underline{\mathsf{shift}}_{\gg}\ e\ prog) = eval\ [\![[Leaf_c\ e] +\!\!+ ls, slide\ 1\ zms, \mathrm{init}\ rs]\!] \tag{4}$$

To prove this equation, we first show a lemma.

Lemma 2. *Let d be an integer, i be an index, and t be a computational tree. Then, the following equation holds.*

$$eval_T\ (slide\ d\ t)\ i = eval_T\ t\ (i-d) \tag{5}$$

Proof. This is shown by induction on *Tree* and *Var*.

The *Node* case:

$$
\begin{aligned}
eval_T\ (slide\ d\ (Node\ f\ l\ r))\ i = &\quad \{\ \text{definition of } slide\ \}\\
&\ eval_T\ (Node\ f\ (slide\ d\ l)\ (slide\ d\ r)))\ i\\
= &\quad \{\ \text{definition of } eval_T\ \}\\
&\ f\ (eval_T\ (slide\ d\ l)\ i, eval_T\ (slide\ d\ r)\ i)\\
= &\quad \{\ \text{induction hypothesis}\ \}\\
&\ f\ (eval_T\ l\ (i-d), eval_T\ r\ (i-d))\\
= &\quad \{\ \text{definition of } eval_T\ \}\\
&\ eval_T\ (Node\ f\ l\ r)\ (i-d)
\end{aligned}
$$

The $Leaf_v$ with *Var* case:

$$
\begin{aligned}
eval_T\ (slide\ d\ (Leaf_v\ f\ (Var\ x\ s)))\ i = &\quad \{\ \text{definition of } slide\ \}\\
&\ eval_T\ (Leaf_v\ f\ (Var\ x\ (s+d)))\ i\\
= &\quad \{\ \text{definition of } eval_T \text{ and } eval_V\ \}\\
&\ f\ (\text{at}\ (i-(s+d))\ x)\\
= &\quad \{\ \text{arithmetic}\ \}\\
&\ f\ (\text{at}\ ((i-d)-s)\ x)\\
= &\quad \{\ \text{definition of } eval_T \text{ and } eval_V\ \}\\
&\ eval_T\ (Leaf_v\ f\ (Var\ x\ s))\ (i-d)
\end{aligned}
$$

Since other cases of $Leaf_v$ and the case of $Leaf_c$ ignore the index i, the equation (5) holds in these cases.

Thus, the equation (5) holds. □

Now, we show the equation (3). In the following, $l = \text{length}\ ls$ and $r = \text{length}\ rs$. First, we assume ls is not empty, i.e. $l > 0$.

$$
\begin{aligned}
&eval_P\ (\underline{\mathsf{shift}}_{\ll}\ e\ prog)\\
= &\quad \{\ \text{definition of } eval_P\ \}\\
&\mathsf{shift}_{\ll}\ e\ (eval_P\ prog)\\
= &\quad \{\ \text{induction hypothesis}\ \}\\
&\mathsf{shift}_{\ll}\ e\ (eval\ [\![\ ls, zms, rs\ ]\!])\\
= &\quad \{\ \text{definition of } eval\ \}\\
&\mathsf{shift}_{\ll}\ e\ (\mathsf{map}\ eval_{T0}\ ls +\!\!+ \mathsf{map}\ (eval_T\ zms)\ [l..(n-r-1)] +\!\!+ \mathsf{map}\ eval_{T0}\ rs)\\
= &\quad \{\ \text{definition of } \mathsf{shift}_{\ll}, \text{and } ls \text{ being not empty}\ \}\\
&\text{tail}\ (\mathsf{map}\ eval_{T0}\ ls) +\!\!+ \mathsf{map}\ (eval_T\ zms)\ [l..(n-r-1)] +\!\!+ \mathsf{map}\ eval_{T0}\ rs +\!\!+ [e]\\
= &\quad \{\ \text{definition of } \mathsf{map} \text{ and } eval_{T0}\ \}\\
&\mathsf{map}\ eval_{T0}\ (\text{tail}\ ls) +\!\!+ \mathsf{map}\ (eval_T\ zms)\ (\mathsf{map}\ (+1)\ [(l-1)..(n-(r+1)-1)])\\
&\qquad\qquad +\!\!+ \mathsf{map}\ eval_{T0}\ (rs +\!\!+ [Leaf_c\ e])\\
= &\quad \{\ \text{distributivity of } \mathsf{map}, \text{and Lemma 2}\ \}\\
&\mathsf{map}\ eval_{T0}\ (\text{tail}\ ls) +\!\!+ \mathsf{map}\ (eval_T\ (slide\ (-1)\ zms))\ [l'..(n-r'-1)]\\
&\qquad\qquad +\!\!+ \mathsf{map}\ eval_{T0}\ (rs +\!\!+ [Leaf_c\ e])\\
&\quad \textbf{where} l' = \text{length tail}\ ls,\ r' = \text{length}\ (rs +\!\!+ [Leaf_c\ e])\\
= &\quad \{\ \text{definition of } eval\ \}\\
&eval\ [\![\ \text{tail}\ ls, slide\ (-1)\ zms, rs +\!\!+ [Leaf_c\ e]\ ]\!]
\end{aligned}
$$

