Lambda Calculus with Algebraic Simplification for Reduction Parallelization by Equational Reasoning

AKIMASA MORIHATA, the University of Tokyo, Japan

Parallel reduction is a major component of parallel programming and widely used for summarization and aggregation. It is not well understood, however, what sorts of nontrivial summarizations can be implemented as parallel reductions. This paper develops a calculus named λ^{AS} , a simply typed lambda calculus with algebraic simplification. This calculus provides a foundation for studying parallelization of complex reductions by equational reasoning. Its key feature is δ abstraction. A δ abstraction is observationally equivalent to the standard λ abstraction, but its body is simplified before the arrival of its arguments by using algebraic properties such as associativity and commutativity. In addition, the type system of λ^{AS} guarantees that simplifications due to δ abstractions do not lead to serious overheads. The usefulness of λ^{AS} is demonstrated on examples of developing complex parallel reductions, including those containing more than one reduction operator, loops with jumps, prefix-sum patterns, and even tree manipulations.

CCS Concepts: • **Theory of computation** → **Parallel computing models**; • **Mathematics of computing** → **Lambda calculus**.

Additional Key Words and Phrases: Parallel Reduction, Lambda Calculus, Algebraic Simplification, Equational Reasoning

ACM Reference Format:

Akimasa Morihata. 2019. Lambda Calculus with Algebraic Simplification for Reduction Parallelization by Equational Reasoning. *Proc. ACM Program. Lang.* 3, ICFP, Article 80 (August 2019), 25 pages. <https://doi.org/10.1145/3341644>

1 INTRODUCTION

Functional programming is commonly regarded as a promising approach in parallel programming. A major reason is the freedom of side effects, enabling evaluation of independent subexpressions in parallel. For example, in the following recursive Fibonacci function,

$$\text{fib } n = \text{if } n \leq 1 \text{ then } 1 \text{ else } \text{fib } (n - 1) + \text{fib } (n - 2),$$

it is syntactically clear that the two recursive calls, $\text{fib } (n - 1)$ and $\text{fib } (n - 2)$, can be simultaneously evaluated. For this reason, functional programming makes parallel programming easy and intuitive.

Another benefit of using functional programs in parallel programming is equational reasoning, which helps certify the correctness of parallel implementations. As an example, consider the following parallel implementation of the fib function in Haskell.

$$\begin{aligned} \text{fib } n = & \text{if } n \leq 1 \text{ then } 1 \text{ else } \text{par } x \text{ (pseq } y \text{ (} x + y \text{))} \\ & \text{where } x = \text{fib } (n - 1) \\ & \quad y = \text{fib } (n - 2) \end{aligned}$$

Author's address: Akimasa Morihata, Graduate School of Arts and Sciences, the University of Tokyo, 3-8-1, Komaba, Meguro-ku, Tokyo, 153-8902, Japan, morihata@graco.c.u-tokyo.ac.jp.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

© 2019 Copyright held by the owner/author(s).

2475-1421/2019/8-ART80

<https://doi.org/10.1145/3341644>

$poly\ x\ [] = 0$	$sumB1\ e\ [] = e$
$poly\ x\ (a : y) = a + x \times poly\ x\ y$	$sumB1\ e\ (a : x) = \text{if } a < 0 \text{ then } e$
(a) Polynomial computation	$\text{else } sumB1\ (e + a)\ x$
	(b) Summation with break
$psum\ []\ e = []$	$rd\ (\text{Assign } v\ e)\ y = (\text{remove } v\ y) \cup \{(v, e)\}$
$psum\ (a : x)\ e = e : (psum\ x\ (e + a))$	$rd\ (\text{Seq } s_1\ s_2)\ y = rd\ s_2\ (rd\ s_1\ y)$
(c) Prefix sum	$rd\ (\text{If } e\ s_1\ s_2)\ y = rd\ s_1\ y \cup rd\ s_2\ y$
	$rd\ (\text{While } e\ s)\ y = y \cup rd\ s\ y$
	(d) Reaching definition analysis

Fig. 1. Examples of nontrivial reductions

In this program, *par* requests the evaluation of its first argument in parallel to that of its second argument, and *pseq* forces the evaluation of its first argument before the evaluation of its second argument. The correctness of this implementation *immediately* follows from the observational equalities of *par* and *pseq*:

$$\begin{aligned} par\ a\ b &= b \\ pseq\ a\ b &= b \quad (\text{unless } a \text{ is undefined}). \end{aligned}$$

Such equational reasoning is useful for not only certification but also *development* of parallel implementations. For example, consider the following usual summation function, *sum*.

$$\begin{aligned} sum\ [] &= 0 \\ sum\ (a : x) &= a + sum\ x \end{aligned}$$

Although this function does not appear to contain independent subexpressions, equational reasoning reveals its potential for parallel evaluation.

$$\begin{aligned} &sum\ (a : b : x) \\ &= \{ \text{unfolding the definition of } sum \} \\ &\quad a + (b + sum\ x) \\ &= \{ \text{associativity of } + \} \\ &\quad (a + b) + sum\ x \\ &= \{ \text{folding the definition of } sum \} \\ &\quad sum\ [a, b] + sum\ x \end{aligned}$$

It is not difficult to generalize the observation above to $sum\ (l \# r) = sum\ l + sum\ r$, where $\#$ denotes a list concatenation operator. That is, *sum* can process the elements of the first half, *l*, and the remaining elements, *r*, in parallel.

Such parallel summation, *sum*, is an instance of *parallel reduction*, also known as parallel summarization or aggregation. Parallel reductions are used for calculating the total, maximum, average, and other results for huge data. Parallel reductions appear everywhere in real programs and are thus supported by most modern parallel programming environments, including MPI¹, OpenMP², Intel Threading Building Blocks³, MapReduce [Dean and Ghemawat 2004], Manticore [Fluet et al. 2008], Repa (REgular PArallel arrays) for Haskell [Keller et al. 2010], and Futhark [Henriksen et al. 2017].

¹<http://mpi-forum.org/>

²<http://openmp.org/wp/>

³<https://www.threadingbuildingblocks.org/>

Despite the importance and usefulness of parallel reductions, the current support for them is not satisfactory. Existing parallel programming environments support only specific patterns of parallel reductions and require programmers to write programs using those patterns. To see the problem, consider the following *poly* function, which calculates the value of a polynomial represented by a list of coefficients, as shown in Figure 1 (a):

$$\text{poly } x [a_0, a_1, \dots, a_n] = a_0 + a_1x + \dots + a_nx^n.$$

Although *poly* is a modest generalization of *sum* (note that $\text{poly } 1 = \text{sum}$), it does not fit the parallel reduction pattern supported by existing environments, because it involves more than one operator (namely, addition and multiplication). In fact, it does not have an immediate divide-and-conquer implementation: there is no operator $\oplus$ that satisfies $\text{poly } x (l \# r) = \text{poly } x l \oplus \text{poly } x r$. Therefore, its parallel implementation is nontrivial. A known parallel implementation uses the length of the input list in addition to the value of *poly*. More formally, the parallel implementation is specified by the following $\text{pl } x y = (\text{poly } x y, \text{length } y)$:

$$\begin{aligned} \text{pl } x (l \# r) = & \text{let } (lp, ll) = \text{pl } x l \\ & (rp, rl) = \text{pl } x r \\ & \text{in } (lp + x^{ll} \times rp, ll + rl). \end{aligned}$$

This parallel implementation appears very different from the original *poly* function.

We hope for parallel programming environments to support a wide variety of nontrivial reductions that real programs may contain, including those with more than one operator like *poly*, those using control operators such as **break** (Figure 1 (b)), those with prefix-sum patterns that calculate not only the summary but also all intermediate results (Figure 1 (c)), and those traversing nonlinear structures such as trees (Figure 1 (d)). Although there have been many studies on systematically developing parallel reductions [Chi and Mu 2011; Chin et al. 1998; Deitz et al. 2006; Emoto et al. 2012, 2010; Farzan and Nicolet 2017; Fedyukovich et al. 2017; Fisher and Ghuloum 1994; Gorlatch 1999; Hu et al. 1997, 1998; Jiang et al. 2018; Matsuzaki et al. 2005, 2006; Morihata and Matsuzaki 2010, 2011; Morihata et al. 2009; Morita et al. 2007; Raychev et al. 2015; Sato and Iwasaki 2011; Suganuma et al. 1996; Xu et al. 2004], those studies consider only specific forms of reductions, and none of them can uniformly deal with all the kinds of reductions shown in Figure 1.

This paper introduces a calculus named λ^{AS} , a simply typed lambda calculus with algebraic simplification. It is designed to provide a foundation for *systematically developing a variety of parallel reductions based on equational reasoning*. The central idea is to regard a parallel reduction as a simplification of functions by using algebraic properties such as associativity and commutativity. For example, consider calculating $\text{sum } [a_0, \dots, a_n]$. The sequential evaluation essentially corresponds to the following expression.

$$a_0 + (\dots (a_{n-1} + (a_n + 0)) \dots)$$

This is not suitable for parallel evaluation, because no independent subexpressions exist. It can be divided into a function and an argument, however, by inserting a lambda abstraction. Then, effective parallel evaluation is possible if the function part can be evaluated during evaluation of the argument. For this example, the function part can be simplified using the associativity of (+).

$$\begin{aligned} & a_0 + (\dots (a_{n-1} + (a_n + 0)) \dots) \\ = & \{ \text{inserting a lambda abstraction} \} \\ & (\lambda x. a_0 + (\dots (a_{k-1} + (a_k + x)) \dots)) (a_{k+1} + (\dots (a_{n-1} + (a_n + 0)) \dots)) \\ \Rightarrow & \{ \text{parallel evaluation} \} \\ & (\lambda x. a_0^k + x) a_{k+1}^n \quad \text{where } a_0^k = \sum_{0 \leq i \leq k} a_i \text{ and } a_{k+1}^n = \sum_{k+1 \leq i \leq n} a_i \end{aligned}$$

This understanding of parallel reduction is not new. It has been used for developing parallel reduction loops [Callahan 1992; Farzan and Nicolet 2017; Fisher and Ghuloum 1994; Jiang et al.

2018; Raychev et al. 2015; Sato and Iwasaki 2011], parallel list/tree reductions [Chin et al. 1998; Hu et al. 1998; Matsuzaki et al. 2005, 2006; Morihata and Matsuzaki 2010; Xu et al. 2004], and parallel querying on semistructured databases [Buneman et al. 2006; Cong et al. 2007, 2012]. Here, λ^{AS} integrates this idea into lambda calculi.

λ^{AS} is a simply typed lambda calculus extended with a special abstraction syntax, namely, δ abstraction. In λ^{AS} , a lambda-abstrated term, $\lambda x. e$, is a value; in other words, the body e is not evaluated until its argument is passed. A δ -abstrated term, $\delta x. e$, is not a value, however, and its body e is *simplified using algebraic properties before the arrival of its argument*. For example,

$$\delta x. x + 2 \times x$$

is not a value and is thus immediately evaluated to

$$\lambda x. 3 \times x.$$

Note that this evaluation may be performed at the same time as the evaluation of the argument. For instance,

$$(\delta x. x + 2 \times x) (2 + 5)$$

has potential for parallel evaluation, as the following evaluation process shows.

$$\begin{aligned} & (\delta x. x + 2 \times x) (2 + 5) \\ \Rightarrow & (\lambda x. 3 \times x) 7 && \text{(parallel evaluation)} \\ \rightarrow & 3 \times 7 \\ \rightarrow & 21 \end{aligned}$$

It is nontrivial to provide a good strategy for simplifying complex expressions. For example, $\delta x_1. \delta x_2. 8 \times ((-1) \times x_1 + x_2) + 5 \times (x_1 \times 3 + x_2 \times (-2))$ can be simplified to $\lambda x_1. \lambda x_2. 7 \times x_1 - 2 \times x_2$ by distributing $\times$ over $+$, whereas $\delta x. x^3 + 3 \times x^2 + 3 \times x + 1$ can be simplified to $\lambda x. (x + 1)^3$ by factorization. Even worse, an inappropriate simplification strategy may significantly decrease efficiency. For instance,

$$\delta x_1. \delta x_2. \dots \delta x_n. (1 + x_1) \times (1 + x_2) \times \dots \times (1 + x_n)$$

may be “simplified”, by distributing $\times$ over $+$, to an exponentially large expression,

$$\lambda x_1. \lambda x_2. \dots \lambda x_n. 1 + x_1 + \dots + x_n + x_1 \times x_2 + x_1 \times x_3 + \dots + x_1 \times x_2 \times \dots \times x_n.$$

To provide a simple and effective simplification strategy, λ^{AS} requires that simplifications must result in *linear* polynomials⁴. For example, $\delta x. \delta y. x \times y$ gets stuck, because its body contains a product of x and y . This linearity requirement is somewhat restrictive but beneficial from several aspects. First, simplifications can be easily achieved by distributing $\times$ over $+$ and then merging terms that have a common variable. Second, the result of the simplification is commonly small, because the size of a linear polynomial is at most proportional to the number of variables. Third, several studies [Emoto et al. 2012, 2010; Matsuzaki et al. 2006; Sato and Iwasaki 2011; Xu et al. 2004] pointed out that linear polynomials are expressive enough to capture a wide variety of parallel reductions. Therefore, the linearity requirement can be regarded as a guideline for developing efficient parallel reductions by introducing δ abstractions. To support such development, λ^{AS} has a type system that checks the linearity requirement.

