Electronic Preprint for Journal of Information Processing Vol.27

Regular Paper

Using Algebraic Properties and Function Fusion to Evaluate Tree Accumulations in Parallel

Akimasa Morihata[1,a)]

Received: September 28, 2018, Accepted: December 25, 2018

Abstract: Parallel evaluation of tree processing using accumulation parameters tends to be difficult because the accumulation parameters may introduce data dependencies between computations for subtrees. Some proposals have broken these data dependencies by using algebraic properties such as associativity and commutativity, but, none has achieved both the capability of complex tree traversals like attribute grammars and a theoretical guarantee of parallel speedup. This paper proposes a tree processing language based on macro tree transducers and provides a parallel evaluation algorithm for programs written in the language. The language can express complex accumulations like attribute grammars, and moreover, the number of parallel computational steps for evaluation is proportional to the height of the input tree. This paper also shows that combining the proposed approach with function fusion for macro tree transducers leads to improvement in the parallel computational complexity. Although comparable complexity improvement can be obtained from known parallel algorithms, the proof and parallel evaluation algorithm here are remarkably simpler.

Keywords: parallel evaluation, accumulation parameters, macro tree transducers, function fusion, semiring, attribute evaluation

1. Introduction

Parallel computing has become popular because even commodity computers contain multicore CPUs. Purely functional programming is commonly regarded as a promising approach for parallel computing. Because of the absence of side effects, independent subexpressions can be evaluated in parallel. For example, the following function *sum* can naturally calculate the summations of values in subtrees in parallel. (The notation below is similar to that of Haskell [27].)

$sum\ (\mathsf{Tip}\ n) = n$

$sum\ (\mathsf{Fork}\ l\ r) = sum\ l + sum\ r$

Nevertheless, purely functional programs often contain an insufficient quantity of independent subexpressions. Even huge data processing, which requires parallel computing, may involve only a few independent subexpressions if *accumulation parameters* are used. For example, consider the following function sum_{acc}.

$sum_{acc}\ (\mathsf{Tip}\ n)\ y = n + y$

$sum_{acc}\ (\mathsf{Fork}\ l\ r)\ y = sum_{acc}\ l\ (sum_{acc}\ r\ y)$

This function calculates the summation by using an accumulation parameter, y. The computations for the left and the right subtrees are not independent, because the computation for the left uses the result of the computation for the right. Therefore, the function appears unsuitable for parallel computing in this case.

The objective of this paper is to provide a method of parallel evaluation for tree processing using accumulation parameters, such as sum_{acc}. The work especially focuses on algebraic properties such as associativity and commutativity. For instance, *sum*, which is more suitable for parallel computing, can be used instead of sum_{acc}: the order of summing up values does not matter because of the associativity and commutativity of addition.

There have been several proposals for parallel tree processing using algebraic properties [1], [12], [21], [22], [30], [32]. To the best of the author's knowledge, however, none satisfies the following three requirements.

Flexible Tree Traversal

Some proposals focus on a few tree traversal patterns, such as bottom-up, top-down, or depth-first traversal. This approach is suitable for (nearly) linear data structures, such as lists or XML data that represent sets of items. Trees, however, may not be nearly linear. They may be used as intermediate data structures, and then their shapes (e.g., the height, width, degree of balance, etc.) depend on the application. As a result, the traversal patterns can be complicated — possibly, some subtrees are neglected or processed in the reverse order, and so on. Flexibility of traversal is essential for tree processing and thus cannot be disregarded.

Summary for Every Substructure

Because trees are often intermediate data structures, we would often like to calculate a summary for every substructure rather than a summary for the whole tree. For example, in the variable live range analysis (on abstract syntax trees), each variable should be associated with its live range. In theory, it is usually not difficult to extend parallel algorithms for summarization to those that calculate a summary for every substructure (see Remark 3.1 in Abrahamson et al. [1], for instance). Yet, for the case of complex accumulative tree processing, no such algorithms have explicitly been shown.

[1] The University of Tokyo, Bunkyo, Tokyo 113–8654, Japan
[a)] morihata@graco.c.u-tokyo.ac.jp

Guarantee of Parallel Speedup

The naive approach that parallelizes apparently parallelizable parts is not acceptable. As accumulative tree processing is generally not suitable for parallel evaluation, such a naive approach would probably result in poor parallel speedup. As Amdahl's law explains, to achieve significant parallel speedup, most computations should be parallelized. In other words, the parallelization should improve the asymptotic computational complexity. Some existing approaches provide a guarantee of parallel speedup, but the guarantee tends to make the parallel algorithm more complex and less expressive.

To satisfy these requirements, this paper focuses on *macro tree transducers* (MTTs) [5], [9]. MTTs are simple but more expressive than attributed tree transducers (ATTs), which are tree transformations modeled by attribute grammars. Therefore, MTTs can express complex tree traversals like attribute grammars.

MTTs do not allow arbitrary accumulative computations. In MTTs, values bound to accumulation parameters can be used but cannot be examined by, for example, conditionals or pattern matching. This restriction seems suitable for parallel computing. Roughly speaking, while accumulation parameters are computed by other threads, the succeeding computation, which will not be seriously affected by the accumulation parameters, may be able to speculatively processed. For this reason, it is expected that MTTs would be good candidates for resolving the trade-off between expressiveness and suitability for parallel computing.

Nevertheless, MTTs are tree-to-tree transformations and cannot be used for usual computations such as additions and multiplications. In addition, MTTs cannot naturally express summarization for substructures. To resolve these issues, while the approach here follows the syntactic restriction of MTTs, it semantically uses *semiring* operators and also provides a support for summarization of substructures.

Several properties of MTTs are known. This paper especially focuses on *fusion transformations* [5], [9], [11], [13] and uses them to improve the complexity of parallel tree processing. The paper first provides a parallel evaluation algorithm and shows that its number of parallel evaluation steps is proportional to the height of the input tree. Then, Shunt trees [25] is applied for improving the complexity. The shunt tree representation of a tree t contains the same information as t, and yet its height is logarithmic in the height of t. Let g be a transformation that restores the original tree from its shunt tree representation. For tree processing f, the fusion transformation for MTTs derives an MTT equivalent to the composition of f and g, $f \circ g$. The complexity of the obtained MTT is proportional to the height of the shunt tree representation, making it logarithmic in the size of the original input. This approach avoids dealing with complex parallel algorithms and thereby leads to a significantly simpler correctness proof and complexity analysis as compared to using known parallel algorithms.

The three major contributions are the following:

- A programming language for parallel tree processing: MTTs are applied to provide a programming language suitable for parallel evaluation. The language can naturally express attribute-grammar-like complex accumulations, as well as summarization for substructures.
- A parallel evaluation algorithm with a complexity guarantee: The paper provides a simple parallel evaluation algorithm, whose number of parallel evaluation steps is proportional to the height of the input tree.
- Complexity improvement through fusion transformation: The paper shows that a combination of the proposed parallel algorithm and fusion transformation significantly improves computational complexity.

The paper is organized as follows. Section 2 reviews basic notations about semirings and MTTs. The programming language for parallel evaluation is defined in Section 3. The parallel evaluation algorithm is provided in Section 4. Section 5 discusses the improvements brought by the fusion transformation. Finally the paper discusses related work in Section 6 and concludes in Section 7.

2. Semiring and Macro Tree Transducer

2.1 Semirings and Linear Polynomials

A semiring abstracts the cooperation of two related operations such as addition and multiplication.

Definition 1 A semiring $(S, \oplus, \otimes, \overline{0}, \overline{1})$ is a five-tuple, where S is a set of values, $\oplus$ and $\otimes$ are binary operators over S, $\overline{0}$ and $\overline{1}$ are elements of S, and the following properties hold.

$a \oplus (b \oplus c) = (a \oplus b) \oplus c$	{ associativity of $\oplus$ }
$a \oplus b = b \oplus a$	{ commutativity of $\oplus$ }
$a \oplus \overline{0} = \overline{0} \oplus a = a$	{ unit of $\oplus$ }
$a \otimes (b \otimes c) = (a \otimes b) \otimes c$	{ associativity of $\otimes$ }
$a \otimes \overline{1} = \overline{1} \otimes a = a$	{ unit of $\otimes$ }
$a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$	{ distributivity }
$a \otimes \overline{0} = \overline{0} \otimes a = \overline{0}$	{ zero }

Examples of semirings include the following: addition and multiplication of integers, $(\mathbb{Z}, +, \times, 0, 1)$; addition with the binary maximum operator $\uparrow$, $(\mathbb{Z} \cup \{-\infty\}, \uparrow, +, -\infty, 0)$; and computations over languages (i.e., sets of strings), $(2^{\Sigma^*}, \bullet, \cup, \{\epsilon\}, \emptyset)$ where Σ^* is a set of strings using alphabet Σ, ϵ is the empty string, and $\bullet$ is the language concatenation operator defined by $S_1 \bullet S_2 = \{w_1 w_2 \mid w_1 \in S_1, w_2 \in S_2\}$.