Next, we assume $ls = [\,]$.

$$
\begin{aligned}
& \mathit{eval}_P\ (\underline{\mathsf{shift}}_{\ll}\ e\ \mathit{prog}) \\
= \quad & \{\ \text{the same as the previous calculation}\ \} \\
& \mathsf{shift}_{\ll}\ e\ (\mathsf{map}\ \mathit{eval}_{T0}\ ls +\!\!+ \mathsf{map}\ (\mathit{eval}_T\ \mathit{zms})\ [l..(n-r-1)] +\!\!+ \mathsf{map}\ \mathit{eval}_{T0}\ rs) \\
= \quad & \{\ \text{definition of } \mathsf{shift}_{\ll}\text{, and } ls \text{ being empty}\ \} \\
& \mathrm{tail}\ (\mathsf{map}\ (\mathit{eval}_T\ \mathit{zms})\ [0..(n-r-1)]) +\!\!+ \mathsf{map}\ \mathit{eval}_{T0}\ rs +\!\!+ [e] \\
= \quad & \{\ \text{definition of tail}\ \} \\
& \mathsf{map}\ (\mathit{eval}_T\ \mathit{zms})\ ([1..(n-r-1)]) +\!\!+ \mathsf{map}\ \mathit{eval}_{T0}\ rs +\!\!+ [e] \\
= \quad & \{\ \text{definition of map and } \mathit{eval}_{T0}\ \} \\
& \mathsf{map}\ (\mathit{eval}_T\ \mathit{zms})\ (\mathsf{map}\ (+1)\ [0..(n-(r+1)-1)]) +\!\!+ \mathsf{map}\ \mathit{eval}_{T0}\ (rs +\!\!+ [\mathit{Leaf}_c\ e]) \\
= \quad & \{\ \text{distributivity of map, Lemma 2, and tail } ls = [\,]\ \} \\
& \mathsf{map}\ \mathit{eval}_{T0}\ (\mathrm{tail}\ ls) +\!\!+ \mathsf{map}\ (\mathit{eval}_T\ (\mathit{slide}\ (-1)\ \mathit{zms}))\ [l'..(n-r'-1)] \\
& \qquad\qquad +\!\!+ \mathsf{map}\ \mathit{eval}_{T0}\ (rs +\!\!+ [\mathit{Leaf}_c\ e]) \\
& \quad \mathbf{where} l' = \mathrm{tail}\ ls,\ r' = \mathrm{length}\ (rs +\!\!+ [\mathit{Leaf}_c\ e]) \\
= \quad & \{\ \text{definition of } \mathit{eval}\ \} \\
& \mathit{eval}\ [\![\, \mathrm{tail}\ ls, \mathit{slide}\ (-1)\ \mathit{zms}, rs +\!\!+ [\mathit{Leaf}_c\ e] \,]\!]
\end{aligned}
$$

Thus, the equation (3) holds.

The equation (4) is shown similarly.

A.4 Inductive Case for zip

What we have to prove is the following equation for two programs $prog_1$ and $prog_2$. Here, $\mathit{compile}\ prog_1 = [\![\, ls_1, zms_1, rs_1 \,]\!]$ and $\mathit{compile}\ prog_2 = [\![\, ls_2, zms_2, rs_2 \,]\!]$.

$$
\begin{aligned}
& \mathit{eval}_P\ (\underline{\mathsf{zip}}\ prog_1\ prog_2) = \mathit{eval}\ [\![\, ls, zms, \mathrm{rev}\ rs \,]\!] \\
& \quad \mathbf{where}\ zms = \mathit{Node}\ \mathrm{id}\ zms_1\ zms_2 \\
& \qquad\quad ls = \mathit{trim}\ \mathit{FromL}\ ls_1\ ls_2\ zms_1\ zms_2 \\
& \qquad\quad rs = \mathit{trim}\ \mathit{FromR}\ (\mathrm{rev}\ rs_1)\ (\mathrm{rev}\ rs_2)\ zms_1\ zms_2
\end{aligned}
\tag{6}
$$

To prove this equation, we first show a lemma.

Lemma 3. *Let i be an index, and t be a computational tree. Then, the following equation holds.*

$$
\mathit{eval}_{T0}\ (\mathit{inst}\ \mathit{FromL}\ i) = \mathit{eval}_T\ t\ i \tag{7}
$$

Proof. This is shown by induction on *Tree* and *Var*.