Formalizing a new lambda calculus, λ^{AS} , should be an important step in developing a powerful reduction parallelization method for practical programming languages. The existing studies on parallel reduction suggest the hypothesis that the idea of using algebraic simplification for parallel reduction is independent of control structures or programming patterns. If this hypothesis is correct,

⁴This paper specifically uses the term “linear” to refer to the linearity of polynomials, and not to the “single-useness” of variables.

$$\begin{array}{ll}
\text{foldr } f \ e \ [] & = e \\
\text{foldr } f \ e \ (a : x) & = f \ a \ (\text{foldr } f \ x)
\end{array}
\qquad
\begin{array}{ll}
\text{foldl } f \ e \ [] & = e \\
\text{foldl } f \ e \ (a : x) & = \text{foldl } (f \ e \ a) \ x
\end{array}$$

Fig. 2. Definitions of standard functions.

it could be a valuable clue to a uniform approach for dealing with various language features and programming patterns used in practical programs. Typed lambda calculi are perfectly suitable for confirming this hypothesis: control structures can be encoded by higher-order expressions, whereas basic (i.e., base-type) computations are clearly distinguished from higher-order features.

This paper contains the following two major contributions.

- Design of λ^{AS} , a lambda calculus with algebraic simplification (Section 3): The type system of λ^{AS} guarantees progress, i.e., the effectiveness of simplifications. Its operational semantics shows that any typed λ^{AS} term is observationally equivalent to the corresponding term of the simply typed lambda calculus.
- Systematic development of a wide variety of parallel reductions by using λ^{AS} (Section 2): The paper discusses reduction patterns including all examples in Figure 1, and others, as well.

2 DEVELOPING COMPLEX PARALLEL REDUCTIONS BY λ^{AS}

This section informally introduces λ^{AS} and demonstrates its effectiveness thorough examples. Figure 2 lists standard functions used in this section. Later, Section 3 develops the formalism.

2.1 Flavor of λ^{AS}

The following is the syntax of λ^{AS} . The type, τ , is the same as that of the simply typed lambda calculus.

$$e ::= x \mid \lambda x^\tau. e \mid e \ e \mid c \mid e \oplus e \mid e \otimes e \mid \delta x^\mathcal{R}. e \mid \dots$$

λ^{AS} extends the simply-typed lambda calculus via the semiring operators, $\oplus$ and $\otimes$, on the carrier set $\mathcal{R}$ ($c \in \mathcal{R}$), and a δ abstraction, $\delta x^\mathcal{R}. e$. Other features, such as conditionals, algebraic datatypes, and recursion, can be added if they are consistent with lambda calculi and do not manipulate semiring values of type $\mathcal{R}$. In the following, such additional features are used where necessary and expressed by the syntax of Haskell.

A semiring abstracts the cooperation of two related operations such as addition and multiplication. For the time being, we consider the semiring of addition and multiplication on integers, i.e., $(\oplus) = (+)$, $(\otimes) = \times$, and $\mathcal{R} = \mathbb{Z}$. Other semirings are introduced as needed.

The operational semantics of λ^{AS} is the standard call-by-value reduction except for δ abstractions. On one hand, a δ abstraction is observationally equivalent to a lambda abstraction, i.e., $\delta x^\mathbb{Z}. e \equiv \lambda x^\mathbb{Z}. e$. Here, two terms are said to be observationally equivalent if they will be reduced to the same value for any surrounding context of the base type. On the other hand, the body of a δ abstraction, namely e in $\delta x^\mathbb{Z}. e$, is evaluated before the argument, x , is specified. The δ -abstracted variable should have the semiring type, $\mathcal{R}$. The type annotations for variables may be omitted if they are apparent from the context.

For example, as discussed in the introduction, for the following term,

$$(\delta x. x + 2 \times x) (2 + 5),$$

the function and argument can be evaluated in parallel. In the following, $\rightarrow$ denotes a reduction step (or possibly a series of them), and $\Rightarrow$ is used instead to emphasize possibilities for parallel evaluations.

$$\begin{aligned}
& (\delta x. x + 2 \times x) (2 + 5) \\
\Rightarrow & (\lambda x. 3 \times x) 7 \\
\rightarrow & 3 \times 7 \\
\rightarrow & 21
\end{aligned}$$

For simplifying the function part without knowing the value of the argument, an evaluation of λ^{AS} involves variables that are not bound to values yet. We call such variables *indeterminates*. λ^{AS} simplifies polynomials over indeterminates by using the algebraic properties of the semiring.

So long as a δ abstraction is not involved, the evaluation is carried out as usual. For example,

$$\lambda y. (\delta x. x + 2 \times x)$$

is not evaluated any further unless the argument is passed, whereas

$$\delta y. (\delta x. x + 2 \times x)$$

is not a value and is evaluated to

$$\lambda y. (\lambda x. 3 \times x).$$

For avoiding serious inefficiency, λ^{AS} requires that an evaluation inside a δ abstraction must result in a linear polynomial over indeterminates. For example,

$$\delta x. \delta y. x \times y$$

gets stuck, because the body is nonlinear, i.e., it involves a multiplication of indeterminates.

Note that the linearity is not a syntactic but semantic requirement. For example,

$$\lambda x. \delta y. x \times y$$

does not get stuck if a constant (i.e., a value that contains no indeterminate) is supplied as the argument; however, it does get stuck if the argument contains indeterminates.

λ^{AS} is associated with a type system that guarantees progress of computation. In other words, the type system of λ^{AS} rejects terms that may involve a multiplication of indeterminates. The rest of this section considers only typeable terms that cause neither non-termination nor errors.

2.2 Parallel let

Lambda calculi can encode useful programming constructs. For example, the non-recursive let expression can be defined as follows.

$$\text{let } x_1 = e_1 \square x_2 = e_2 \square \cdots \square x_k = e_k \text{ in } e_{k+1} \equiv (\lambda x_1. \lambda x_2. \cdots \lambda x_k. e_{k+1}) e_1 e_2 \cdots e_k$$

This simultaneously defines $x_1, \dots, x_k$, and thus, $e_1, \dots, e_k$ must not contain any of $x_1, \dots, x_k$. Accordingly, $e_1, \dots, e_k$ can be evaluated in parallel. This captures the usual **async-finish** pattern.

λ^{AS} is slightly more expressive. For instance, consider the following term.

$$\text{let } x = 3 + 5 \text{ in let } y = x \times (7 + 3) \text{ in } x + y$$

As y depends on x , its parallel evaluation appears to be impossible. In λ^{AS} , by replacing the first **let** with **plet**, defined by

$$\text{plet } x = e_1 \text{ in } e_2 \equiv (\delta x. e_2) e_1,$$

a parallel evaluation becomes possible.

$$\begin{aligned}
& \text{plet } x = 3 + 5 \text{ in let } y = x \times (7 + 3) \text{ in } x + y \\
= & (\delta x. (\lambda y. x + y) (x \times (7 + 3))) (3 + 5) \\
\Rightarrow & (\lambda x. x \times 11) 8 \\
\rightarrow & 88
\end{aligned}$$

As seen in this example, the introduction of **plet**, or equivalently a δ abstraction, enables parallel evaluation regardless of data dependency. To see the effectiveness of **plet**, consider the following sequence of **let** expressions.

$$\text{let } x_0 = a_0 \text{ in let } x_1 = a_1 + x_0 \text{ in let } \cdots \text{ in let } x_n = a_n + x_{n-1} \text{ in } x_n$$

This program cannot gain any parallel speedup even by using **plet** instead of **let**, because each right-hand-side expression cannot be simplified further. Nevertheless, by inserting **plet**, it can be transformed into an equivalent program that is more suitable for parallel evaluation.

$$\begin{aligned} & \text{let } x_0 = a_0 \text{ in let } x_1 = a_1 + x_0 \text{ in let } \cdots \text{ in let } x_n = a_n + x_{n-1} \text{ in } x_n \\ = & \text{plet } z = (\text{let } x_0 = a_0 \text{ in let } x_1 = a_1 + x_0 \text{ in let } \cdots \text{ in let } x_k = a_k + x_{k-1} \text{ in } x_k) \\ & \text{in } (\text{let } x_{k+1} = a_{k+1} + z \text{ in let } \cdots \text{ in let } x_n = a_n + x_{n-1} \text{ in } x_n) \\ \Rightarrow & (\lambda x. a_{k+1}^n + x) a_0^k \quad \text{where } a_0^k = \sum_{0 \leq i \leq k} a_i \text{ and } a_{k+1}^n = \sum_{k+1 \leq i \leq n} a_i \end{aligned}$$

The introduction of **plet** thus breaks the data dependency and yields two terms that can be evaluated in parallel.

2.3 Parallel List Reduction

Now let us consider parallel list reductions.

Example 1 (Summation). We start with the simplest example, *sum*. Given lists l and r , the goal is to calculate $\text{sum } (l \# r)$ by processing l and r independently. This can be achieved by inserting a δ abstraction.

$$\begin{aligned} & \text{sum } (l \# r) \\ = & \{ \text{let } l = [a_0, a_1, \dots, a_m] \} \\ & a_0 + (a_1 + (\cdots (a_m + \text{sum } r) \cdots)) \\ = & \{ \text{introducing a } \delta \text{ abstraction} \} \\ & (\delta x. a_0 + (a_1 + (\cdots (a_m + x) \cdots))) (\text{sum } r) \\ = & \{ \text{introducing foldr} \} \\ & (\delta x. \text{foldr } (+) x l) (\text{sum } r) \end{aligned}$$

The derived implementation processes l and r in parallel, and its correctness follows from the equational reasoning. Moreover, the function part, $\delta x. \text{foldr } (+) x l$, can be effectively simplified to a linear polynomial of the form of $a + x$, where a is a constant and x is an indeterminate.

Example 2 (Polynomial Evaluation). The development of parallel *sum* can be generalized to several interesting applications. For example, recall the *poly* function discussed in the introduction. A calculation similar to the case of *sum* leads to the following program.

$$\text{poly } x (l \# r) = (\delta z. \text{foldr } (\lambda a. \lambda y. a + x \times y) z l) (\text{poly } x r)$$

The function part can be effectively simplified because its body forms a linear expression. For example, $\text{poly } 10 ([2, 1, 3] \# [5, 9])$ is evaluated as follows.

$$\begin{aligned} & \text{poly } 10 ([2, 1, 3] \# [5, 9]) \\ \rightarrow & (\delta z. \text{foldr } (\lambda a. \lambda y. a + 10 \times y) z [2, 1, 3]) (\text{poly } 10 [5, 9]) \\ \Rightarrow & (\delta z. (2 + 10 \times (1 + 10 \times (3 + 10 \times z)))) (5 + 10 \times (9 + 10 \times 0)) \\ \Rightarrow & (\lambda z. 312 + 1000 \times z) 95 \\ \rightarrow & 95312 \end{aligned}$$

Recall that the parallel implementation of *poly* discussed in the introduction additionally uses the length of the input list. In the development using λ^{AS} above, the function part is simplified to a linear expression that has two coefficients that correspond to the value of *poly* and the length of the input list.