For a semiring $\mathcal{R} = (S, \oplus, \otimes, \overline{0}, \overline{1})$, $\mathcal{R}$ and S may be used interchangeably if the meaning is apparent from the context. For example, we may write $s \in \mathcal{R}$, i.e., "s is an element of $\mathcal{R}$," instead of $s \in S$.

Given a set of variables, $Y = \{y_1, y_2, \ldots, y_m\}$, polynomials over $\mathcal{R} = (S, \oplus, \otimes, \overline{0}, \overline{1})$ are naturally defined. In particular, a polynomial of the following form, where $s_0, s_1, \ldots, s_m \in S$,

$$s_0 \oplus (s_1 \otimes y_1) \oplus \cdots \oplus (s_m \otimes y_m),$$

is called a *left-linear polynomial* over $(\mathcal{R}, Y)$ [*1]. We need not mention $\mathcal{R}$ and Y if they are clear from the context. Note also the potential confusion between the linearity of a polynomial and the linearity of a variable. To avoid the confusion, the latter type of linearity is called *single-use* in this paper.

[*1] While only left-linear polynomials are considered in this paper, the same discussion can be applied to the dual, namely *right linear polynomials*.

2.2 Macro Tree Transducers

A *macro tree transducer* (MTT) models a tree transformation by recursive functions. This paper considers total deterministic MTTs, which can be understood as a subset of first-order functional programs.

The syntax of MTTs is defined below, where f, each v_i ($v \in \{x, y, z\}, i \in \mathbb{N}$), σ, and δ are metavariables that respectively denote the name of a recursive function, a variable, an input tree constructor, and an output tree constructor. Each function, as well as each constructor, takes a certain number of arguments, called the arity.

$$
\begin{aligned}
\mathit{prog} &::= e \ \textbf{where}\ \mathit{decl} \cdots \mathit{decl} \\
\mathit{decl} &::= f\ (\sigma\ x_1 \cdots x_n)\ y_1 \cdots y_m = e \\
e &::= y_i \mid f\ x_i\ e \cdots e \mid \delta\ e \cdots e \\
&\quad \mid \textbf{let}\ z_i = e\ \textbf{in}\ e \mid z_i
\end{aligned}
$$

A program consists of an initial expression and a set of recursive function definitions. The initial expression contains a special variable x_1 that binds the input tree. Functions are defined by mutual recursion and traverse the input tree structural-recursively. Each function definition consists of $\langle f, \sigma\rangle$-rules, which are the equations of the form $f\ (\sigma\ x_1 \cdots x_n)\ y_1 \cdots y_m = e$. Note that only the first argument, which binds a subtree of the input, is the subject of pattern matching. To syntactically clarify this restriction, we use different names for the variables binding subtrees of the inputs from the others. This naming convention may be violated for better readability of some examples.

MTTs in this paper can introduce local variables through **let** bindings, but recursive bindings, i.e., **letrec**, are not allowed. The **let** bindings may appear to be syntactic sugar. Later, special semantics are provided to support easy expression of substructure summarization. Each variable name introduced by a **let** binding is assumed unique.

The semantics of MTTs is defined by reduction, as usual. We only deal with type-correct, error-free MTTs. Every used variable should be defined, and the $\langle f, \sigma\rangle$-rules should be exhaustive and non-overlapping. Note that the evaluation of an MTT terminates because the size of the first argument decreases.

An MTT is said to be *single-use restricted* [20] if (i) each variable y_i or z_i is used at most once in each $\langle f, \sigma\rangle$-rule, and (ii) for each σ, each f, and each x_i, at most one $\langle f', \sigma\rangle$-rule contains a recursive call of the form of $f\ x_i\ e \cdots e$, and the recursive call cannot appear more than once in the rule. In evaluating a single-use restricted MTT, each recursive function visits each subtree at most once, and the result of the recursive call is used at most once. Note, however, that different recursive functions may visit the same subtree.

In the following, if an MTT contains only one recursive function, it is denoted by the name of the recursive function.

Examples

The following MTT *toBin* outputs a complete binary tree of a given height. Each leaf stores the path from the root.

$$
\begin{aligned}
&\mathit{toBin}\ x_1\ \mathsf{Nil} \\
&\quad \textbf{where} \\
&\qquad \mathit{toBin}\ \mathsf{Z}\ y_1 = \mathsf{Tip}\ y_1 \\
&\qquad \mathit{toBin}\ (\mathsf{S}\ x_1)\ y_1 = \mathsf{Fork}\ (\mathit{toBin}\ x_1\ (\mathsf{L}\ y_1)) \\
&\qquad\qquad\qquad\qquad\quad (\mathit{toBin}\ x_1\ (\mathsf{R}\ y_1))
\end{aligned}
$$

This MTT is not single-use restricted: y_1 is used twice in the $\langle \mathit{toBin}, \mathsf{S}\rangle$-rule, and *toBin* visits x_1 twice.

The next MTT finds redexes, i.e., immediately reducible applications, in a lambda expression. Var, Abs, App, and $\overline{\mathsf{App}}$ respectively denote a variable, a lambda abstraction, a function application, and a redex.

$$
\begin{aligned}
&\mathit{redex}\ x_1 \\
&\quad \textbf{where} \\
&\qquad \mathit{redex}\ (\mathsf{Var}\ v) = \mathsf{Var}\ v \\
&\qquad \mathit{redex}\ (\mathsf{Abs}\ v\ e) = \mathsf{Abs}\ v\ (\mathit{redex}\ e) \\
&\qquad \mathit{redex}\ (\mathsf{App}\ e_1\ e_2) = \mathit{check}\ e_1\ (\mathit{redex}\ e_2) \\
&\qquad \mathit{check}\ (\mathsf{Var}\ v)\ y_1 = \mathsf{App}\ (\mathsf{Var}\ v)\ y_1 \\
&\qquad \mathit{check}\ (\mathsf{Abs}\ v\ e)\ y_1 = \overline{\mathsf{App}}\ (\mathsf{Abs}\ v\ (\mathit{redex}\ e))\ y_1 \\
&\qquad \mathit{check}\ (\mathsf{App}\ e_1\ e_2)\ y_1 = \mathsf{App}\ (\mathit{check}\ e_1\ (\mathit{redex}\ e_2))\ y_1
\end{aligned}
$$

Because *check* e_1 occurs in both the $\langle \mathit{redex}, \mathsf{App}\rangle$-rule and the $\langle \mathit{check}, \mathsf{App}\rangle$-rule, this MTT also is not single-use restricted.

The following MTT calculates 2^n for a given natural number n. Both the input and the output are represented by Peano numbers denoted as S and Z.

$$
\begin{aligned}
&\mathit{exp}\ x_1\ \mathsf{Z} \\
&\quad \textbf{where}\ \ \mathit{exp}\ \mathsf{Z}\ y_1 = \mathsf{S}\ y_1 \\
&\qquad\qquad \mathit{exp}\ (\mathsf{S}\ x_1)\ y_1 = \mathit{exp}\ x_1\ (\mathit{exp}\ x_1\ y_1)
\end{aligned}
$$

Again, this MTT is not single-use restricted, because *exp* visits x_1 twice.

As the final example, the following MTT constructs a list from a binary tree by gathering leaves.

$$
\begin{aligned}
&\mathit{flat}\ x_1\ [\,] \\
&\quad \textbf{where}\ \mathit{flat}\ (\mathsf{Tip}\ a)\ y_1 = a : y_1 \\
&\qquad\qquad \mathit{flat}\ (\mathsf{Fork}\ x_1\ x_2)\ y_1 = \mathit{flat}\ x_1\ (\mathit{flat}\ x_2\ y_1)
\end{aligned}
$$

This MTT is single-use restricted.

2.3 Attributed Tree Transducers

An *attributed tree transducer* (ATT) transforms trees according to an attribute grammar [18]. Although the computational models of MTTs and ATTs are different, the following two aspects of correspondence are known.

- Any tree transformation definable by an ATT are definable by an MTT; moreover, the corresponding MTT can be constructed from the ATT [9].
- It is computable whether a tree transformation described by an MTT is definable by an ATT, and if being definable, the corresponding ATT can be constructed [10].

This correspondence indicates that MTTs can express complex tree traversals as attribute grammars.

It is beyond the scope of this paper to explain the correspondence between MTTs and ATTs in detail. The following rough correspondence may be informative for understanding the paper.