The *Node* case:

$$
\begin{aligned}
\mathit{eval}_{T0}\ (\mathit{inst}\ \mathit{FromL}\ (\mathit{Node}\ f\ l\ r)\ i) = \quad & \{\ \text{definition of } \mathit{inst}\ \} \\
& \mathit{eval}_{T0}\ (\mathit{Node}\ f\ (\mathit{inst}\ \mathit{FromL}\ l\ i)\ (\mathit{inst}\ \mathit{FromL}\ r\ i)) \\
= \quad & \{\ \text{definition of } \mathit{eval}_{T0}\ \} \\
& f\ (\mathit{eval}_{T0}\ (\mathit{inst}\ \mathit{FromL}\ l\ i), \mathit{eval}_{T0}\ (\mathit{inst}\ \mathit{FromL}\ r\ i)) \\
= \quad & \{\ \text{induction hypothesis}\ \} \\
& f\ (\mathit{eval}_T\ l\ i, \mathit{eval}_T\ r\ i) \\
= \quad & \{\ \text{definition of } \mathit{eval}_T\ \} \\
& \mathit{eval}_T\ (\mathit{Node}\ f\ l\ r)\ i
\end{aligned}
$$

The $Leaf_v$ with Var case:

$$\begin{aligned}
eval_{T0}\ (inst\ FromL\ (Leaf_v\ f\ (Var\ x\ s))\ i) = &\quad \{\ \text{definition of } inst\ \}\\
&\ eval_{T0}\ (Leaf_v\ f\ (Fix\ x\ (-s+i)\ FromL))\\
= &\quad \{\ \text{definition of } eval_{T0}\ \}\\
&\ f\ (\text{at}\ (-s+i)\ x)\\
= &\quad \{\ \text{definition of } eval_T\ \}\\
&\ eval_T\ (Var\ x\ s)\ i
\end{aligned}$$

Since other cases of $Leaf_v$ and the case of $Leaf_c$ merely return the given tree and this tree ignores the index, the equation (7) holds in these cases.

Thus, the equation (7) holds. □

We also use the following lemma.

Lemma 4. *Let i be an index, and t be a computational tree. Then, the following equation holds.*

$$eval_{T0}\ (inst\ FromR\ i) = eval_T\ t\ (n-1-i)$$

Proof. Similar to the proof of Lemma 3. □

In the following, $l_1 = \text{length}\ ls_1$, $r_1 = \text{length}\ rs_1$, $l_2 = \text{length}\ ls_2$ and $r_2 = \text{length}\ rs_2$. First, we assume $l_1 \geq l_2$ and $r_1 \geq r_2$.