The parallel implementation of *poly* calculates different kinds of results for l and r . While the result for l is a linear polynomial that consists of two coefficients, the result for r contains only one value. This is not problematic. As the following equation shows, any number of independent sublists can be processed by this implementation, in which $\circ$ denotes a function composition.

$$\begin{aligned} & \text{poly } x (y_0 \# y_1 \# \dots \# y_n) \\ &= ((\delta z. \text{poly}'_x z y_0) \circ (\delta z. \text{poly}'_x z y_1) \circ \dots \circ (\delta z. \text{poly}'_x z y_n)) (\text{poly } x []) \\ & \quad \text{where } \text{poly}'_x e w = \text{foldr } (\lambda a. \lambda y. a + x \times y) e w \end{aligned}$$

In the above expression, all poly'_x can be evaluated in parallel.

Example 3 (Maximum Prefix Sum). Given a list of numbers, maximum prefix sum [Hu et al. 1997; Morita et al. 2007] is the problem of finding the largest among the summations of prefixes of the list. For example, the maximum prefix sum of $[5, -2, 1, 6, -7, 3]$ is $5 + (-2) + 1 + 6 = 10$. The function to compute the maximum prefix sum, *mps*, is defined as follows, where $\uparrow$ is the binary maximum operator.

$$\begin{aligned} \text{mps } [] &= 0 \\ \text{mps } (a : x) &= 0 \uparrow (a + \text{mps } x) \end{aligned}$$

Note that $\uparrow$ and $+$ forms a semiring on $\mathbb{Z} \cup \{-\infty\}$, where $\uparrow$ is the addition and $+$ is the multiplication. Therefore, exactly the same process as in the case of *poly* gives the following parallel implementation of *mps*.

$$\text{mps } (l \# r) = (\delta z. \text{foldr } (\lambda a. \lambda y. 0 \uparrow (a + y)) z l) (\text{mps } r)$$

2.4 Loop

As the proposed approach is not specific to *foldr*, we next consider loops.

Example 4 (Summation by Loop). The first example is *sumL*, which calculates the summation by a loop.

$$\text{sumL} = \text{foldl } (+) 0$$

The function can be reasoned as follows.

$$\begin{aligned} & \text{sumL } (l \# r) \\ &= \{ \text{definition of } \text{sumL} \} \\ & \quad \text{foldl } (+) 0 (l \# r) \\ &= \{ \text{let } l = [a_0, a_1, \dots, a_m] \} \\ & \quad \text{foldl } (+) (\dots((0 + a_0) + a_1) \dots) + a_m) r \\ &= \{ \text{introducing } \delta \text{ abstraction} \} \\ & \quad (\delta x. \text{foldl } (+) x r) ((\dots((0 + a_0) + a_1) \dots) + a_m) \\ &= \{ \text{because } l = [a_0, a_1, \dots, a_m] \} \\ & \quad (\delta x. \text{foldl } (+) x r) (\text{sumL } l) \end{aligned}$$

As in the case of *sum*, the function part and the argument can be evaluated in parallel because the function part, $\delta x. \text{foldl } (+) x r$, can be simplified to a linear polynomial of the form of $x + a$, where a is a constant and x is an indeterminate.

Example 5 (Loop with Jump). A more complicated example is a loop with jumps. Consider *sumB1* in Figure 1 (b), which sums up all elements until encountering a negative element. This is a typical example of a loop with a break statement. The presence of the break causes the computation result to depend on the order of the elements. Nevertheless, its parallelization is possible.

$$\begin{aligned}
& \text{sumB1 } e \ (l \uplus r) \\
= & \ \{ \text{suppose } l = [a_0, a_1] \} \\
& \text{if } a_0 < 0 \text{ then } e \text{ else if } a_1 < 0 \text{ then } e + a_0 \text{ else } \text{sumB1 } ((e + a_0) + a_1) \ r \\
= & \ \{ \text{introducing abstractions} \} \\
& (\lambda f. \text{ if } a_0 < 0 \text{ then } e \text{ else if } a_1 < 0 \text{ then } e + a_0 \text{ else } f \ ((e + a_0) + a_1)) \ (\delta x. \text{sumB1 } x \ r)
\end{aligned}$$

The equational reasoning above derives a term in which the left sublist, l , and the right sublist, r are contained in independent subexpressions. Unfortunately, the function part cannot be simplified, because it uses not a δ abstraction but a lambda abstraction; moreover, because the abstraction binds a function, it cannot be replaced by a δ abstraction. In fact, however, this problem is not essential. The lambda-abstracted function, f , is used last; therefore, it is sufficient to take the function after finishing list processing.

$$\begin{aligned}
& \text{sumB1 } e \ (l \uplus r) \\
= & \ \{ \text{suppose } l = [a_0, a_1]; \text{ reasoning above} \} \\
& (\lambda f. \text{ if } a_0 < 0 \text{ then } e \text{ else if } a_1 < 0 \text{ then } e + a_0 \text{ else } f \ ((e + a_0) + a_1)) \ (\delta x. \text{sumB1 } x \ r) \\
= & \ \{ \text{distribute lambda abstraction to conditional branches} \} \\
& (\text{if } a_0 < 0 \text{ then } \lambda f. e \text{ else if } a_1 < 0 \text{ then } \lambda f. e + a_0 \text{ else } \lambda f. f \ ((e + a_0) + a_1)) \ (\delta x. \text{sumB1 } x \ r) \\
= & \ \{ \text{introducing } \text{sumB1}' \text{ defined below} \} \\
& (\text{sumB1}' \ e \ l) \ (\delta x. \text{sumB1 } x \ r)
\end{aligned}$$

Then, the left sublist is processed by the following $\text{sumB1}'$ during processing of the right sublist.

$$\begin{aligned}
\text{sumB1}' \ e \ [] &= \lambda f. e \\
\text{sumB1}' \ e \ (a : x) &= \text{if } a < 0 \text{ then } \lambda f. e \text{ else } \text{sumB1}' \ (e + a) \ x
\end{aligned}$$

We should distinguish the case of sumB1 from the following case of sumB2 , in which the computation terminates if the calculated value becomes negative.

$$\begin{aligned}
\text{sumB2 } e \ [] &= e \\
\text{sumB2 } e \ (a : x) &= \text{if } e < 0 \text{ then } e \text{ else } \text{sumB2 } (e + a) \ x
\end{aligned}$$

λ^{AS} does not allow this program, because the semiring value stored in e cannot be manipulated by $<$, which is not a semiring operator⁵. This is not, however, a limitation. In fact, sumB2 cannot be parallelized by a simple divide-and-conquer approach as in the case of sumB1 . In this way, λ^{AS} provides a guideline for reduction parallelization.

2.5 Prefix Sum

Prefix sums, also known as scans [Blelloch 1993; Ladner and Fischer 1980], are also important idiomatic patterns in parallel programming. Prefix sums are somewhat similar to reductions, but record all the intermediate results of a reduction. The function psum in Figure 1 (c) is a typical example.

$$\text{psum } e \ [a_0, a_1, \dots, a_n] = [e, e + a_0, e + a_0 + a_1, \dots, e + a_0 + \dots + a_{n-1}]$$

The following equational reasoning leads to a parallel implementation.⁶

⁵Strictly speaking, sumB1 is also not typeable, because each list element, a , is accessed by $<$, and moreover, a should have the semiring type. We can avoid this problem by using a function that translates usual values to semiring values. For example, by using $\text{lift}_{\mathbb{Z}} \mathbb{Z} \rightarrow \mathcal{R}$, sumL could be defined as $\text{sumL} = \text{foldl } (\lambda y. \lambda a. y + \text{lift}_{\mathbb{Z}} a) (\text{lift}_{\mathbb{Z}} 0)$. For simplicity of presentation, this paper neglects this issue.

⁶Strictly speaking, the derived program violates a restriction of λ^{AS} by applying a non-semiring operator (in this case, the list constructor) to δ abstracted values. Because a constructor application does not cause essential computation, this is not problematic. This issue can be avoided by using Church encoding for the output list.

$$\begin{aligned}
& psum\ (l \# r)\ e \\
= & \{ \text{let } l = [a_0, a_1, \dots, a_m] \} \\
& e : (e + a_0) : \dots : (e + a_0 + \dots + a_{m-1}) : psum\ r\ (e + a_0 + \dots + a_m) \\
= & \{ \text{definition of } psum \text{ and } sumL \} \\
& psum\ l\ e \# psum\ r\ (sumL\ e\ l) \\
= & \{ \text{introducing a } \delta \text{ abstraction} \} \\
& psum\ (l \# r)\ e = psum\ l\ e \# (\delta e.\ psum\ r\ e)\ (sumL\ e\ l)
\end{aligned}$$

Note that the final step, introducing a δ abstraction, is essential. The δ abstraction breaks the dependency from the computation for l (i.e., $sumL\ e\ l$) to the computation for r (i.e., $psum\ r$). This parallel implementation works well.

$$\begin{aligned}
& psum\ [3, 1, 7, 5, 2, 6, 9, 4]\ 0 \\
\rightarrow & psum\ [3, 1]\ 0 \# (\delta e.\ psum\ [7, 5, 2, 6, 9, 4]\ e)\ (sumL\ 0\ [3, 1]) \\
\rightarrow & psum\ [3, 1]\ 0 \# (\delta e.\ psum\ [7, 5]\ e \# (\delta e.\ psum\ [2, 6, 9, 4]\ e)\ (sumL\ [7, 5]\ e))\ (sumL\ 0\ [3, 1]) \\
\rightarrow & psum\ [3, 1]\ 0 \# (\delta e.\ psum\ [7, 5]\ e \# (\delta e.\ psum\ [2, 6]\ e \# (\delta e.\ psum\ [9, 4]\ e)\ (sumL\ [2, 6]\ e))\ (sumL\ [7, 5]\ e))\ (sumL\ 0\ [3, 1]) \\
\Rightarrow & [0, 3] \# (\lambda e.\ [e, e + 7] \# (\delta e.\ [e, e + 2] \# (\delta e.\ [e, e + 9])\ (e + 8))\ (e + 12))\ 4 \\
\rightarrow & [0, 3] \# [4, 4 + 7] \# [16, 16 + 2] \# [24, 24 + 9] \\
\Rightarrow & [0, 3] \# [4, 11] \# [16, 18] \# [24, 33]
\end{aligned}$$

The implementation consists of three major steps. First, prefix-sum computation is applied to every sublist. At the same time, the total sum of each sublist is calculated. Then, the total sum is propagated globally by resolving the lambda abstraction. Finally, the propagated values are supplied to each element, resulting in the final output.

There is another known parallel prefix sum algorithm, which delays the prefix-sum computation until the total sum is propagated. This algorithm can be expressed by the following program.

$$\begin{aligned}
psum\ (l \# r) = & \text{let } s = \delta e.\ sumL\ e\ l \\
& \square\ r' = \delta e.\ psum\ r\ e \\
& \text{in } \lambda e.\ psum\ l\ e \# r'\ (s\ e)
\end{aligned}$$

It is fairly easy to check that this implementation is observationally equivalent to the previous one. Once we regard δ abstractions as λ abstractions, standard reasoning on a lambda calculus shows their equivalence.

$$\begin{aligned}
& psum\ (l \# r) \\
= & \{ \text{the first parallel implementation} \} \\
& \lambda e.\ psum\ l\ e \# (\delta e.\ psum\ r\ e)\ (sumL\ e\ l) \\
= & \{ \text{introducing } \delta \text{ abstraction} \} \\
& \lambda e.\ psum\ l\ e \# (\delta e.\ psum\ r\ e)\ ((\delta e.\ sumL\ e\ l)\ e) \\
= & \{ \text{introducing let} \} \\
& \lambda e.\ (\text{let } s = \delta e.\ sumL\ e\ l \square\ r' = \delta e.\ psum\ r\ e \text{ in } psum\ l\ e \# r'\ (s\ e)) \\
= & \{ \text{swapping let and the outermost } \lambda \text{ abstraction} \} \\
& \text{let } s = \delta e.\ sumL\ e\ l \square\ r' = \delta e.\ psum\ r\ e \text{ in } \lambda e.\ psum\ l\ e \# r'\ (s\ e)
\end{aligned}$$

Nevertheless, this implementation shows a different behavior from that of the previous one.

```

    psum [3, 1, 7, 5, 2, 6, 9, 4] 0
→ (let s = δe. sumL e [3, 1] [] r' = δe. psum [7, 5, 2, 6, 9, 4] e
   in λe. psum [3, 1] 0 + r' (s e)) 0
→ (let s = δe. sumL e [3, 1]
    [] r' = δe. (let s = δe. sumL e [7, 5]
                 [] r' = δe. (let s = δe. sumL e [2, 6] [] r' = δe. psum [9, 4] e
                              in λe. psum [2, 6] e + r' (s e)) e
                 in λe. psum [7, 5] e + r' (s e)) e
   in λe. psum [3, 1] e + r' (s e)) 0
⇒ (λe. psum [3, 1] e + (λe. psum [7, 5] e +
                        (λe. psum [2, 6] e +
                         (λe. [e, e + 9]) ((λe. e + 8) e)) ((λe. e + 12) e)) ((λe. e + 4) e)) 0
→ psum [3, 1] 0 + psum [7, 5] 4 + psum [2, 6] 16 + (λe. [e, e + 9]) 24
⇒ [0, 3] + [4, 11] + [16, 18] + [24, 33]

```

Although rather complicated, this implementation also consists of three major steps. First, the summation is calculated for every sublist. Then, the calculated value is propagated globally by resolving the lambda abstraction. Finally, prefix sum computation is applied to each sublist.