Most ATTs correspond to well-presented MTTs [9] [*2]. Roughly speaking, in a well-presented MTT, if a function visits a subtree more than once, then the sets of values passed as accumulation parameters should be identical. For example, the above MTT example consisting of *redex* and *check* is well-presented: although both the $\langle \mathit{redex}, \mathsf{App}\rangle$-rule and the $\langle \mathit{check}, \mathsf{App}\rangle$-rule

[*2] Consistent MTTs [10], a superclass of well-presented MTTs, more closely correspond to ATTs. They are based on a more careful analysis on the uniqueness of accumulation parameters than the case of well-presented MTTs.

contain calls of *check* e_1, the accumulation parameter is the same, *redex* e_2. In contrast, the MTT example *toBin* is not well-presented because subtree x_1 is visited with two different accumulation parameters, L y_1 and R y_1. Similarly the MTT *exp* is not well-presented because subtree x_1 is visited with two different accumulation parameters, *exp* x_1 y_1 and y_1. In fact, neither *toBin* nor *exp* is definable by an ATT.

Note that any single-use restricted MTT is well-presented by definition, and therefore definable by an ATT. In fact, single-use restricted MTTs exactly correspond to single-use restricted ATTs.

From the above-mentioned correspondence, ATTs are regarded as a subclass of MTTs. Moreover, we can implicitly translate MTTs to ATTs and vice versa. Hence, the rest of this paper uses the following conventions:

- "ATTs" is abused to mean MTTs definable by ATTs.
- Single-use restricted MTTs and single-use restricted ATTs are referred to interchangeably.

3. Language for Parallel Tree Processing

3.1 Language Design

The objective is to provide a parallel evaluation algorithm for tree processing specified by MTTs. It is not ideal, however, to consider MTTs themselves as seen from the following MTT [*3]. It models an intra-procedural reachable definition analysis on an abstract syntax tree.

$$
\begin{aligned}
&\mathit{dfa}\ x_1\ \emptyset \\
&\quad \textbf{where} \\
&\qquad \mathit{dfa}\ (\mathsf{Assign}\ v\ e)\ y_1 = (\mathit{remove}\ v\ y_1) \cup \{(v, e)\} \\
&\qquad \mathit{dfa}\ (\mathsf{Seq}\ s_1\ s_2)\ y_1 = \mathit{dfa}\ s_2\ (\mathit{dfa}\ s_1\ y_1) \\
&\qquad \mathit{dfa}\ (\mathsf{If}\ e\ s_1\ s_2)\ y_1 = \mathit{dfa}\ s_1\ y_1 \cup \mathit{dfa}\ s_2\ y_1 \\
&\qquad \mathit{dfa}\ (\mathsf{While}\ e\ s_1)\ y_1 = \mathit{dfa}\ s_1\ y_1
\end{aligned}
$$

In the program, Assign v e, Seq s_1 s_2, If e s_1 s_2, and while e s_1 respectively denote an assignment statement like $v := e$, a sequential statement like s_1; s_2, a conditional statement like `if` (e) s_1 `else` s_2, and a loop like `while` (e) s_1. Function *remove* v y_1 removes definitions of variable v from the set of definitions y_1.

This MTT is not a reachable definition analysis for the following two reasons. First, an MTT expresses a pure tree-to-tree transformation and therefore cannot express usual computations such as numerical computations. Even though $\cup$ and $\cap$ might appear to be set operations, they are in fact constructors. Although there are practically important pure tree transformations such as XML transformations, most practical tree processing contains usual computations. Therefore, the proposed language should be able to deal with tree processing containing usual computations.

Second, while *dfa* computes the definitions available after executing *all statements*, the objective of the usual reachable definition analysis is to determine the definitions available for *each statement or expression*. This is not specific to reachable definition analysis. Balanced trees or heaps should satisfy their own shape requirement for efficiency. To check these requirements, we must calculate metrics for each subtree, such as the height and the size. Queries (using an XPath expression, for example) should check every node for whether it can be matched with the query formula. It is difficult to express these cases with existing MTT generalizations, such as modular tree transducers [7] and tree transducers with external functions [8], whose syntaxes are similar to that of MTTs but allow operators other than tree constructors.

In summary, a language for parallel tree processing are required to satisfy the following two properties.

- Usual operators such as addition and multiplication are available in it.
- It supports calculating summaries for substructures.

3.2 Language Definition

According to the discussion in the previous subsection, the following design policy is adopted for the proposed parallel tree processing language.

- Compute an interpretation of the output tree by using semiring operators.
- Specify values that should be remembered during recursive calls, and return all such values.

The language assigns an interpretation using semiring operators to each output constructor of an MTT. For example, for reachable definition analysis, we consider bit vectors each of whose bits corresponds to a variable in the program, and interpret the output tree by using a semiring consisting of the bitwise logical OR operator $\vee$ and the bitwise logical AND operator $\wedge$. Here, the output constructors $\cup$, $\cap$, and *remove* v y_1 are interpreted as $\vee$, $\wedge$, and $\neg v \wedge y_1$, respectively, where $\neg v$ is a bit vector with each bit is set to 1 except for the bit corresponding to v. This approach enables the language to express a variety of tree processing without negating the simplicity and theoretical results of MTTs.

The semantics of the language is to gather values assigned at **let** bindings. For example, the following modification for *dfa* enables obtaining the definitions reachable for each assignment.

$$\mathit{dfa}\ (\mathsf{Assign}\ v\ e)\ y_1 = \textbf{let}\ z_1 = y_1\ \textbf{in}\ (\mathit{remove}\ v\ y_1) \cup \{(v, e)\}$$

Although the inserted **let** binding appears useless, in this language it is interpreted as a command to store the values assigned to y_1 for each Assign.

This approach is influenced by attribute grammars. While the final outcome of an attribute grammar is usually a value assigned to a synthesized attribute of the root node, the evaluation of an attribute grammar is commonly understood as a value assignment to every attribute of every node. Because MTTs do not have attribute names, the language instead uses **let** bindings to introduce such names.

The language for parallel tree processing is now defined. To formalize its semantics, we assume that each subtree t of the input tree has a unique ID (such as its address on the heap) $ID(t) \in I$. The set of IDs, I, contains a special ID, $\top$, that corresponds to the initial expression. In the following, Z denotes the set of variables introduced at **let** bindings.

[*3] This program does not perfectly fit the MTT syntax shown in Section 2.2, because the right-hand-side expressions contain subtrees of the input tree, such as v and e. To bridge this gap, we interpret Assign v e as a constructor of arity 0 instead of interpreting Assign as a constructor of arity 2. Similarly, for the outputs, we interpret *remove* v as a constructor of arity 1. This interpretation does not affect the discussion in this paper even though it introduces infinitely many kinds of constructors.

$$\Gamma \triangleright_\eta y_i \to \Gamma(y_i), \emptyset$$

$$\Gamma \triangleright_\eta z_i \to \Gamma(z_i), \emptyset$$

$$\frac{\Gamma \triangleright_\eta e_1 \to v_1, \Delta_1 \qquad \Gamma \cup \{z_i \mapsto v_1\} \triangleright_\eta e_2 \to v_2, \Delta_2}{\Gamma \triangleright_\eta \textbf{let } z_i = e_1 \textbf{ in } e_2 \to v_2, \Delta_1 \cup \Delta_2 \cup \{(\eta, z_i) \mapsto v_1\}}$$

$$\frac{\Gamma \triangleright_\eta e_i \to v_i, \Delta_i \text{ (for } 1 \le i \le m) \qquad [\![\delta]\!]\ v_1 \cdots v_m = v_*}{\Gamma \triangleright_\eta \delta\ e_1 \cdots e_m \to v_*, \bigcup_{1 \le i \le m} \Delta_i}$$

$$\frac{\begin{array}{c}\Gamma(x_i) = \sigma\ t_1 \cdots t_k \qquad \langle f, \sigma\rangle\text{-rule is } f\ (\sigma\ x_1 \cdots x_k)\ y_1 \cdots y_m = e_0 \\ \Gamma \triangleright_\eta e_i \to v_i, \Delta_i \text{ (for } 1 \le i \le m) \\ \{x_1 \mapsto t_1, \ldots, x_k \mapsto t_k, y_1 \mapsto v_1, \ldots, y_m \mapsto v_m\} = \Gamma' \\ \Gamma' \triangleright_{ID(\Gamma(x_i))} e_0 \to v_*, \Delta_0\end{array}}{\Gamma \triangleright_\eta f\ x_i\ e_1 \cdots e_m \to v_*, \bigcup_{0 \le i \le m} \Delta_i}$$

Fig. 1 Semantics of $(\mathcal{M}, \mathcal{R}, [\![\cdot]\!])$.