$$\begin{aligned}
&eval_P\ (\underline{\mathsf{zip}}\ prog_1\ prog_2)\\
=&\quad \{\ \text{definition of } eval_P\ \}\\
&\mathsf{zip}\ (eval_P\ prog_1, eval_P\ prog_2)\\
=&\quad \{\ \text{induction hypothesis}\ \}\\
&\mathsf{zip}\ (eval\ [\![\ ls_1, zms_1, rs_1\ ]\!])\ (eval\ [\![\ ls_2, zms_2, rs_2\ ]\!])\\
=&\quad \{\ \text{definition of } eval \text{ and } \mathsf{zip}\text{, letting } ls_1 = ls_{11} +\!\!+ ls_{12} \text{ and } rs_1 = rs_{11} +\!\!+ rs_{12}\ \}\\
&\mathsf{zip}\ (\mathsf{map}\ eval_{T0}\ ls_{11})\ (\mathsf{map}\ eval_{T0}\ ls_2)\\
&\ +\!\!+ \mathsf{zip}\ (\mathsf{map}\ eval_{T0}\ ls_{11})\ (\mathsf{map}\ (eval_T\ zms_2)\ [l_2..(l_1-1)])\\
&\ +\!\!+ \mathsf{zip}\ (\mathsf{map}\ (eval_T\ zms_1)\ [l_1..(n-r_1-1)])\ (\mathsf{map}\ (eval_T\ zms_2)\ [l_1..(n-r_1-1)])\\
&\ +\!\!+ \mathsf{zip}\ (\mathsf{map}\ eval_{T0}\ rs_{11})\ (\mathsf{map}\ (eval_T\ zms_2)\ [(n-r_1)..(n-r_2-1)])\\
&\ +\!\!+ \mathsf{zip}\ (\mathsf{map}\ eval_{T0}\ rs_{12})\ (\mathsf{map}\ eval_{T0}\ rs_2)\\
=&\quad \{\ \text{Lemma 3 and Lemma 4}\ \}\\
&\mathsf{zip}\ (\mathsf{map}\ eval_{T0}\ ls_{11})\ (\mathsf{map}\ eval_{T0}\ ls_2)\\
&\ +\!\!+ \mathsf{zip}\ (\mathsf{map}\ eval_{T0}\ ls_{11})\ (\mathsf{map}\ eval_{T0}\ (\mathsf{map}\ (inst\ FromL\ zms_2)\ [l_2..(l_1-1)])\\
&\ +\!\!+ \mathsf{zip}\ (\mathsf{map}\ (eval_T\ zms_1)\ [l_1..(n-r_1-1)])\ (\mathsf{map}\ (eval_T\ zms_2)\ [l_1..(n-r_1-1)])\\
&\ +\!\!+ \mathsf{zip}\ (\mathsf{map}\ eval_{T0}\ rs_{11})\ (\mathsf{map}\ eval_{T0}\ (\mathsf{map}\ (inst\ FromR\ zms_2)\ [(r_1-1)..r_2])\\
&\ +\!\!+ \mathsf{zip}\ (\mathsf{map}\ eval_{T0}\ rs_{12})\ (\mathsf{map}\ eval_{T0}\ rs_2)\\
=&\quad \{\ \text{introducing } (Node\ \text{id}), \text{and definition of } eval \text{ and } eval_{T0}\ \}\\
&\mathsf{map}\ eval_{T0}\ (\mathsf{zipWith}\ (Node\ \text{id})\ ls_{11}\ ls_2)\\
&\ +\!\!+ \mathsf{map}\ eval_{T0}\ (\mathsf{zipWith}\ (Node\ \text{id})\ ls_{11}\ (\mathsf{map}\ (inst\ FromL\ zms_2)\ [l_2..(l_1-1)]))\\
&\ +\!\!+ \mathsf{map}\ (eval_T\ (Node\ \text{id}\ zms_1\ zms_2))\ [l_1..(n-r_1-1)]\\
&\ +\!\!+ \mathsf{map}\ eval_{T0}\ (\mathsf{zipWith}\ (Node\ \text{id})\ rs_{11}\ (\mathsf{map}\ (inst\ FromR\ zms_2)\ [(r_1-1)..r_2]))\\
&\ +\!\!+ \mathsf{map}\ eval_{T0}\ (\mathsf{zipWith}\ (Node\ \text{id})\ rs_{12}\ rs_2)
\end{aligned}$$

$$
\begin{aligned}
&= \quad \{ \text{ combining edge elements } \} \\
&\quad \mathsf{map}\ \mathit{eval}_{T0}\ (\mathsf{zipWith}\ (\mathit{Node}\ \mathrm{id})\ \mathit{ls}_1\ (\mathit{ls}_2 +\!\!+ \mathsf{map}\ (\mathit{inst}\ \mathit{FromL}\ \mathit{zms}_2)\ [l_2..(l_1-1)])) \\
&\quad +\!\!+ \mathsf{map}\ (\mathit{eval}_T\ (\mathit{Node}\ \mathrm{id}\ \mathit{zms}_1\ \mathit{zms}_2))\ [l_1..(n-r_1-1)] \\
&\quad +\!\!+ \mathsf{map}\ \mathit{eval}_{T0}\ (\mathsf{zipWith}\ (\mathit{Node}\ \mathrm{id})\ \mathit{rs}_1\ (\mathsf{map}\ (\mathit{inst}\ \mathit{FromR}\ \mathit{zms}_2)\ [(r_1-1)..r_2] +\!\!+ \mathit{rs}_2)) \\
&= \quad \{ \text{ definition of } \mathit{eval} \text{ and } \mathit{trim} \ \} \\
&\quad \mathit{eval}\ [\![\, \mathit{ls}, \mathit{zms}, \mathrm{rev}\ \mathit{rs}\,]\!] \\
&\qquad \textbf{where}\ \mathit{zms} = \mathit{Node}\ \mathrm{id}\ \mathit{zms}_1\ \mathit{zms}_2 \\
&\qquad\qquad \mathit{ls} = \mathit{trim}\ \mathit{FromL}\ \mathit{ls}_1\ \mathit{ls}_2\ \mathit{zms}_1\ \mathit{zms}_2 \\
&\qquad\qquad \mathit{rs} = \mathit{trim}\ \mathit{FromR}\ (\mathrm{rev}\ \mathit{rs}_1)\ (\mathrm{rev}\ \mathit{rs}_2)\ \mathit{zms}_1\ \mathit{zms}_2
\end{aligned}
$$

The other cases ($l_1 \geq l_2 \wedge r_1 < r_2$, $l_1 < l_2 \wedge r_1 \geq r_2$ and $l_1 < l_2 \wedge r_1 < r_2$) are similarly shown.

Thus, the equation (6) holds.