The discussions to this point demonstrate a typical use scenario for λ^{AS} . Several parallel implementations of reduction-related computations can be developed by introducing δ abstractions, and their correctness can easily be checked by equational reasoning. In this way, λ^{AS} supports *reduction parallelization* rather than providing parallel reductions as a primitive parallel computation pattern.

2.6 Beyond List Processing

Existing reduction parallelization methods mainly consider list/array processing. By virtue of the expressiveness of λ^{AS} , it can also deal with programs that process data structures other than lists.

Example 6 (Bottom-Up Tree Processing). It appears straightforward to evaluate bottom-up tree processing in parallel. For example, the following sum_{Tree} can process independent subtrees in parallel.

$$\begin{aligned}
 \text{sum}_{\text{Tree}} (\text{Nd } n \ l \ r) &= n + \text{sum}_{\text{Tree}} l + \text{sum}_{\text{Tree}} r \\
 \text{sum}_{\text{Tree}} (\text{Lf } n) &= n
 \end{aligned}$$

This naive approach cannot achieve sufficient parallel speedup, however, if the input is a list-like tall tree. This limitation is not insignificant, because practical tree structures, such as XML data and syntax trees, are very often list-like.

λ^{AS} enables a bold approach. Given a tree t , consider dividing t in the middle such that $t = c[t']$, where t' is a subtree of t , c is a tree context that has a unique “hole” denoted by $\bullet$, and $c[t']$ denotes the tree obtained by substituting t' for the hole in c . The goal is to develop a function $\text{sum}'_{\text{Tree}}$ that satisfies the following equation.

$$\text{sum}_{\text{Tree}} (c[t']) = (\delta x. \text{sum}'_{\text{Tree}} c \ x) (\text{sum}_{\text{Tree}} t')$$

Equational reasoning easily leads to the definition of $\text{sum}'_{\text{Tree}}$. In the following, we use $c[\bullet]$ instead of c to express that c is not a tree but a context.

$$\begin{aligned}
& \text{sum}'_{\text{Tree}} c \ x \\
= & \{ \text{the characteristic equation above} \} \\
& \text{sum}_{\text{Tree}} (c[t']) \quad \text{where } x = \text{sum}_{\text{Tree}} t' \\
= & \{ \text{suppose } c[t'] = t' \text{ (i.e., } c = \bullet) \} \\
& \text{sum}_{\text{Tree}} t' \quad \text{where } x = \text{sum}_{\text{Tree}} t' \\
= & \{ \text{definition of } x \} \\
& x \\
& \text{sum}'_{\text{Tree}} c \ x \\
= & \{ \text{the characteristic equation above} \} \\
& \text{sum}_{\text{Tree}} (c[t']) \quad \text{where } x = \text{sum}_{\text{Tree}} t' \\
= & \{ \text{suppose } c[t'] = \text{Nd } n \ (l[t']) \ r \} \\
& n + \text{sum}_{\text{Tree}} (l[t']) + \text{sum}_{\text{Tree}} r \quad \text{where } x = \text{sum}_{\text{Tree}} t' \\
= & \{ \text{induction} \} \\
& n + \text{sum}'_{\text{Tree}} l \ x + \text{sum}_{\text{Tree}} r
\end{aligned}$$

The case of $c[t'] = \text{Nd } n \ l \ r([t'])$ is similar. In summary, the following definition is obtained.

$$\begin{aligned}
\text{sum}'_{\text{Tree}} \bullet \ x &= x \\
\text{sum}'_{\text{Tree}} (\text{Nd } n \ (l[\bullet]) \ r) \ x &= n + \text{sum}'_{\text{Tree}} (l[\bullet]) \ x + \text{sum}_{\text{Tree}} r \\
\text{sum}'_{\text{Tree}} (\text{Nd } n \ l \ (r[\bullet])) \ x &= n + \text{sum}_{\text{Tree}} l + \text{sum}'_{\text{Tree}} (r[\bullet]) \ x
\end{aligned}$$

Because the computation of $\text{sum}'_{\text{Tree}}$ consists of additions, $\text{sum}'_{\text{Tree}} c$ can be computed independently with $\text{sum}_{\text{Tree}} t'$.

The definition of $\text{sum}'_{\text{Tree}}$ shows the possibility of processing independent subtrees in parallel; therefore, subtrees can be recursively divided. Moreover, as the following reasoning shows, even contexts can be recursively divided. Accordingly, this approach of dividing a tree into a context and a subtree can lead to substructures of similar sizes, thereby providing a good load balancing even for list-like trees.

$$\begin{aligned}
& \text{sum}'_{\text{Tree}} (c_1[c_2[\bullet]]) \ x \\
= & \{ \text{the characteristic equation} \} \\
& \text{sum}_{\text{Tree}} (c_1(c_2[t'])) \quad \text{where } x = \text{sum}_{\text{Tree}} t' \\
= & \{ \text{the characteristic equation} \} \\
& \text{sum}'_{\text{Tree}} c_1 \ x' \quad \text{where } x' = \text{sum}_{\text{Tree}} (c_2[t']) \\
= & \{ \text{the characteristic equation} \} \\
& \text{sum}'_{\text{Tree}} c_1 \ x' \quad \text{where } x' = \text{sum}'_{\text{Tree}} c_2 \ x \\
= & \{ \text{introducing } \delta \text{ abstraction} \} \\
& (\delta z. \text{sum}'_{\text{Tree}} c_1 \ z) (\text{sum}'_{\text{Tree}} c_2 \ x)
\end{aligned}$$

Note that this approach is a generalization of the divide-and-conquer approach for parallel list processing. If x is a list, then $x = c[x']$ is equivalent to $x = c \# x'$. In addition, the approach also generalizes the naive bottom-up tree processing, which divides $\text{Nd } n \ l \ r$ into $\text{Nd } n \ [\bullet] \ r$ and l .

Example 7 (Complex Tree Processing with Accumulations). The next example is a more complex case of tree processing: *rd* shown in Figure 1 (d). The program expresses a reaching definition analysis of a simple imperative program. Assign $v \ e$, Seq $s_1 \ s_2$, If $e \ s_1 \ s_2$, and while $e \ s$ respectively denote an assignment statement like $v := e$, a sequential statement like $s_1; s_2$, a conditional statement like if $(e) \ s_1 \text{ else } s_2$, and a loop statement like while $(e) \ s$. The function *remove* $v \ y$ removes definitions of variable v from the set of definitions, y .

The program of *rd* appears unsuitable to parallel processing. In the case of Seq, a computation of a subtree depends on the result of another subtree via an accumulation parameter. In λ^{AS} , however,

the dependency can be broken by introducing δ abstraction:

$$rd (\text{Seq } s_1 s_2) y = (\delta y. rd s_2 y) (rd s_1 y).$$

Therefore, the dependency is not problematic if the computation of rd involves only semiring operators. Consider a semiring whose carriers are bit vectors such that each bit corresponds to a variable in the program and whose operators are the bitwise logical OR operator $\vee$ and the bitwise logical AND operator $\wedge$. Then, the computation of rd can be expressed via a semiring: $\cup$, $\cap$, and $remove\ v\ y$ can be regarded as $\vee$, $\wedge$, and $\neg v \wedge y$, respectively, where $\neg v$ is a bit vector with each bit set to 1 except for the bit corresponding to v . In summary, the introduction of δ abstraction is safe and leads to parallel evaluation of rd .

As in the case of sum_{Tree} , dividing a syntax tree into a context and a subtree may improve load balancing. The situation here is more difficult, however, than the case of sum_{Tree} . Given a tree $c[t]$, the accumulation parameter for processing t depends on $c[\bullet]$; therefore, the computations of $c[\bullet]$ and t are mutually dependent. The dependency can be solved if the computation for $c[\bullet]$ returns two values: an accumulation parameter y' passed to t , and a function f_c that takes the result of t .

$$rd (c[t]) y = \text{let } (f_c, y') = rd' c\ y \ \parallel \ f_t = \delta y. rd\ t\ y \text{ in } f_c (f_t y')$$

We can develop the definition of rd' by equational reasoning. The objective is to find f_c and y' such that $rd (c[t]) y = f_c (rd\ t\ y')$.

$$\begin{aligned} & rd (c[t]) y \\ &= \{ \text{suppose } c[t] = \text{Seq } (s_1[t]) s_2 \} \\ & \quad rd\ s_2 (rd (s_1[t]) y) \\ &= \{ \text{induction} \} \\ & \quad \text{let } (f_c, y') = rd' s_1\ y \text{ in } rd\ s_2 (f_c (rd\ t\ y')) \\ &= \{ \text{introducing } \delta \text{ abstraction} \} \\ & \quad \text{let } (f_c, y') = rd' s_1\ y \text{ in } (\delta z. rd\ s_2\ z) (f_c (rd\ t\ y')) \\ &= \{ \text{introducing let} \} \\ & \quad \text{let } (f_c, y') = rd' s_1\ y \ \parallel \ f_t = \delta z. rd\ s_2\ z \text{ in } f_t (f_c (rd\ t\ y')) \end{aligned}$$

The reasoning to this point leads to the following equation.

$$rd' (\text{Seq } (s_1[\bullet]) s_2) y = \text{let } (f_c, y') = rd' (s_1[\bullet]) y \ \parallel \ f_t = \delta z. rd\ s_2\ z \text{ in } (\delta z. f_t (f_c z), y')$$

The other cases can be similarly dealt with. The result is the following.

$$\begin{aligned} rd' \bullet y &= (\lambda z. z, y) \\ rd' (\text{Seq } (s_1[\bullet]) s_2) y &= \text{let } (f_c, y') = rd' (s_1[\bullet]) y \ \parallel \ f_t = \delta z. rd\ s_2\ z \text{ in } (\delta z. f_t (f_c z), y') \\ rd' (\text{Seq } s_1 (s_2[\bullet])) y &= (\delta z. rd' (s_2[\bullet]) z) (rd\ s_1\ y) \\ rd' (\text{If } e (s_1[\bullet]) s_2) y &= \text{let } (f_c, y') = rd' (s_1[\bullet]) y \ \parallel \ y'' = rd\ s_2\ y \text{ in } (\delta z. f_c z \cup y'', y') \\ rd' (\text{If } e\ s_1 (s_2[\bullet])) y &= \text{let } y' = rd\ s_1\ y \ \parallel \ (f_c, y'') = rd' (s_2[\bullet]) y \text{ in } (\delta z. y' \cup f_c z, y'') \\ rd' (\text{While } e (s[\bullet])) y &= \text{let } (f_c, y') = rd' (s[\bullet]) y \text{ in } (\delta z. y \cup f_c z, y') \end{aligned}$$

It is not easy to understand the behavior of this implementation. Nevertheless, the equational reasoning certifies its correctness; moreover, the type system guarantees the linearity of polynomials and thereby the efficiency of its parallel evaluation.