Definition 2 Given a set of constructors Δ and a semiring $\mathcal{R} = (S, \oplus, \otimes, \overline{0}, \overline{1})$, an interpretation, $[\![\cdot]\!]$, of Δ using $\mathcal{R}$ is said to be *left linear* if $[\![\cdot]\!]$ interprets each constructor $\delta \in \Delta$ of arity k as the following left-linear polynomial with k variables and $k+1$ coefficients, $s_0, s_1, \ldots, s_k \in S$.

$[\![\delta]\!] = \lambda x_1\ x_2 \cdots x_k.\ s_0 \oplus (s_1 \otimes x_1) \oplus \cdots \oplus (s_k \otimes x_k)$

For MTT $\mathcal{M}$ and semiring $\mathcal{R}$, $[\![\cdot]\!]$ is called a left-linear interpretation of $\mathcal{M}$ using $\mathcal{R}$ if $[\![\cdot]\!]$ is a left-linear interpretation using $\mathcal{R}$ for each output constructor of $\mathcal{M}$.

The functions sum_{acc}, discussed in Section 1, and *dfa* can be understood as left-linear interpretations.

Definition 3 Let $\mathcal{M}$ be an MTT whose initial expression is e_0, $\mathcal{R}$ be a semiring, and $[\![\cdot]\!]$ be a left-linear interpretation of $\mathcal{M}$ using $\mathcal{R}$. The semantics of triple $(\mathcal{M}, \mathcal{R}, [\![\cdot]\!])$ for input tree t_0 is to calculate a value $v_* \in \mathcal{R}$ and mapping $\Delta \subseteq ((I \times Z) \to \mathcal{R})$ satisfying

$$\{x_1 \mapsto t_0\} \triangleright_\top e_0 \to v_*, \Delta$$

according to the evaluation rules shown in **Fig. 1**.

Evaluation results in a value v_* and a set Δ of the stored bindings. Δ is a mapping to a value from an input tree ID and a variable name introduced at a **let** binding, so that we can see how the value is calculated and stored.

The evaluation rules are ordinary. The differences from those for MTTs are the use of the interpretation instead of the output constructor and the calculation of the mapping.

Note that all arguments should be evaluated before a recursive function call. Therefore, if a result from a subtree is used as an accumulation parameter for processing another subtree, these subtrees cannot be processed in parallel.

3.3 Well-definedness

The semantics defined above calculates a mapping from a subtree ID and a variable name to a value. There exist MTTs, however, for which the mapping is not well defined.

For instance, consider the following MTT, where the output constructors are interpreted as $[\![\mathsf{Z}]\!] = 0$ and $[\![\mathsf{S}]\!] = \lambda x.\ x + 1$.

exp x_1 Z
  where
    exp Z y_1 = S y_1
    exp (S x_1) y_1 = **let** z_1 = y_1 **in** *exp* x_1 (*exp* x_1 y_1)

Let the input tree be t_0 = S t_1, where t_1 = S Z. Function *exp* is invoked twice for t_1, and the accumulation parameters of the two invocations are respectively Z and S Z. Then, from the semantics shown in Fig. 1, the resulting mapping should be $\{(ID(t_1), z_1) \mapsto 0\} \cup \{(ID(t_1), z_1) \mapsto 1\}$. This mapping is nonsense, however, because two different values are associated with the same key.

For a program written in the proposed language, we assume that a unique value is associated with each key, consisting of subtree and a variable name. For instance, the above-mentioned *exp* is invalid.

There are a few reasons for this assumption. First, as seen in MTT *dfa*, a common application of the language is to calculate summaries for subtrees of the input. In such an application, it would be strange that two different values are assigned to the same subtree. Second, it is not straightforward to refine the semantics to deal with problematic cases. Because possibly very many values can be associated with a key, as in the case of *exp*, it does not seem useful to calculate all associated values unless we can know how each value is obtained. Using output tree IDs instead of input tree IDs is also not useful, because a **let** binding may have no corresponding output constructors, as in the case of *exp*. Third, a large class of MTTs satisfies the assumption. Note that, in an MTT, an expression results in multiple, different values only if multiple, different accumulation parameters are passed. Therefore, any well-presented MTT (more precisely, any ATT) satisfies the assumption. Moreover, even MTTs that are not ATTs can satisfy the assumption. A typical case is the following, in which only the initial expression contains a **let** binding.

let z_1 = *exp* x_1 Z **in** z_1
  where *exp* Z y_1 = S y_1
        exp (S x_1) y_1 = *exp* x_1 (*exp* x_1 y_1)

Clearly, variable z_1 binds only one value. In general, the assumption is fulfilled if each variable stores the final outcome, rather than the trace, of a recursive function call.

Whether an MTT satisfies the assumption would be algorithmically checkable by a method similar to that of checking well-presentedness. Further study is left for a future work.

4. Parallel Evaluation Algorithm

4.1 Overview of Algorithm

This section introduces a parallel evaluation algorithm for the triple $(\mathcal{M}, \mathcal{R}, [\![\cdot]\!])$ and shows that the number of parallel computation steps necessary for evaluation is proportional to the height of the input tree. Achieving this computational cost requires processing independent subtrees in parallel even if a computation of one subtree depends on a result of another subtree via an accumulation parameter. To achieve this, the parallel evaluation algorithm uses two phases, *summarization* and *dependency resolution*.

The objective of the summarization phase is to process independent subtrees as much as possible. For example, suppose that the computation of a subtree essentially corresponds to

$$\lambda y_1, y_2.\ 3 \times (5 + y_2 + 4 \times (y_1 + 2 \times y_2 - 3)),$$

where y_1 and y_2 are variables (i.e., accumulation parameters) that will be bound with results of other subtrees. Then, regardless

of the values of y_1 and y_2, we can simplify the computation and obtain the following summary.

$$\lambda y_1, y_2.\ 12 \times y_1 + 27 \times y_2 - 21$$

In general, such simplification is possible if the computation corresponds to a left-linear polynomial of the semiring. A left-linear polynomial with n variables can be characterized as $n + 1$ coefficients. Therefore, any complex expression can be reduced to a simple linear polynomial if the number of variables is small. The summarization phase performs this simplification.

Next, the dependency resolution phase calculates the final result from the summarizes calculated in the summarization phase. The process is similar to usual evaluation. The difference is that the dependency resolution phase avoids recursive calls and instead uses the summaries from the summarization phase to quickly finish the computation on each node.

4.2 Summarization Phase

The kernel of the summarization phase is simplification of left-linear polynomials. In the following definition of the simplification, we assume that every left-linear polynomial contains the same set of variables. Because we can introduce a variable to a polynomial by associating it with the zero coefficient, this assumption is not a restriction.

Definition 4 Consider expressions defined using a semiring $(S, \oplus, \otimes, \overline{0}, \overline{1})$ and a set of variables $\{y_1, \ldots, y_m\}$. A binary relation $\Rightarrow$, which simplifies such expressions, is defined as follows, where $P = s_0 \oplus (s_1 \otimes y_1) \oplus \cdots \oplus (s_m \otimes y_m)$, $P' = s'_0 \oplus (s'_1 \otimes y_1) \oplus \cdots \oplus (s'_m \otimes y_m)(s_j, s'_j \in S)$, and $s \in S$:

$$P \oplus P' \Rightarrow (s_0 \oplus s'_0) \oplus \left(\bigoplus_{1 \le j \le m} (s_j \oplus s'_j) \otimes y_j \right)$$

$$s \otimes P \Rightarrow (s \otimes s_0) \oplus \left(\bigoplus_{1 \le j \le m} (s \otimes s_j) \otimes y_j \right).$$

The summarization phase can be naturally defined using the simplification relation. In the following, $e_0[e_1/w_1, \ldots, e_m/w_m]$ denotes the expression obtained by substituting every variable w_i in e_0 with an expression e_i, respectively.

Definition 5 Let $\mathcal{M}$ be an MTT whose initial expression is e_0, $\mathcal{R}$ be a semiring, and $[\![\cdot]\!]$ be a left-linear interpretation of $\mathcal{M}$ using $\mathcal{R}$. The summary of triple $(\mathcal{M}, \mathcal{R}, [\![\cdot]\!])$ for input tree t_0 is a left-linear polynomial v_* that satisfies

$$\{x_1 \mapsto t_0\} \triangleright e_0 \dashrightarrow v_*$$

according to the rules shown in **Fig. 2**.

The summarization phase has three characteristics. First, it results in a left-linear polynomial, in which each accumulation parameter y_i becomes a variable. Second, as the interpretation of each output constructor is left-linear, the simplification always results in a left-linear polynomial. Note that an expression obtained by substituting left-linear polynomials for another left-linear polynomial can be simplified to a left-linear polynomial. Third, all subexpressions, especially recursive function calls and their accumulation parameters, can be evaluated in parallel. For example, even for the expression $f_1\ x_1\ (f_2\ x_2)$, the recursive calls of f_1 to subtree x_1 and f_2 to subtree x_2 are simultaneously evaluated.

$$\Gamma \triangleright y_i \dashrightarrow \overline{1} \otimes y_i$$

$$\Gamma \triangleright z_i \dashrightarrow \Gamma(z_i)$$

$$\frac{\Gamma \triangleright e_1 \dashrightarrow v_1 \qquad \Gamma \cup \{z_i \mapsto v_1\} \triangleright e_2 \dashrightarrow v_2}{\Gamma \triangleright \mathbf{let}\ z_i = e_1\ \mathbf{in}\ e_2 \dashrightarrow v_2}$$

$$\frac{\Gamma \triangleright e_i \dashrightarrow v_i\ (\text{for } 1 \le i \le m) \qquad [\![\delta]\!] = \lambda w_1\ w_2 \cdots w_m.\ e_\delta \qquad e_\delta[v_1/w_1, v_2/w_2, \ldots, v_m/w_m] \Rightarrow v_*}{\Gamma \triangleright \delta\ e_1 \cdots e_m \dashrightarrow v_*}$$

$$\frac{\Gamma(x_i) = \sigma\ t_1 \cdots t_k \qquad \langle f, \sigma \rangle\text{-rule is } f\ (\sigma\ x_1 \cdots x_k)\ y_1 \cdots y_m = e_0 \qquad \{x_1 \mapsto t_1, \ldots, x_k \mapsto t_k\} \triangleright e_0 \dashrightarrow v_0 \qquad \Gamma \triangleright e_i \dashrightarrow v_i\ (\text{for } 1 \le i \le m) \qquad v_0[v_1/y_1, v_2/y_2, \ldots, v_m/y_m] \Rightarrow v_*}{\Gamma \triangleright f\ x_i\ e_1 \cdots e_m \dashrightarrow v_*}$$

Fig. 2 Summarization phase.

The summarization phase is characterized by the following theorem. In the following, let $\text{VARS}(p) = \{y_1, \ldots, y_m\}$, where $p = s_0 \oplus (s_1 \otimes y_1) \oplus \cdots \oplus (s_m \otimes y_m)$, and $\text{INPUTS}(\Gamma) = \{t_1, \ldots, t_k\}$, where $\Gamma = \{x_1 \mapsto t_1, \ldots, x_k \mapsto t_k, y_1 \mapsto v_1, \ldots, y_m \mapsto v_m, z_1 \mapsto v'_1, \ldots, z_n \mapsto v'_n\}$.

Theorem 6 For any expression e and environment Γ that does not contain any y_i ($i \in \mathbb{N}$), the following holds. Given $\Gamma_Y = \{y_1 \mapsto s_1, \ldots, y_m \mapsto s_m\}$ and a left-linear polynomial w, define $\Gamma_Y(w)$ as follows:

$$\Gamma_Y(w) = s \iff w[\Gamma_Y(y_{i_1})/y_{i_1}, \ldots, \Gamma_Y(y_{i_m})/y_{i_m}] \Rightarrow s \quad \text{where } \{y_{i_1}, \ldots, y_{i_m}\} = \text{VARS}(w).$$

Then, $\Gamma \triangleright e \dashrightarrow v$ implies $\Gamma' \cup \Gamma_Y \triangleright e \to \Gamma_Y(v)$, where Γ' is an environment obtained by substituting every $z_i \mapsto v_i$ in Γ with $z_i \mapsto \Gamma_Y(v_i)$.

Proof Sketch The proof uses induction on the structures of e and $\text{INPUTS}(\Gamma)$. The most nontrivial case is $e = f\ x_i\ e_1 \cdots e_m$. Following the rules in Fig. 2, let e_0 be the right-hand-side expression of the function definition used for the recursive call of $f\ x_i$. Because the induction hypothesis can be applied to every e_i ($0 \le i \le m$), it is sufficient to show that $v_0[v_1/y_1, \ldots, v_m/y_m] \Rightarrow v_*$ implies $\Gamma_Y(v_*) = \Gamma_Y(v_0[v_1/y_1, \ldots, v_m/y_m]) = v_0[\Gamma_Y(v_1)/y_1, \ldots, \Gamma_Y(v_m)/y_m]$, where each v_i is the summary obtained from e_i. This holds because $\text{VARS}(v_0) = \{y_1, \ldots, y_m\}$, which can easily be shown by a similar induction, and because the binary relation $\Rightarrow$ preserves the semantics of left-linear polynomials, owing to the properties of semirings. The other cases are similar or trivial. □

As a result of the summarization phase, the summary obtained from sum_{acc} discussed in Section 1 is a left-linear polynomial of the form $n + 1 \times y$, where n is the summation of the tree and y is a variable of the polynomial. The summary obtained from *dfa* is a left-linear polynomial of the form $v \vee (v \wedge y_1)$, where v is the bit vector corresponding to variables introduced or updated in the tree.

4.3 Dependency Resolution Phase

The dependency resolution phase uses the result of the summarization phase as follows.

Definition 7 Let $\mathcal{M}$ be an MTT whose initial expression is e_0, $\mathcal{R}$ be a semiring, and $[\![\cdot]\!]$ be a left-linear interpretation of $\mathcal{M}$

$$\Gamma \triangleright_\eta y_i \hookrightarrow \Gamma(y_i), \emptyset$$

$$\Gamma \triangleright_\eta z_i \hookrightarrow \Gamma(z_i), \emptyset$$

$$\frac{\Gamma \triangleright_\eta e_1 \hookrightarrow v_1, \Delta_1 \qquad \Gamma \cup \{z_i \mapsto v_1\} \triangleright_\eta e_1 \hookrightarrow v_2, \Delta_2}{\Gamma \triangleright_\eta \textbf{let } z_i = e_1 \textbf{ in } e_2 \hookrightarrow v_2, \Delta_1 \cup \Delta_2 \cup \{(\eta, z_i) \mapsto v_1\}}$$

$$\frac{\Gamma \triangleright_\eta e_i \hookrightarrow v_i, \Delta_i \text{ (for } 1 \le i \le m) \qquad [\![\delta]\!]\ v_1 \cdots v_m = v_*}{\Gamma \triangleright_\eta \delta\ e_1 \cdots e_m \hookrightarrow v_*, \bigcup_{1 \le i \le m} \Delta_i}$$

$$\frac{\begin{array}{c}\Gamma(x_i) = \sigma\ t_1 \cdots t_k \qquad \langle f, \sigma\rangle\text{-rule is } f\ (\sigma\ x_1 \cdots x_k)\ y_1 \cdots y_m = e_0 \\ \Gamma \triangleright_\eta e_i \hookrightarrow v_i, \Delta_i \text{ (for } 1 \le i \le m) \\ \Gamma' = \{x_1 \mapsto t_1, \ldots, x_k \mapsto t_k\} \\ \Gamma' \triangleright e_0 \dashrightarrow v_0 \qquad v_0[v_1/y_1, v_2/y_2, \ldots, v_m/y_m] = v_* \\ \Gamma' \cup \{y_1 \mapsto v_1, \ldots, y_m \mapsto v_m\} \triangleright_{ID(\Gamma(x_i))} e_0 \hookrightarrow \_, \Delta_0\end{array}}{\Gamma \triangleright_\eta f\ x_i\ e_1 \cdots e_m \hookrightarrow v_*, \bigcup_{0 \le i \le m} \Delta_i}$$

Fig. 3 Dependency resolution phase.

using $\mathcal{R}$. The dependency resolution of triple $(\mathcal{M}, \mathcal{R}, [\![\cdot]\!])$ for input tree t_0 consists of a value $v_* \in \mathcal{R}$ and mapping $\Delta \subseteq ((I \times Z) \to \mathcal{R})$ that satisfy

$$\{x_1 \mapsto t_0\} \triangleright_\top e_0 \hookrightarrow v_*, \Delta$$

according to the rules shown in **Fig. 3**, in which _ denotes a value unnecessary to compute.

The dependency resolution phase is the same as the usual evaluation except for the case of a recursive call. In this phase, a recursive call calculates not the value but the mapping. The value is instead calculated using the summary, i.e., a left-linear polynomial, obtained by the summarization phase. Therefore, even if a recursive call depends on a value calculated by another recursive call, the two recursive calls can be evaluated in parallel. For example, consider evaluating $f_1\ x_1\ (f_2\ x_2)$. In the usual evaluation, the evaluation of f_1 starts after that of $f_2\ x_2$. In the dependency resolution phase, however, the evaluations of $f_1\ x_1$ and $f_2\ x_2$ can be performed simultaneously because the accumulation parameter of $f_1\ x_1$ can be obtained from the summary for $f_2\ x_2$.