Example 8 (Recurrence Equation). As a final example, consider a purely numerical computation: calculating a numerical sequence defined by the following recurrence equation, which generalizes calculation of the Fibonacci numbers.

$$\begin{aligned} f\ 0 &= m_0 \\ f\ 1 &= m_1 \\ f\ n &= a \times f\ (n - 1) + b \times f\ n + c \end{aligned}$$

It is well known that the following program provides a linear-time implementation.

$$\begin{aligned} f\ n &= \text{let } (m_{n-1}, m_n) = f'\ (n-1) \text{ in } m_n \\ f'\ 0 &= (m_0, m_1) \\ f'\ n &= \text{let } (m_{n-1}, m_n) = f'\ (n-1) \text{ in } (m_n, a \times m_{n-1} + b \times m_n + c) \end{aligned}$$

λ^{AS} can then be used for developing a divide-and-conquer implementation.

$$\begin{aligned} & f'\ (n+k) \\ &= \{ \text{let } g\ (m_{n-1}, m_n) = (m_n, a \times m_{n-1} + b \times m_n + c) \} \\ & \quad \underbrace{g\ (g\ (\dots (g\ (f'\ n) \dots))}_{k} \\ &= \{ \text{let } f''\ k\ v_1\ v_2 = \text{if } k \equiv 0 \text{ then } (v_1, v_2) \text{ else } g\ (f''\ (k-1)\ v_1\ v_2) \} \\ & \quad \text{let } (m_{n-1}, m_n) = f'\ n \parallel f'_k = \delta x. \delta y. f''\ k\ x\ y \text{ in } f'_k\ m_{n-1}\ m_n \end{aligned}$$

Because the computation of f'' consists of additions and multiplications, the introduction of δ abstractions is valid. Then, $f'\ n$ and $f''\ k$ can be calculated in parallel.

In fact, parallel computations are unnecessary for this case.

$$\begin{aligned} & f'\ (n+n) \\ &= \{ \text{parallel implementation of } f' \} \\ & \quad \text{let } (m_{n-1}, m_n) = f'\ n \parallel f'_n = \delta x. \delta y. f''\ n\ x\ y \text{ in } f'_n\ m_{n-1}\ m_n \\ &= \{ \text{unfolding } f' \text{ once more; note } f'\ 0 = (m_0, m_1) \} \\ & \quad \text{let } (m_{n-1}, m_n) = (\text{let } f'_n = \delta x. \delta y. f''\ n\ x\ y \text{ in } f'_n\ m_0\ m_1) \parallel f'_n = \delta x. \delta y. f''\ n\ x\ y \\ & \quad \text{in } f'_n\ m_{n-1}\ m_n \\ &= \{ \text{common subexpression elimination} \} \\ & \quad \text{let } f'_n = \delta x. \delta y. f''\ n\ x\ y \text{ in let } (m_{n-1}, m_n) = f'_n\ m_0\ m_1 \text{ in } f'_n\ m_{n-1}\ m_n \end{aligned}$$

The computational cost of the obtained recursive program is $O(\log n)$. This example shows the possibility of using λ^{AS} beyond parallel processing.

3 FORMAL DEFINITION OF λ^{AS}

3.1 Preliminaries: Semirings and Linear Polynomials

λ^{AS} is based on semirings. Formally, a semiring $(S, \oplus, \otimes, \bar{0}, \bar{1})$ is a five-tuple, where S is the set of values, $\oplus$ and $\otimes$ are binary operators over S , $\bar{0}$ and $\bar{1}$ are elements of S , and the following properties hold.

$$\begin{array}{ll} a \oplus (b \oplus c) = (a \oplus b) \oplus c & \{ \text{associativity of } \oplus \} \\ a \oplus b = b \oplus a & \{ \text{commutativity of } \oplus \} \\ a \oplus \bar{0} = \bar{0} \oplus a = a & \{ \text{unit of } \oplus \} \\ a \otimes (b \otimes c) = (a \otimes b) \otimes c & \{ \text{associativity of } \otimes \} \\ a \otimes \bar{1} = \bar{1} \otimes a = a & \{ \text{unit of } \otimes \} \\ a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c) & \{ \text{left distributivity} \} \\ (b \oplus c) \otimes a = (b \otimes a) \oplus (c \otimes a) & \{ \text{right distributivity} \} \\ a \otimes \bar{0} = \bar{0} \otimes a = \bar{0} & \{ \text{zero} \} \end{array}$$

Section 2 introduced the following semirings: 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 bit vectors $(\{0, 1\}^n, \vee, \wedge, \vec{0}, \vec{1})$, where $\{0, 1\}^n$ is the set of n -bits vectors, $\vec{0}$ is the vector with each bit set to 0, and $\vec{1}$ is the vector with each bit set to 1.

For a semiring $\mathcal{R} = (S, \oplus, \otimes, \bar{0}, \bar{1})$, we may use $\mathcal{R}$ and S 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 indeterminates X and a semiring $\mathcal{R} = (S, \oplus, \otimes, \bar{0}, \bar{1})$, a polynomial of the following form, where $c_0, c_1, \dots, c_m \in S$ and $x_1, x_2, \dots, x_m \in X$,

$$c_0 \oplus (c_1 \otimes x_1) \oplus \dots \oplus (c_m \otimes x_m),$$

is called a *left-linear polynomial* over $(\mathcal{R}, X)$. We may omit $\mathcal{R}$ and X if they are clear from the context. Similarly, a polynomial of the following form,

$$c_0 \oplus (x_1 \otimes c_1) \oplus \dots \oplus (x_m \otimes c_m),$$

is called a *right-linear polynomial*. When $\otimes$ is commutative, left- and right-linear polynomials coincide and are called *linear polynomials*.

3.2 Syntax and Operational Semantics

For simplicity, this section considers λ^{AS} defined by the following syntax. Section 3.5 discusses further extensions.

$$\begin{aligned} e &::= x \mid \lambda x^{\tau}. e \mid e e \mid c \mid e \oplus e \mid e \otimes e \mid \delta x^{\mathcal{R}}. e \\ \tau &::= \mathcal{R}_{\alpha} \mid \tau \rightarrow \tau \\ \alpha &::= C \mid P \end{aligned}$$

A metavariable x is used to denote a variable (or indeterminate). $\mathcal{R}$ denotes the underlying semiring, and c is a value in $\mathcal{R}$. Each base type, $\mathcal{R}$, is annotated by either P (polynomial) or C (constant). Later, Section 3.3 explains the meanings of these annotations.

Values in λ^{AS} are defined as follows. For now, $\otimes$ is assumed to be commutative, and thus, only linear polynomials are considered.

$$v ::= c_0 \oplus (c_1 \otimes x_1) \oplus \dots \oplus (c_m \otimes x_m) \mid \lambda x^{\tau}. e$$

Values are functions and linear polynomials, and constants are special cases of linear polynomials. Note that δ abstractions are *not* values.

The operational semantics is defined by the set of reduction rules shown in Figure 3, in which $e[v/x]$ denotes the capture-avoiding substitution of v to x in e . The first four rules are the same as those of the usual call-by-value simply typed lambda calculus. The fifth and sixth rules simplify the body of a δ abstraction. A δ abstraction becomes a λ abstraction if the body is completely simplified. Linear polynomials are simplified according to the algebraic properties of the semiring. We assume that every linear polynomial contains the same set of indeterminates. Because an indeterminate can be introduced to a polynomial by associating it with a zero coefficient, this assumption is not restrictive. To keep linearity, at least one operand of multiplication must be a constant.

3.3 Type System

Figure 4 shows the typing rules of λ^{AS} . An environment Γ maps a variable to its type. $\Gamma\{x : \tau\}$ denotes an extension of Γ by a binding $x : \tau$, i.e., $\Gamma\{x : \tau\}(x) = \tau$, and $\Gamma\{x : \tau\}(y) = \Gamma(y)$ if $x \neq y$. A λ^{AS} term e is said to be typeable if there exist an environment Γ and a type τ such that $\Gamma \vdash e : \tau$.

The typing rules contain two key differences from those of the simply typed lambda calculus. First, each base type is annotated by either P or C. In the rules, a metavariable α is used to denote P or C. A term of type $\mathcal{R}_C$ should be reduced to a constant that contains no indeterminate. The annotations are used for guaranteeing the safety of multiplication, in which the operands must contain a constant. Second, a special rule is prepared for $\delta x^{\mathcal{R}}. e$. Because e is to be simplified before the argument is passed, e should be typeable even if x is an indeterminate and therefore has the $\mathcal{R}_P$ type. Moreover, because $\delta x^{\mathcal{R}}. e$ is regarded as a usual function after the simplification of e , $\delta x^{\mathcal{R}}. e$ should have the same type as $\lambda x^{\mathcal{R}}. e$. Accordingly, the body, e , is typechecked twice. Note that the

$$\begin{aligned}
& (\lambda x^\tau. e) v \rightarrow e[v/x] \\
& e_1 e_2 \rightarrow e'_1 e'_2 \quad \text{if } e_1 \rightarrow e'_1 \text{ and } e_2 \rightarrow e'_2, \text{ or, } e_i = e'_i \text{ and } e_j \rightarrow e'_j \text{ (} i, j \in \{1, 2\}, i \neq j \text{)} \\
& e_1 \oplus e_2 \rightarrow e'_1 \oplus e'_2 \quad \text{if } e_1 \rightarrow e'_1 \text{ and } e_2 \rightarrow e'_2, \text{ or, } e_i = e'_i \text{ and } e_j \rightarrow e'_j \text{ (} i, j \in \{1, 2\}, i \neq j \text{)} \\
& e_1 \otimes e_2 \rightarrow e'_1 \otimes e'_2 \quad \text{if } e_1 \rightarrow e'_1 \text{ and } e_2 \rightarrow e'_2, \text{ or, } e_i = e'_i \text{ and } e_j \rightarrow e'_j \text{ (} i, j \in \{1, 2\}, i \neq j \text{)} \\
& \delta x. e \rightarrow \delta x. e' \quad \text{if } e \rightarrow e' \\
& \delta x. v \rightarrow \lambda x. v \\
& (c_0 \oplus (c_1 \otimes x_1) \oplus \cdots \oplus (c_m \otimes x_m)) \oplus (c'_0 \oplus (c'_1 \otimes x_1) \oplus \cdots \oplus (c'_m \otimes x_m)) \\
& \quad \rightarrow c''_0 \oplus (c''_1 \otimes x_1) \oplus \cdots \oplus (c''_m \otimes x_m) \quad \text{where } \mathcal{R} \ni c''_i = c_i \oplus c'_i \text{ (} 0 \leq i \leq m \text{)} \\
& c_0 \otimes (c'_0 \oplus (c'_1 \otimes x_1) \oplus \cdots \oplus (c'_m \otimes x_m)) \\
& \quad \rightarrow c''_0 \oplus (c''_1 \otimes x_1) \oplus \cdots \oplus (c''_m \otimes x_m) \quad \text{where } \mathcal{R} \ni c''_i = c_0 \otimes c'_i \text{ (} 0 \leq i \leq m \text{)} \\
& (c_0 \oplus (c_1 \otimes x_1) \oplus \cdots \oplus (c_m \otimes x_m)) \otimes c'_0 \\
& \quad \rightarrow c''_0 \oplus (c''_1 \otimes x_1) \oplus \cdots \oplus (c''_m \otimes x_m) \quad \text{where } \mathcal{R} \ni c''_i = c_i \otimes c'_0 \text{ (} 0 \leq i \leq m \text{)}
\end{aligned}$$

Fig. 3. Reduction rules for λ^{AS} .

$$\begin{aligned}
& \frac{\Gamma(x) = \tau}{\Gamma \vdash x^\tau : \tau} \quad \frac{\Gamma\{x : \tau\} \vdash e : \tau'}{\Gamma \vdash \lambda x^\tau. e : \tau \rightarrow \tau'} \quad \frac{\Gamma \vdash e_1 : \tau \rightarrow \tau' \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 e_2 : \tau'} \\
& \frac{\Gamma\{x : \mathcal{R}_\alpha\} \vdash e : \tau' \quad \Gamma\{x : \mathcal{R}_\beta\} \vdash e : \tau''}{\Gamma \vdash \delta x^{\mathcal{R}}. e : \mathcal{R}_\alpha \rightarrow \tau'} \quad \Gamma \vdash c : \mathcal{R}_\alpha \\
& \frac{\Gamma \vdash e_1 : \mathcal{R}_\alpha \quad \Gamma \vdash e_2 : \mathcal{R}_\alpha}{\Gamma \vdash e_1 \oplus e_2 : \mathcal{R}_\alpha} \quad \frac{\Gamma \vdash e_1 : \mathcal{R}_C \quad \Gamma \vdash e_2 : \mathcal{R}_\alpha}{\Gamma \vdash e_1 \otimes e_2 : \mathcal{R}_\alpha} \quad \frac{\Gamma \vdash e_1 : \mathcal{R}_\alpha \quad \Gamma \vdash e_2 : \mathcal{R}_C}{\Gamma \vdash e_1 \otimes e_2 : \mathcal{R}_\alpha}
\end{aligned}$$

Fig. 4. Typing rules for λ^{AS} .

following simpler rule is safe but too restrictive.

$$\frac{\Gamma\{x : \mathcal{R}_\beta\} \vdash e : \tau'}{\Gamma \vdash \delta x^{\mathcal{R}}. e : \mathcal{R}_\beta \rightarrow \tau'}$$

This rule regards nearly every δ -abstracted function as returning non-constants. For instance, it infers $\emptyset \vdash (\delta x^{\mathcal{R}}. x) : \mathcal{R}_\beta \rightarrow \mathcal{R}_\beta$ and thus rejects apparently safe terms such as $(\delta x^{\mathcal{R}}. x) 1 \times (\delta x^{\mathcal{R}}. x) 1$.