The correctness of the dependency resolution phase follows from Theorem 6.

Theorem 8 For any environment Γ, expression e, and subtree ID η, $\Gamma \triangleright_\eta e \hookrightarrow v, \Delta$ implies $\Gamma \triangleright_\eta e \to v, \Delta$.

Proof Sketch As similar to Theorem 6, the proof uses induction on the structures of e and INPUTS(Γ). The case of a recursive function call is the most nontrivial, but it is immediately justified from Theorem 6. The other cases are trivial. □

4.4 Computational Complexity

The computational complexity of the parallel evaluation algorithm is now examined. Let N, H, and P be the size of the input, the height of the input, and the number of processors, respectively. The size of the MTT is regarded as a constant. Note that not only the number of the recursive functions but also the number of variables is constant, because only those defined in the program will appear during the evaluation.

As explained here, the time complexity of the parallel evaluation algorithm is O(N/P + H). In the summarization phase, each recursive function visits each subtree at most once. Multiple visits that may result from a naive use of the rules shown in Fig. 2 can be avoided by memoization because every visit yields the same result. In addition to the recursive call, for each node, each function performs computations at most proportional to the right-hand-side expression of the function. Note that the computations consist of simplifications and substitutions to left-linear polynomials having a constant number of variables. In summary, then, the work of the summarization phase is at most proportional to the size of the input tree. Because function calls to independent subtrees can be computed in parallel, the cost of the summarization phase is O(N/P + H). Next, for the dependency resolution phase, assume that the summarization phase is complete and its results for each subtree have been memoized. Then, the amount of computation necessary for each node and each function is also at most proportional to the size of the right-hand-side expression of the function. Moreover, after calculations of accumulation parameters, which are done in constant time, independent subtrees can be processed in parallel. Therefore, the cost of the dependency resolution phase is also O(N/P + H).

The discussion so far can be formalized via the following lemmas and theorem. Let SIZE(t) and HEIGHT(t) be the size and height of tree t, respectively. Furthermore, let these definitions extend to environments, i.e., $\text{SIZE}(\Gamma) = \sum_{t \in \text{INPUTS}(\Gamma)} \text{SIZE}(t)$ and $\text{HEIGHT}(\Gamma) = \max_{t \in \text{INPUTS}(\Gamma)} \text{HEIGHT}(t)$.

Lemma 9 For any environment Γ and expression e, a value v satisfying

$$\Gamma \triangleright e \dashrightarrow v$$

can be calculated in O(SIZE(Γ)) work and O(HEIGHT(Γ)) parallel computational steps.

Proof Sketch The proof uses induction on the structure of INPUTS(Γ). Because recursive calls to independent subtrees can be done in parallel, it is sufficient to show that $\Gamma \triangleright e \dashrightarrow v$ can be calculated in O(1) works and O(1) parallel computational steps if every result (i.e., summary) of a recursive call to any of INPUTS(Γ) is known. This can be straightforwardly proved by induction on the structure of expression e. Note that the costs of simplifying and substituting left-linear polynomials are O(1) because each left-linear polynomial contains O(1) variables. □

Lemma 10 For any environment Γ and expression e, a value v and mapping Δ satisfying

$$\Gamma \triangleright e \hookrightarrow v, \Delta$$

can be calculated in O(SIZE(Γ)) work and O(HEIGHT(Γ)) parallel computational steps if every summary of a function for a subtree of INPUTS(Γ) is known.

Proof Sketch The proof begins by showing that the evaluation costs O(1) work and O(1) parallel computation steps if the costs of recursive calls are ignored. Similarly to Lemma 9, this can easily be shown by induction on the structure of expression e. Then, the cost of calculating the mappings by recursive calls is estimated. Recursive calls to independent subtrees can be performed in parallel; moreover, because of the assumption discussed in Section 3.3, it is sufficient for each function to visit each subtree at most once. Finally, induction on the structure of INPUTS(Γ) proves this lemma. □

Theorem 11 Let $\mathcal{M}$ be an MTT whose initial expression is

e_0, $\mathcal{R}$ be a semiring, and $[\![\cdot]\!]$ be a left-linear interpretation of $\mathcal{M}$ using $\mathcal{R}$. The semantics of triple $(\mathcal{M}, \mathcal{R}, [\![\cdot]\!])$ for input tree t_0, namely,

$$\{x_1 \mapsto t_0\} \triangleright_\top e_0 \to v_*, \Delta,$$

can be calculated on an exclusive-read exclusive-write parallel random-access machine consisting of P processors in time $O(N/P + H)$, where N and H are respectively the size and height of t_0.

Proof Sketch From Theorem 8, $\{x_1 \mapsto t_0\} \triangleright_\top e_0 \hookrightarrow v_*, \Delta$. The amount of work and the number of parallel computational steps are given by Lemmas 9 and 10. Then, the computational complexity follows from Brent's scheduling principle [4]. □

Note that, in general, the evaluation of an MTT cannot be finished in a time proportional to the size of input tree. For instance, the computational cost of MTT *exp*, discussed in Section 2.2, is exponential, because the size of the output is exponential to that of the input. The proposed algorithm avoids this inefficiency for the following two reasons. First, a program written in the proposed language calculates not the output tree but its interpretation; thus, it is unnecessary to calculate huge outputs. Second, when an MTT outputs a tree significantly larger than the input, it does the same computations more than once by visiting the same subtree several times or duplicating computed trees. In contrast, the proposed algorithm avoids this inefficiency: the summarization phase uses memoization, and the dependency resolution phase never visits the same subtree by virtue of the assumption discussed in Section 3.3. In other words, the assumption enables efficient parallel tree processing.

5. Improving Computational Complexity through Fusion Transformation

The computational complexity, $O(N/P + H)$, given by Theorem 11 is ideal if the input tree is balanced, i.e., if $H \in O(\log N)$. If the input is a list-like structure whose height is proportional to its size, however, then the complexity is $O(N)$ regardless of the number of processors, showing little parallel speedup. Lists are the most widely used data structures in functional programming, and practical tree structures, such as XML data and syntax trees, are very often list-like. Thus, this is a serious shortcoming.

A divide-and-conquer approach is typical for parallel processing of list-like structures. The input is split at the middle, and then, each part is processed in parallel. A recursive divide-and-conquer approach for a list can be understood as transforming the list to a complete binary tree. Now recall the MTT *flat* discussed in Section 2.2 and let $F\ x = \mathit{flat}\ x\ [\,]$. Then, a recursive divide-and-conquer approach can be regarded as the following strategy:

> Instead of calculating function f for list x, prepare a complete binary tree t such that $F\ t = x$ and do parallel evaluation of $f \circ F$.

This strategy has the following two requirements.

- The complete binary tree t such that $F\ t = x$ must be efficiently calculated.
- Efficient parallel evaluation of $f \circ F$ must be possible.

Several approaches can be used to fulfill the first requirement. For example, if the list is implemented as an array, as usual in the context of parallel computing, then the transformation to a complete binary tree is unnecessary.

The major requirement is the second one. The discussion to this point implies the following.

> A list processing function f has an efficient parallel implementation if $f \circ F$ can be specified by a triple $(\mathcal{M}, \mathcal{R}, [\![\cdot]\!])$.

Fusion transformations for MTTs are useful for understanding when $f \circ F$ can be specified by a triple $(\mathcal{M}, \mathcal{R}, [\![\cdot]\!])$. In particular, the following fact about ATTs and single-use restricted MTTs is significant [*4].

Theorem 12 [Refs. [11], [13]] Given an ATT $\mathcal{A}$ and a single-use restricted MTT $\mathcal{M}$, there is ATT $\mathcal{A}'$ that is equivalent to the composition $\mathcal{A} \circ \mathcal{M}$. Moreover, $\mathcal{A}'$ can be constructed from $\mathcal{A}$ and $\mathcal{M}$.

This theorem describes a fusion transformation, because it obtains a single MTT from a composition of two MTTs.

Because *flat* is single-use restricted, Theorem 12 immediately leads to the following theorem.

Theorem 13 For a list processing function expressed by a triple $(\mathcal{M}, \mathcal{R}, [\![\cdot]\!])$, if $\mathcal{M}$ is an ATT, it can be evaluated in $O(N/P + \log N)$ time on an exclusive-read exclusive-write parallel random-access machine, where N is the length of the input list and P is the number of processors.

Proof Sketch This theorem is a special case of Theorem 14, which will be proved later. □

For example, consider the following list processing function sum_{list}.

$sum_{list}\ (a : x) = a + sum_{list}\ x$
$sum_{list}\ [\,] = 0$

This function can be expressed by an ATT. The result of fusing it with *flat* is exactly sum_{acc}.

As a more complex example, consider a tail-recursive summation function, sum_{tr}.

$sum_{tr}\ (a : x)\ y = sum_{tr}\ x\ (y + a)$
$sum_{tr}\ [\,]\ y = y$

This function can also be expressed by an ATT. Fusion with *flat* results in the following. It is fairly complex but indeed an MTT, and therefore, efficient parallel evaluation is possible.

$sum^2_{acc}\ x_1\ (sum^1\ x_1\ 0)$
where
$sum^1_{acc}\ (\mathsf{Tip}\ n)\ y = y + n$
$sum^1_{acc}\ (\mathsf{Fork}\ l\ r)\ y = sum^1_{acc}\ r\ (sum^1_{acc}\ l\ y)$
$sum^2_{acc}\ (\mathsf{Tip}\ n)\ y = y$
$sum^2_{acc}\ (\mathsf{Fork}\ l\ r)\ y = sum^2_{acc}\ l\ (sum^2_{acc}\ r\ y)$

Shunt trees [25] enable generalizing this approach for list processing to tree processing. A shunt tree remembers the process of applying parallel tree contraction algorithms [29] to an input tree. Given a tree of size N, the size and height of the corresponding shunt tree are $O(N)$ and $O(\log N)$, respectively. In addition, the original tree can be recovered from the shunt tree by a single-use restricted MTT. This leads to the following theorem, which is a generalization of Theorem 13 because shunt trees corresponding to lists are in fact complete binary trees.

[*4] The focus here is not on MTTs but on triples. This difference does not affect Theorems 13 and 14 at all.

Theorem 14 For tree processing specified by a triple $(\mathcal{M}, \mathcal{R}, [\![\cdot]\!])$, if $\mathcal{M}$ is an ATT, it can be evaluated in $O(N/P + \log N)$ time on an exclusive-read exclusive-write parallel random-access machine, where N is the size of the input tree and P is the number of processors.

Proof Sketch First, construct the shunt tree corresponding to the input tree. A parallel tree contraction algorithm achieves this in time $O(N/P + \log N)$ [25]. Second, by Theorem 12, obtain an ATT equivalent to the composition of the triple and the single-use restricted MTT that restores the original input from the shunt tree. Finally, evaluate the obtained ATT according to Theorem 11. The computational complexity follows from the fact that the size and height of the shunt tree are $O(N)$ and $O(\log N)$, respectively. □

Limitation

Theorem 14 enables efficient parallel evaluation for tree processing specified by an ATT. It is natural to also consider tree processing specified by an MTT; however, efficient parallel evaluation seems difficult for that case.

In general, a composition of an MTT and a single-use restricted MTT cannot be expressed by an MTT. For instance, consider a composition of the MTT *exp* discussed in Section 2.2 and the following single-use restricted MTT *count*.

count x_1 Z

where *count* (Tip a) y_1 = S y_1

count (Fork x_1 x_2) y_1 = *count* x_1 (*count* x_2 y_1)

If the input is a complete binary tree that has height n and therefore 2^{n-1} leaves, *count* results in a sequence of length 2^{n-1}; hence, applying *exp* to the sequence results in a sequence of length $2^{2^{n-1}}$. Because an MTT cannot produce a tree whose height is doubly-exponential to the height of the input [9], the composition cannot be expressed by an MTT. In short, it is not straightforward to generalize Theorem 14 to all tree transformations specified by MTTs.

Another approach would be to generalize the parallel algorithm to a class of tree transformations that is more general than MTTs. One such class is high-level tree transducers [6]. It is difficult, however, to provide efficiency-guaranteed parallel evaluation of high-level tree transducers because they are too expressive: they can express computations similar to those of the simply-typed lambda calculus.

On the other hand, even for a tree processing that cannot be specified by an ATT, the proposed approach may be applicable. For example, consider macro forest transducers [26], which generalize MTTs because they can yield a sequence of trees, i.e., a forest, by using a forest concatenation operator. In most semiring-based interpretations of macro forest transducers, it is not harmful to regard the forest concatenation operator as a constructor: the forest concatenation corresponds merely to an addition or a binary maximum if the objective is a kind of summation (e.g., counting) or maximization (e.g., height calculation), respectively. In such a case, the proposed method is straightforwardly applicable to the tree processing specified by a macro forest transducer, obtaining good parallel speedup.

6. Related Work

This paper has proposed a parallel evaluation method for tree processing. The method was built on a combination of two preceding proposals [24], [25] by the author, together with a colleague in one case.

First, Morihata and Matsuzaki [25] proposed shunt trees to encapsulate parallel tree contraction algorithms in data structures. Shunt trees represent the tree processing patterns of parallel tree contraction algorithms. For any tree, the height of its shunt tree representation is logarithmic in its size. Therefore, tree processing can show ideal parallel speedup if it can be specified as a series consisting of top-down or bottom-up processing over the shunt tree representation corresponding to the input tree. In addition, that study discussed the usefulness of fusion transformations for systematically deriving parallel processing over shunt trees. The limitation was that the derivations of parallel processing were rather complicated and done by hand. In contrast, the current paper has observed that the derivation can be automatic for tree processing specified by ATTs, which covers all examples discussed by that study.

Second, Morihata [24] showed that a single-use restricted MTT can be efficiently evaluated using a parallel tree contraction algorithm, but that result has the following two major limitations. First, the single-use restriction is a strong requirement. For a pure tree transformation, theoretically the restriction is not severe. Unless the transformation can cause an unlimited amount of copying, we can avoid copying of calculated trees by constructing the same tree more than once. For tree processing concerning usual value computations, however, it is not practical to do the same computation more than once. Therefore, the single-use restriction makes it difficult to deal with tree processing beyond pure tree transformations. In contrast, the current paper has eliminated that restriction and showed a parallel evaluation algorithm for any MTT or ATT. Second, the method in Ref. [24] relies on a parallel tree contraction algorithm, which is fairly complex. The complexity makes it difficult for those unfamiliar with parallel tree contraction algorithms to understand the method; moreover, because of the complexity, it was unclear whether the method could be generalized to MTTs or ATTs that are not single-use restricted. The current paper has simplified the algorithm as well as the correctness proof by using shunt trees and fusion transformations, thereby showing the possibility of generalization. The simplification also enables dealing with summarization for substructures. Note that the method of Morihata [24] is immediately obtained by the use of shunt trees and Theorem 12, which yields a single-use restricted MTT from a composition of two single-use restricted MTTs.

Apart from the theoretical simplicity, the approach of using shunt trees and fusion transformations has a practical benefit. Parallel tree contraction algorithms intricately process input trees according to careful scheduling. In contrast, the proposed algorithm simply processes an input tree in nearly top-down and bottom-up manners. Therefore, existing optimization for parallel tree processing, such as flattening trees to nested arrays [2], [15], [28], can be naturally applied.

The proposed approach allows complex tree accumulations and yet guarantees ideal parallel speedups. To the best of the author's knowledge, no existing approach provides both of these charac-

teristics, except for the one described above [24] for single-use restricted MTTs. Other works [1], [12], [21], [22], [30], [32] studied when certain tree processing patterns, especially top-down and bottom-up processing, can be efficiently evaluated on parallel computers. None of them considered complex tree traversals as attribute grammars. On the other hand, there have been many studies on parallel evaluation of attribute grammars [3], [14], [16], [17], [19], [23], [30], [31], but most of them do not guarantee parallel speedups. A notable exception is Reps' scan grammars [30], which provided an efficiency-guaranteed parallel implementation for an attribution that scans leaves by using an associative operator. The current approach can be regarded as a generalization of scan grammars: it can perform arbitrary traversals by using semiring operators, and moreover, it can deal with MTTs rather than attribute grammars.

7. Conclusion and Future Work

This paper has examined tree processing defined by MTTs and semirings. A parallel evaluation algorithm was proposed for this parallel tree processing approach, and the condition for the algorithm to guarantee ideal asymptotic parallel speedup was discussed. It was shown that the number of parallel evaluation steps is proportional to the height of the input tree if the tree processing is specified by an MTT, and logarithmic in the size if specified by an ATT. These results were easily obtained through careful design of the language for describing parallel tree processing and the use of fusion transformations to improve parallel speedup.

Several issues remain.

First, this paper considered not tree transformations but tree processing using semirings. Parallel evaluation of tree transformations has important applications including queries to tree-structured databases. We might expect that the result of this paper would also be applicable to tree transformations. A tree is identified by a set of paths from the root to the leaves, and the paths can be calculated by using a semiring for languages. In practice, however, substitutions and simplifications of left-linear polynomials may require duplication of strings, thereby making the computational complexity worse. Morihata [24] considered single-use restricted MTTs so as to avoid this difficulty concerning tree duplication. It is unclear whether the use of directed acyclic graphs or similar structures could resolve the issue. Further study is left for future work.