Except for these two differences, the typing rules of λ^{AS} are the same as those of the simply typed lambda calculus. A λ^{AS} term containing no δ abstraction is typeable if and only if it is typeable in the simply typed lambda calculus.

The following discussion considers only typeable λ^{AS} terms.

3.4 Properties

λ^{AS} can be regarded as a variant of the call-by-value simply typed lambda calculus with a speculative evaluation. The following theorem states that the speculative evaluation of λ^{AS} is not problematic because a δ abstraction is observationally equivalent to a λ abstraction.

The formalization uses the notion of contexts. A context of λ^{AS} is a λ^{AS} term that contain exactly one special variable $\bullet$. The following gives the definition.

$$C ::= \bullet \mid \lambda x^\tau. C \mid C e \mid e C \mid C \oplus e \mid e \oplus C \mid C \otimes e \mid e \otimes C \mid \delta x^{\mathcal{R}}. C$$

Here, $C[e]$ denotes a λ^{AS} term obtained by substituting $\bullet$ for e in C .

THEOREM 3.1. *Let $\rightarrow^*$ be the reflective transitive closure of $\rightarrow$. For any λ^{AS} term e and λ^{AS} context C , if $C[\delta x^{\mathcal{R}}. e] \rightarrow^* c_1$ and $C[\lambda x^{\mathcal{R}}. e] \rightarrow^* c_2$, where $c_1, c_2 \in \mathcal{R}$, then $c_1 = c_2$.*

PROOF. Consider another reduction relation $\rightarrow$, defined as follows.

- $e \rightarrow e'$ implies $e \rightarrow e'$.
- $e \rightarrow e'$ implies $C[e] \rightarrow C[e']$ for any context C .
- $\delta x^{\mathcal{R}}. e \rightarrow \lambda x^{\mathcal{R}}. e$.

Because $C[\delta s^{\mathcal{R}}. e] \rightarrow C[\lambda x^{\mathcal{R}}. e]$, the theorem follows from the confluence of $\rightarrow$. Note that $\rightarrow$ expresses β reductions with algebraic simplifications using semiring properties, and the algebraic simplification can be regarded as a confluent rewriting system. Because a rewriting system obtained by combining confluent rules with β reductions of the simply typed lambda calculus is confluent [Tannen 1988; Tannen and Gallier 1991], $\rightarrow$ is confluent. $\square$

Theorem 3.1 only considers successful evaluations that yield values. This is not a serious limitation if we consider only typeable terms. Evaluation of a typeable term never gets stuck, as we now prove.

First, the following lemma shows the relationship between the types and results of evaluations. Let $\text{fvs}(e)$ denote the set of all free variables in e , i.e., variables not bound by any λ abstraction or δ abstraction. We assume that every free variable is an indeterminate and thus has type $\mathcal{R}_P$, because free variables except for indeterminates cannot be values in λ^{AS} .

LEMMA 3.2. *Assume that $\Gamma \vdash v : \tau$, and that $\Gamma(x) = \mathcal{R}_P$ for any $x \in \text{fvs}(v)$; then, the following equations hold.*

- $v = c$ if $\tau = \mathcal{R}_C$.
- $v = c_0 \oplus (c_1 \otimes x_1) \oplus \dots \oplus (c_m \otimes x_m)$ if $\tau = \mathcal{R}_P$.
- $v = \lambda x^{\tau_1}. e'$ if $\tau = \tau_1 \rightarrow \tau_2$.

PROOF. The proof follows immediately from the typing rules of λ^{AS} . $\square$

Next, the following two theorems show that, for any typeable λ^{AS} term without free variables, its evaluation will not get stuck.

THEOREM 3.3. *If $\Gamma \vdash e : \tau$ and $e \rightarrow e'$, then $\Gamma \vdash e' : \tau$.*

PROOF. The proof follows straightforwardly from a case analysis over the rules of $\rightarrow$. The only nontrivial case is the beta reduction, $(\lambda x^{\tau}. e) v \rightarrow e[v/x]$. This case can be straightforwardly proved by an induction over the structure of e . $\square$

THEOREM 3.4. *If $\Gamma \vdash e : \tau$ and $\Gamma(x) = \mathcal{R}_P$ for any $x \in \text{fvs}(e)$, then there exists e' such that $e \rightarrow e'$ unless e is a value.*

PROOF. The proof uses an induction over the structure of e . Every case is easily proved by using Lemma 3.2. Note that it is safe to regard any $x \in \text{fvs}(e)$ as an indeterminate of a polynomial, because $\Gamma(x) = \mathcal{R}_P$. $\square$

3.5 Extending the Calculus

3.5.1 Non-commutative Multiplications. If $\otimes$ is not commutative, then simplifications become more difficult. For instance, neither $c \otimes x \otimes c'$ nor $(c_1 \otimes x \otimes c'_1) \oplus (c_2 \otimes x \otimes c'_2)$ can be simplified further. Therefore, to guarantee the simplicity of polynomials, the operational semantics and the type system should distinguish left- and right-linear polynomials. Every left (right) operand of multiplication should be either a constant or a right-linear (left-linear, respectively) polynomial. Addition of a left-linear polynomial and a right-linear polynomial should not be allowed.

First, we can refine the operational semantics of additions and multiplications as follows.

$$\begin{aligned}
& (c_0 \oplus (c_1 \otimes x_1) \oplus \cdots \oplus (c_m \otimes x_m)) \oplus (c'_0 \oplus (c'_1 \otimes x_1) \oplus \cdots \oplus (c'_m \otimes x_m)) \\
& \quad \rightarrow c''_0 \oplus (c''_1 \otimes x_1) \oplus \cdots \oplus (c''_m \otimes x_m) \quad \text{where } \mathcal{R} \ni c''_i = c_i \oplus c'_i \ (0 \leq i \leq m) \\
& (c_0 \oplus (x_1 \otimes c_1) \oplus \cdots \oplus (x_m \otimes c_m)) \oplus (c'_0 \oplus (x_1 \otimes c'_1) \oplus \cdots \oplus (x_m \otimes c'_m)) \\
& \quad \rightarrow c''_0 \oplus (x_1 \otimes c''_1) \oplus \cdots \oplus (x_m \otimes c''_m) \quad \text{where } \mathcal{R} \ni c''_i = c_i \oplus c'_i \ (0 \leq i \leq m) \\
& c_0 \otimes (c'_0 \oplus (c'_1 \otimes x_1) \oplus \cdots \oplus (c'_m \otimes x_m)) \\
& \quad \rightarrow c''_0 \oplus (c''_1 \otimes x_1) \oplus \cdots \oplus (c''_m \otimes x_m) \quad \text{where } \mathcal{R} \ni c''_i = c_0 \otimes c'_i \ (0 \leq i \leq m) \\
& (c_0 \oplus (x_1 \otimes c_1) \oplus \cdots \oplus (x_m \otimes c_m)) \otimes c'_0 \\
& \quad \rightarrow c''_0 \oplus (x_1 \otimes c''_1) \oplus \cdots \oplus (x_m \otimes c''_m) \quad \text{where } \mathcal{R} \ni c''_i = c_i \otimes c'_0 \ (0 \leq i \leq m)
\end{aligned}$$

Second, in the type system, $\mathcal{R}$ should be annotated by either LP (left-linear polynomial), RP (right-linear polynomial), or C (constant). We thus refine the typing rules for δ abstractions and multiplications as follows.

$$\begin{array}{c}
\frac{\Gamma\{x : \mathcal{R}_\alpha\} \vdash e : \tau' \quad \Gamma\{x : \mathcal{R}_\beta\} \vdash e : \tau'' \quad \beta \in \{\text{LP}, \text{RP}\}}{\Gamma \vdash \delta x^{\mathcal{R}}. e : \mathcal{R}_\alpha \rightarrow \tau'} \\
\\
\frac{\Gamma \vdash e_1 : \mathcal{R}_C \quad \Gamma \vdash e_2 : \mathcal{R}_\beta \quad \beta \in \{\text{LP}, \text{C}\}}{\Gamma \vdash e_1 \otimes e_2 : \mathcal{R}_\beta} \quad \frac{\Gamma \vdash e_1 : \mathcal{R}_\beta \quad \beta \in \{\text{RP}, \text{C}\} \quad \Gamma \vdash e_2 : \mathcal{R}_C}{\Gamma \vdash e_1 \otimes e_2 : \mathcal{R}_\beta}
\end{array}$$

Although these refinements make the whole calculus more complicated, they maintain the major properties, namely, Theorems 3.1, 3.3, and 3.4.

We can consider other algebraic structures as well. In general, λ^{AS} can use an algebraic structure for simplification if the following conditions hold:

- There is a simplification strategy that forms a confluent rewriting system.
- Effectiveness of simplification can be (possibly by a type system) guaranteed.

An interesting example includes data structures with holes [Minamide 1998]. Data structures with holes form a (non-commutative) monoid, which can be seen as a non-commutative semiring without additions. However, the linearity requirement of polynomials is not sufficient to guarantee efficiency in this case, because using a structure more than once may cause duplication of the structure, and it is hoped that the type system guarantee the single-useness of each structure. Such modifications on the type system maintains the major properties of λ^{AS} .

3.5.2 Other Programming Constructs. λ^{AS} is designed so that it can be extended with standard program constructs such as conditionals, data structures, and recursions. Note that Theorems 3.1, 3.3, and 3.4 do not depend on details of the calculus such as the evaluation order and termination. Accordingly, any construct can be added if it can be expressed by a lambda calculus (neglecting the evaluation strategy) and does not directly manipulate semiring values.

For instance, it is possible to add the fixed-point operator to λ^{AS} . Note that Theorems 3.1, 3.3, and 3.4 deal only with terminating evaluations. If an evaluation terminates in n steps, then we can use an n -fold unfolding operator instead of the fixed-point operators⁷. Adding the n -fold unfolding operator does not break the properties of λ^{AS} because it can be expressed in the simply typed lambda calculus. Therefore, $\delta x^{\mathcal{R}}. e$ is observationally equivalent to $\lambda x^{\mathcal{R}}. e$ if evaluations of these two terms terminate. Note, however, that they may have different termination behaviors. For instance, when $\perp$ is non-terminating, $(\lambda x^{\mathcal{R}}. 1) (\lambda z^{\mathcal{R}}. \perp)$ terminates, whereas $(\lambda x^{\mathcal{R}}. 1) (\delta z^{\mathcal{R}}. \perp)$ does not.

3.5.3 More Than One Semiring. Conceptually, it is not difficult to deal with programs that are specified using more than one semiring if those semirings are clearly distinguished. In practice, however, multiple semirings may share operators and values. For example, integers and integer additions are used in both $(\mathbb{Z}, +, \times, 0, 1)$ and $(\mathbb{Z} \cup \{-\infty\}, \uparrow, +, -\infty, 0)$. The type system should thus

⁷This approach is known as Levy's labeled reduction technique [Lévy 1976].

distinguish these two semirings and be aware of problematic terms like $\delta x. (-2) \times (x \uparrow 1)$, which consists of operators, $\uparrow$ and $\times$, that do not form a semiring.