Section 5 discussed the difficulty of extending the result of this paper to deal with more general classes of tree transformations such as high-level tree transducers. There may exist cases of tree processing that are not definable by an MTT but have the possibility of efficient parallel evaluation. It might be interesting to look for such cases of tree processing.

Fusion transformations were used to improve parallel computational complexity. This approach of considering an intermediate structure suitable for the objective and then applying a fusion transformation to derive a function on the intermediate structure, is not specific to parallelization. Potential applications include computations for compressed data without decompression, computations for indexed data, and incremental computation of results according to the modification of inputs.

Acknowledgments I am grateful to the reviewers for their valuable comments helping to improve this paper. The work is supported by JSPS Grant-in-Aid for Young Researchers (B), 24700019.

References

[1] Abrahamson, K.R., Dadoun, N., Kirkpatrick, D.G. and Przytycka, T.M.: A Simple Parallel Tree Contraction Algorithm, *J. Algorithms*, Vol.10, No.2, pp.287–302 (1989).
[2] Blelloch, G.E., Hardwick, J.C., Sipelstein, J., Zagha, M. and Chatterjee, S.: Implementation of a Portable Nested Data-Parallel Language, *J. Parallel Distrib. Comput.*, Vol.21, No.1, pp.4–14 (1994).
[3] Boehm, H.-J. and Zwaenepoel, W.: Parallel Attribute Grammar Evaluation, *Proc. 7th International Conference on Distributed Computing Systems, Berlin, Germany, September 1987*, IEEE Computer Society Press, pp.347–355 (1987).
[4] Brent, R.P.: The Parallel Evaluation of General Arithmetic Expressions, *J. ACM*, Vol.21, No.2, pp.201–206 (1974).
[5] Engelfriet, J. and Vogler, H.: Macro Tree Transducers, *J. Comput. Syst. Sci.*, Vol.31, No.1, pp.71–146 (1985).
[6] Engelfriet, J. and Vogler, H.: High Level Tree Transducers and Iterated Pushdown Tree Transducers, *Acta Inf.*, Vol.26, No.1/2, pp.131–192 (1988).
[7] Engelfriet, J. and Vogler, H.: Modular Tree Transducers, *Theor. Comput. Sci.*, Vol.78, No.2, pp.267–303 (1991).
[8] Fülöp, Z., Herrmann, F., Vágvölgyi, S. and Vogler, H.: Tree Transducers with External Functions, *Theor. Comput. Sci.*, Vol.108, No.2, pp.185–236 (1993).
[9] Fülöp, Z. and Vogler, H.: *Syntax-Directed Semantics: Formal Models Based on Tree Transducers*, Springer-Verlag New York, Inc., Secaucus, NJ, USA (1998).
[10] Fülöp, Z. and Vogler, H.: A Characterization of Attributed Tree Transformations by a Subclass of Macro Tree Transducers, *Theory of Computing Systems*, Vol.32, No.6, pp.649–676 (1999).
[11] Ganzinger, H. and Giegerich, R.: Attribute coupled grammars, *Proc. 1984 SIGPLAN Symposium on Compiler Construction*, pp.157–170, ACM (1984).
[12] Gibbons, J., Cai, W. and Skillicorn, D.B.: Efficient Parallel Algorithms for Tree Accumulations, *Sci. Comput. Program.*, Vol.23, No.1, pp.1–18 (1994).
[13] Giegerich, R.: Composition and evaluation of attribute coupled grammars, *Acta Inf.*, Vol.25, No.4, pp.335–423 (1988).
[14] Jourdan, M.: A Survey of Parallel Attribute Evaluation Methods, *Attribute Grammars, Applications and Systems, International Summer School SAGA, Prague, Czechoslovakia, June 4–13, 1991, Proc.*, Lecture Notes in Computer Science, Vol.545, pp.234–255, Springer (1991).
[15] Keller, G. and Chakravarty, M.M.T.: Flattening Trees, *Euro-Par '98 Parallel Processing, 4th International Euro-Par Conference, Southampton, UK, September 1–4, 1998, Proc.*, Lecture Notes in Computer Science, Vol.1470, pp.709–719, Springer (1998).
[16] Klaiber, A.C. and Gokhale, M.: Parallel Evaluation of Attribute Grammars, *IEEE Trans. Parallel Distrib. Syst.*, Vol.3, No.2, pp.206–220 (1992).
[17] Klein, E.: Parallel ordered attribute grammars, *Proc. 1992 International Conference on Computer Languages, Oakland, California, USA, 20–23 Apr. 1992*, pp.106–116, IEEE (1992).
[18] Knuth, D.E.: Semantics of Context-Free Languages., *Mathematical Systems Theory*, Vol.2, No.2, pp.127–145 (1968).
[19] Kuiper, M.F. and Swierstra, S.D.: Parallel Attribute Evaluation: Structure of Evaluators and Detection of Parallelism, *Attribute Grammars and their Applications, International Conference WAGA, Paris, Fance, September 19–21, 1990, Proc.*, Lecture Notes in Computer Science, Vol.461, pp.61–75, Springer (1990).
[20] Maneth, S.: The Macro Tree Transducer Hierarchy Collapses for Functions of Linear Size Increase, *FST&TCS 2003: Foundations of Software Technology and Theoretical Computer Science, 23rd Conference, Mumbai, India, December 15–17, 2003, Proc.*, Lecture Notes in Computer Science, Vol.2914, pp.326–337, Springer (2003).
[21] Matsuzaki, K., Hu, Z. and Takeichi, M.: Parallelization with Tree Skeletons, *Euro-Par 2003: Parallel Processing, 9th International Euro-Par Conference, Klagenfurt, Austria, August 26–29, 2003, Proc.*, Lecture Notes in Computer Science, Vol.2790, pp.789–798, Springer (2003).
[22] Matsuzaki, K., Hu, Z. and Takeichi, M.: Parallel skeletons for manipulating general trees, *Parallel Comput.*, Vol.32, No.7-8, pp.590–603 (2006).
[23] Meyerovich, L.A., Torok, M.E., Atkinson, E. and Bodík, R.: Parallel

schedule synthesis for attribute grammars, *Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '13, Shenzhen, China, February 23–27*, pp.187–196, ACM (2013).
[24] Morihata, A.: Macro Tree Transformations of Linear Size Increase Achieve Cost-Optimal Parallelism, *Programming Languages and Systems - 9th Asian Symposium, APLAS 2011, Kenting, Taiwan, December 5–7, 2011, Proc.*, Lecture Notes in Computer Science, Vol.7078, pp.204–219, Springer (2011).
[25] Morihata, A. and Matsuzaki, K.: Balanced trees inhabiting functional parallel programming, *Proc. 16th ACM SIGPLAN International Conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*, pp.117–128, ACM (2011).
[26] Perst, T. and Seidl, H.: Macro forest transducers, *Inf. Proc. Lett.*, Vol.89, No.3, pp.141–149 (2004).
[27] Peyton Jones, S. (Ed.): *Haskell 98 Language and Libraries: The Revised Report*, Cambridge University Press, Cambridge, UK (2003).
[28] Peyton Jones, S.L., Leshchinskiy, R., Keller, G. and Chakravarty, M.M.T.: Harnessing the Multicores: Nested Data Parallelism in Haskell, *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2008, December 9-11, 2008, Bangalore, India, Dagstuhl Seminar Proceedings*, Vol.08004, Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany (2008).
[29] Reif, J.H. (Ed.): *Synthesis of Parallel Algorithms*, Morgan Kaufmann Publishers (1993).
[30] Reps, T.W.: Scan Grammars: Parallel Attribute Evaluation via Data-Parallelism, *Proc. 5th Annual ACM Symposium on Parallel Algorithms and Architectures, June 30 - July 2, 1993, Velen, Germany*, pp.367–376 (1993).
[31] Saraiva, J. and Henriques, P.: Concurrent attribute evaluation, *Computing Systems in Engineering*, Vol.6, No.4–5, pp.451–457 (1995).
[32] Skillicorn, D.B.: Structured Parallel Computation in Structured Documents, *J. Univ. Comput. Sci.*, Vol.3, No.1, pp.42–68 (1997).

Akimasa Morihata was born in 1981. He received a Ph.D. from Graduate School of Information Science and Technology, the University of Tokyo, in 2009. He entered a JSPS research fellowship for young scientists in 2009, then became a research associate at Research Institute of Electrical Communication, Tohoku University, in 2010. Later, he became a lecturer at Graduate School of Arts and Sciences, the University of Tokyo, in 2014, and then an associate professor in 2017. His research interests include program transformation, functional programming, parallel programming, and systematic development of efficient algorithms. He is a member of the Japan Society for Software Science and Technology and Information Processing Society of Japan.