It is possible but not satisfactory to develop a type system that rejects all terms in which a δ abstraction involves more than one different semiring. A better approach is to provide a method that enables restructuring of terms so that the body of a δ abstraction contains computation of at most one semiring. For instance, $(\delta x. (-2) \times (x \uparrow 1)) e$ is equivalent to $(\delta y. (-2) \times y) ((\delta x. x \uparrow 1) e)$, which is not problematic. Further investigation of this notion is left for future work.

3.6 Encoding by Hindley-Milner Typing

The type system of λ^{AS} is not satisfactory from a practical perspective. First, its typechecking cost is exponential. The rule for a multiplication requires examining two possibilities; moreover, the rule for a δ abstraction requires typechecking the body twice. Second, the types in λ^{AS} are monomorphic. For example, let $id = \lambda x. x$ in $\delta x. id \times id \ 1$ is not typeable (type annotations are omitted here because no annotation is appropriate). Because id takes an indeterminate x as an argument, the type of id should be $\mathbb{Z}_p \rightarrow \mathbb{Z}_p$; then, the type of $id \ 1$ is $\mathbb{Z}_p$, which violates the requirement of multiplication. Third, the type system is custom-made and thus not available in widely used programming languages.

A promising approach to solve these problems is to encode the type system by using the standard Hindley-Milner type system. If this approach is feasible, the type system would become polymorphic and available in practical programming languages; moreover, existing efficient implementations could be used for typechecking.

Recall that the body of a δ abstraction is essentially evaluated twice, and that the two evaluations may take arguments of different types. We can informally express this situation by the following equation. Here, $[x]$ denotes an indeterminate rather than a variable.

$$\delta x^{\mathcal{R}}. e \equiv \text{let } y = (\lambda x^{\mathcal{R}}. e) [x] \text{ in } \lambda x^{\mathcal{R}}. y$$

That is, $\delta x^{\mathcal{R}}. e$ is first takes $[x]$ as its argument, and after that, additionally takes the actual argument. Note that **let** polymorphism can express this situation, because it enables a function to take two arguments of possibly different types without preparing two instances of the function. Based on this observation, we can check the following expression instead of directly typechecking $\delta x^{\mathcal{R}}. e$.

$$\text{let } f = \lambda x^{\mathcal{R}}. e \text{ in let } _ = f [x] \text{ in } f$$

That is, $\delta x^{\mathcal{R}}. e$ is essentially regarded as $\lambda x^{\mathcal{R}}. e$, but in addition, its applicability to an indeterminate $[x] :: \mathcal{R}_p$ is checked⁸.

The polymorphism of the Hindley-Milner type system can encode subtyping [Fluet and Pucella 2006], which is useful for expressing other typing rules of λ^{AS} . For example, the types of constants and $\oplus$ can be expressed by $\forall \alpha. \mathcal{R}_\alpha$ and $\forall \alpha. \mathcal{R}_\alpha \rightarrow \mathcal{R}_\alpha \rightarrow \mathcal{R}_\alpha$, respectively.

Unfortunately, the rule for multiplications cannot be encoded by the Hindley-Milner type system. This problem is essential because multiplications do not have the most general type schema. For instance, the type of $\lambda x^{\mathcal{R}}. \lambda y^{\mathcal{R}}. x \otimes y$ is either $\forall \alpha. \mathcal{R}_\alpha \rightarrow \mathcal{R}_C \rightarrow \mathcal{R}_\alpha$ or $\forall \alpha. \mathcal{R}_C \rightarrow \mathcal{R}_\alpha \rightarrow \mathcal{R}_\alpha$, and these two types are incomparable. A natural workaround is to use two kinds of multiplications, $(\otimes_L) :: \forall \alpha. \mathcal{R}_C \rightarrow \mathcal{R}_\alpha \rightarrow \mathcal{R}_\alpha$ and $(\otimes_R) :: \forall \alpha. \mathcal{R}_\alpha \rightarrow \mathcal{R}_C \rightarrow \mathcal{R}_\alpha$. This modification may, however, make some typeable terms not typeable. For instance, although $\text{let } f = \lambda x^{\mathcal{R}}. \lambda y^{\mathcal{R}}. x \otimes y \text{ in } \delta x^{\mathcal{R}}. f \ 1 \ x \oplus f \ 1$ is typeable if $\otimes$ is commutative, neither $\otimes_L$ nor $\otimes_R$ can be used instead of $\otimes$.

We can extend this approach to deal with non-commutative semirings. When using a non-commutative semiring $\mathcal{R}$, every base type has two kinds of annotations: left-linear L or right-linear

⁸Here, we use the type notation for Haskell, $::$, to avoid potential confusion with the original typing of λ^{AS} .

R, and constant C or polynomial P. Accordingly, we can encode the types of semiring values, semiring operators, and indeterminates as follows.

$$\begin{aligned} c &:: \forall \alpha, \beta. \mathcal{R}_{\alpha, \beta} \\ (\oplus) &:: \forall \alpha, \beta. \mathcal{R}_{\alpha, \beta} \rightarrow \mathcal{R}_{\alpha, \beta} \rightarrow \mathcal{R}_{\alpha, \beta} \\ (\otimes_L) &:: \forall \alpha, \beta. \mathcal{R}_{\alpha, C} \rightarrow \mathcal{R}_{L, \beta} \rightarrow \mathcal{R}_{L, \beta} \\ (\otimes_R) &:: \forall \alpha, \beta. \mathcal{R}_{R, \beta} \rightarrow \mathcal{R}_{\alpha, C} \rightarrow \mathcal{R}_{R, \beta} \\ [x] &:: \forall \alpha. \mathcal{R}_{\alpha, P} \end{aligned}$$

Again, this encoding rejects some typeable terms. For instance, $(1 \otimes_L 1) \oplus (1 \otimes_R 1)$ cannot pass the typechecking based on this encoding.

In summary, we can make the type system of λ^{AS} efficient, polymorphic, and available in standard programming languages by paying the cost of annotating multiplications and rejecting some typeable terms. The cost seems acceptable relative to the benefit.

3.7 Limitation

To this point, we have discussed the following properties of λ^{AS} .

- $\delta x^{\mathcal{R}}. e$ and $\lambda x^{\mathcal{R}}. e$ are observationally equivalent if both terms are successfully reduced to values.
- For any typeable λ^{AS} term, the cost of algebraic simplification is proportional to the number of indeterminates.

These properties are useful for developing parallel reductions. Nevertheless, their usefulness is somewhat restrictive.

The first property is the key to developing parallel implementations by equational reasoning. As discussed in Section 3.5, however, it does not prevent erroneous behaviors (such as non-termination) unless the type system can detect them. This issue could be resolved by combining λ^{AS} with an existing method of analyzing safety.

The second property guarantees the absence of some apparently inefficient situations. Nevertheless, it does not guarantee that the parallel evaluation is faster than the corresponding sequential evaluation.

First, λ^{AS} does not guarantee load balancing, and therefore, parallel evaluation may simply be useless. For instance, the following program is correct but does not achieve any parallel speedup, because the δ abstracted subterm, $\delta y^{\mathbb{Z}}. a + y$, cannot be simplified any further.

$$\begin{aligned} \text{sum } [] &= 0 \\ \text{sum } (a : x) &= (\delta y^{\mathbb{Z}}. a + y) (\text{sum } x) \end{aligned}$$

Second, δ abstractions introduce overheads. For instance, the parallel implementation of *poly* calculates two coefficients of a linear polynomial, and therefore, gets about twice as much work as the sequential implementation. In general, if a calculated linear polynomial contains k indeterminates, then its simplification is about $k + 1$ times as slow as the usual evaluation⁹. This overhead is often essential for reduction parallelization, as in the case of *poly*. Moreover, this overhead is negligible in terms of asymptotic complexity if the number of indeterminates is at most a constant.

Note, however, the overhead of simplifying linear polynomials is not negligible in general, because a small program may be evaluated to a linear polynomial that contains many indeterminates. For example, consider the following reduction process.

⁹Note that the simplification and the usual evaluation differs only about the base types, namely, linear polynomials and constants. Therefore, multiple uses of the same variable that bound a linear polynomial do not make the situation worse. Such a multiple use duplicates a linear polynomial, instead of a constant, but the overhead of the duplication is still proportional to the size of the polynomial.

$$\begin{aligned}
& \text{let } g = \lambda f. \lambda x. \delta y. f(x+y) y \text{ in } g(g(g(\lambda v. \lambda w. v+w))) 1 \\
& \rightarrow (\lambda f. \lambda x. \delta y. f(x+y) y) ((\lambda f. \lambda x. \delta y. f(x+y) y) (\lambda x. \delta y. (\lambda v. \lambda w. v+w) (x+y) y)) 1 \\
& \rightarrow \delta y. (\lambda x. \delta y. (\lambda x. \delta y. (\lambda v. \lambda w. v+w) (x+y) y) (x+y) y) (1+y) y \\
& \equiv \delta y_0. (\lambda x. \delta y_1. (\lambda x. \delta y_2. (\lambda v. \lambda w. v+w) (x+y_2) y_2) (x+y_1) y_1) (1+y_0) y_0 \\
& \rightarrow \delta y_0. (\delta y_1. (\delta y_2. \underline{1+y_0+y_1+2 \times y_2}) y_1) y_0 \\
& \rightarrow \lambda y_0. 1+4 \times y_0
\end{aligned}$$

The underlined linear polynomial consists of three indeterminates, y_0 , y_1 , and y_2 , all of which originates from the same δ abstraction in the function g . We can obtain more indeterminates by composing more g s in the same manner: $g(g(\dots(g(g(\lambda v. \lambda w. v+w)))\dots)) 1$. The type system of λ^{AS} does not reject such programs that use many indeterminates, and therefore, it gives only a limited guarantee of efficiency.

4 RELATED WORK

The characteristic feature of λ^{AS} is its use of algebraic properties for simplifying functions. This feature is closely related to partial evaluation [Jones 1996]. Given a subset of inputs, which are called static, partial evaluation generates a program specialized for the static inputs without knowing the other inputs, called dynamic. The function simplification in λ^{AS} is closely related to partial evaluation. On one hand, offline partial evaluation, in which usual evaluation may invoke partial evaluation at runtime, can implement the function simplification in λ^{AS} . When the evaluator encounters $\delta x. e$, it regards x as a dynamic input and requests a partial evaluator to simplify e . On the other hand, δ abstraction can be used for expressing (semiring-based) offline partial evaluation. For example, given a function $f(s, d)$, where s and d are respectively the static and dynamic input, its partial evaluation with fixing the static input s to 1 can be expressed by $(\lambda s. \delta d. f(s, d)) 1$. From this perspective, one of the most closely related studies is the parallel partial evaluation by Consel and Danvy [1992]. While λ^{AS} makes use of algebraic properties and requires linearity over indeterminates to model efficient parallel reductions, however, their approach does not impose any requirements on programs and therefore provides no support for parallel programming, especially for developing parallel reductions.

Several studies have shown the usefulness of partial evaluation or function simplification for developing parallel reductions, including those on deriving parallel reductions on arrays/lists and trees [Callahan 1992; Chin et al. 1998; Farzan and Nicolet 2017; Fisher and Ghuloum 1994; Hu et al. 1998; Jiang et al. 2018; Matsuzaki et al. 2005; Morihata and Matsuzaki 2010; Raychev et al. 2015] and those on parallel querying of semi-structured databases [Buneman et al. 2006; Cong et al. 2007, 2012]. λ^{AS} extends those studies and builds a foundation for studying parallel reductions in general-purpose higher-order languages.

A δ abstraction can be read as an annotation to express speculative evaluation. From this viewpoint, λ^{AS} is similar to the evaluation strategy approach for parallel computations [Marlow et al. 2010], in which parallelism is specified and controlled by evaluation strategies. There is a crucial difference, however: in the evaluation strategy approach, programmers can control evaluation strategies only when the language does not specify the order of evaluation. In contrast, a δ abstraction in λ^{AS} requires subterm simplification that the standard evaluation strategy does not allow.

Castro et al. [2016, 2018] proposed a type-based approach for introducing parallelism to purely functional programs. Their type system certifies not only the correctness but also the cost of an obtained parallel program. Their approach, using a type system to support parallelization of functional programs, is somewhat similar to the current proposal, but there are two essential differences. First, they focused on structured functional programs, especially those specified by

algorithmic skeletons [Cole 1989]. Algorithmic skeletons are reusable parallel programming patterns such as *map*, reductions, and prefix sum patterns. The focus on structured programs enables their method to analyze programs in detail. In contrast, the current proposal seeks to provide a foundation that can deal with complex unstructured programs. Second, while their method mainly deal with programs that apparently contain independent subexpressions, the current proposal also considers programs whose divide-and-conquer implementations require breaking data dependencies by using algebraic properties.

Nishimura and Ohori [1999] proposed a higher-order functional programming language that has a special construct, called a parallel map, for modeling parallel reductions. Similar to λ^{AS} , the parallel map is based on substitutions for indeterminates. λ^{AS} refines their proposal in the following aspects. First, their language does not explicitly account for algebraic simplification; therefore, it is unclear when the parallel map implements efficient parallel reduction. In contrast, λ^{AS} explicitly deals with simplifications and provides a type system that guarantees successful simplification. Moreover, the parallel map is based on communications guided by the pointer structure of recursive data. Consequently, it uses a “pointer jumping” strategy, which is less efficient than the standard divide-and-conquer approach. In contrast, λ^{AS} does not rely on pointer-based structures and can express the divide-and-conquer strategy.

λ^{AS} focuses on linear polynomials on semiring operators. The importance of linear polynomials in the context of parallel reductions has already been discussed. Xu et al. [2004] developed an automatic parallelization system for list reductions. Their idea is to trace algebraic operators and the linearity condition by using a type system. Matsuzaki et al. [2006] and Sato and Iwasaki [2011] developed similar systems for automatic parallelization of tree reductions and reduction loops, respectively. λ^{AS} is strongly influenced by those works and provides a primitive construct, the δ abstraction, that enables us to study those parallelization strategies. The basic idea of λ^{AS} is that their essence, i.e., the use of algebraic properties for modeling complex reductions, is independent of the control structures that express iterations/recursions.

The operational semantics of λ^{AS} interleaves the usual sort of evaluations of lambda calculi with simplifications based on algebraic properties. These simplifications can be regarded as a kind of semantic evaluations as they are based on the mathematical properties of the operators. Accelerating evaluations of lambda calculi through semantic evaluations is not a new idea. Terui [2012] showed that semantic evaluations enable efficient sequential evaluations of lambda expressions, thereby leading to a precise bound on computational costs. Kobayashi et al. [2012] used type-based semantic evaluations to perform computations on compressed data without decompression.

5 CONCLUSION AND FUTURE WORK

This paper has developed λ^{AS} , a simply-typed lambda calculus with algebraic simplifications. The key characteristic of λ^{AS} is the δ abstraction whose function body is simplified using algebraic properties before its arguments’ arrival. The operational semantics and type system of λ^{AS} were formalized. The type system guarantees that the simplification results in linear polynomials and, in turn, rules out the major possibility of unsuccessful parallelization. The usefulness of λ^{AS} for modeling parallel reductions was demonstrated on several nontrivial examples.

This is the first step in providing a foundation for parallel reductions based on lambda calculi. There are many directions for further investigation.

Inferring Evaluation Costs. As discussed in Section 3.7, the type system of λ^{AS} does not guarantee that parallel implementations are faster than sequential implementations. A precise cost inference for λ^{AS} is more challenging than those for usual lambda calculi, because the cost depends on the number of indeterminates used during simplifications, and moreover, a δ abstraction may generate

more than one, possibly unboundedly many, indeterminates. It is natural to seek for a practical subset of λ^{AS} in which the number of necessary indeterminates is known. Indeed, every example discussed in Section 2 requires a constant number of indeterminates. Such a subset might be obtained by considering structural recursions, as in the study of Castro et al. [2016, 2018], and restrict duplication of δ -abstracted functions. If such a subset is found, it might be worthwhile to consider nonlinear polynomials as well, because exponential blowups cannot occur.

Strategy for Introducing Parallelism. It is hoped to have a good strategy of introducing δ abstractions. This issue is closely related to cost inference. If evaluation costs can be precisely inferred, even the following naive strategy would be useful: replace a λ abstraction to a δ abstraction if the computation of the corresponding argument takes time and the introduction of δ abstraction does not make the cost significantly worse. This strategy is, however, not sufficient to deal with recursive functions that process large data, such as *foldr* and *foldl*. As discussed in Section 2, to obtain efficient parallel reductions for large data processing, we should combine δ abstractions with the divide-and-conquer approach.

Compilation to Existing Calculus. Although λ^{AS} is a theoretical model for studying reduction parallelization, it would provide better understanding of λ^{AS} to formulate a compilation to an existing calculus (or an abstract machine) that supports parallel evaluation. As discussed in Section 3.7, evaluation of λ^{AS} may lead to unboundedly many free variables (indeterminates). This situation is somewhat similar to the case of lazy evaluations, in which a heap in addition to an environment is necessary, because an evaluation may lead to unboundedly many thunks [Launchbury 1993]. However, a naive compilation to a heap-based calculus would thread computations and thereby prohibit exploiting parallelism.

Parallelization of Practical Programs. The original motivation in developing λ^{AS} is its application for reduction parallelization of programs written in practical programming languages. Although λ^{AS} is extensible, it is unclear whether it can incorporate practical, complex programming constructs and be applied to reason on practical complex programs.

ACKNOWLEDGMENTS

The author is grateful to Makoto Hamana for his suggestion of related articles that were useful for proving Theorem 3.1, and Shigeyuki Sato for insightful discussion with him. The author is also grateful to the reviewers for their valuable comments helping to improve this paper. The author is supported by JSPS Grant-in-Aid for Young Scientists (B), 15K15965.

REFERENCES

- Guy E. Blelloch. 1993. Prefix Sums and Their Applications. In *Synthesis of Parallel Algorithms*, John H. Reif (Ed.). Morgan Kaufmann Publishers, Chapter 1.
- Peter Buneman, Gao Cong, Wenfei Fan, and Anastasios Kementsietsidis. 2006. Using Partial Evaluation in Distributed Query Evaluation. In *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006*. ACM, 211–222.
- David Callahan. 1992. Recognizing and Parallelizing Bounded Recurrences. In *Languages and Compilers for Parallel Computing, Fourth International Workshop, Santa Clara, California, USA, August 7-9, 1991, Proceedings (Lecture Notes in Computer Science)*, Vol. 589. Springer, 169–185.
- David Castro, Kevin Hammond, and Susmit Sarkar. 2016. Farms, pipes, streams and reforestation: reasoning about structured parallel processes using types and hylomorphisms. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*. ACM, 4–17.
- David Castro, Kevin Hammond, Susmit Sarkar, and Yasir Alguwaifi. 2018. Automatically deriving cost models for structured parallel processes using hylomorphisms. *Future Generation Comp. Syst.* 79 (2018), 653–668.

- Yun-Yan Chi and Shin-Cheng Mu. 2011. Constructing List Homomorphisms from Proofs. In *Programming Languages and Systems - 9th Asian Symposium, APLAS 2011, Kenting, Taiwan, December 5-7, 2011. Proceedings (Lecture Notes in Computer Science)*, Vol. 7078. Springer, 74–88.
- Wei-Ngan Chin, Akihiko Takano, and Zhenjiang Hu. 1998. Parallelization via Context Preservation. In *Proceedings of the 1998 International Conference on Computer Languages, ICCL '98, May 14-16, 1998, Chicago, IL, USA*. IEEE Computer Society, 153–162.
- Murray I. Cole. 1989. *Algorithmic Skeletons: Structural Management of Parallel Computation*. MIT Press.
- Gao Cong, Wenfei Fan, and Anastasios Kementsietsidis. 2007. Distributed query evaluation with performance guarantees. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007*. ACM, 509–520.
- Gao Cong, Wenfei Fan, Anastasios Kementsietsidis, Jianzhong Li, and Xianmin Liu. 2012. Partial Evaluation for Distributed XPath Query Processing and Beyond. *ACM Trans. Database Syst.* 37, 4 (2012), 32:1–32:43.
- Charles Consel and Olivier Danvy. 1992. Partial Evaluation in Parallel. *Lisp and Symbolic Computation* 5, 4 (1992), 327–342.
- Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *6th Symposium on Operating System Design and Implementation (OSDI 2004), December 6-8, 2004, San Francisco, California, USA*. 137–150.
- Steven J. Deitz, David Callahan, Bradford L. Chamberlain, and Lawrence Snyder. 2006. Global-view abstractions for user-defined reductions and scans. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2006, New York, New York, USA, March 29-31, 2006*. ACM, 40–47.
- Kento Emoto, Sebastian Fischer, and Zhenjiang Hu. 2012. Filter-embedding semiring fusion for programming with MapReduce. *Formal Asp. Comput.* 24, 4-6 (2012), 623–645.
- Kento Emoto, Zhenjiang Hu, Kazuhiko Kakehi, Kiminori Matsuzaki, and Masato Takeichi. 2010. Generators-of-Generators Library with Optimization Capabilities in Fortress. In *Euro-Par 2010 - Parallel Processing, 16th International Euro-Par Conference, Ischia, Italy, August 31 - September 3, 2010, Proceedings, Part II (Lecture Notes in Computer Science)*, Vol. 6272. Springer, 26–37.
- Azadeh Farzan and Victor Nicolet. 2017. Synthesis of divide and conquer parallelism for loops. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*. ACM, 540–555.
- Grigory Fedyukovich, Maaz Bin Safeer Ahmad, and Rastislav Bodík. 2017. Gradual synthesis for static parallelization of single-pass array-processing programs. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*. ACM, 572–585.
- Allan L. Fisher and Anwar M. Ghuloum. 1994. Parallelizing Complex Scans and Reductions. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI), Orlando, Florida, June 20-24, 1994*. ACM, 135–146.
- Matthew Fluet and Riccardo Pucella. 2006. Phantom types and subtyping. *J. Funct. Program.* 16, 6 (2006), 751–791.
- Matthew Fluet, Mike Rainey, John H. Reppy, and Adam Shaw. 2008. Implicitly threaded parallelism in Manticore. *J. Funct. Program.* 20, 5-6 (2008), 537–576.
- Sergei Gorlatch. 1999. Extracting and Implementing List Homomorphisms in Parallel Program Development. *Science of Computer Programming* 33, 1 (1999), 1–27.
- Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. 2017. Futhark: purely functional GPU-programming with nested parallelism and in-place array updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*. ACM, 556–571.
- Zhenjiang Hu, Hideya Iwasaki, and Masato Takeichi. 1997. Formal derivation of efficient parallel programs by construction of list homomorphisms. *ACM Transactions on Programming Languages and Systems* 19, 3 (1997), 444–461.
- Zhenjiang Hu, Masato Takeichi, and Wei-Ngan Chin. 1998. Parallelization in Calculational Forms. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, January 19-21, 1998, San Diego, CA, USA*. ACM, 316–328.
- Peng Jiang, Linchuan Chen, and Gagan Agrawal. 2018. Revealing parallel scans and reductions in recurrences through function reconstruction. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques, PACT 2018, Limassol, Cyprus, November 01-04, 2018*. ACM, 10:1–10:13.
- Neil D. Jones. 1996. An Introduction to Partial Evaluation. *Comput. Surveys* 28, 3 (1996), 480–503.
- Gabriele Keller, Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon L. Peyton Jones, and Ben Lippmeier. 2010. Regular, shape-polymorphic, parallel arrays in Haskell. In *Proceeding of the 15th ACM SIGPLAN International Conference on Functional Programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*. ACM, 261–272.
- Naoki Kobayashi, Kazutaka Matsuda, Ayumi Shinohara, and Kazuya Yaguchi. 2012. Functional programs as compressed data. *Higher-Order and Symbolic Computation* 25, 1 (2012), 39–84.
- Richard E. Ladner and Michael J. Fischer. 1980. Parallel Prefix Computation. *J. ACM* 27, 4 (1980), 831–838.

- John Launchbury. 1993. A Natural Semantics for Lazy Evaluation. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina, USA*. ACM, 144–154.
- Jean-Jacques Lévy. 1976. An Algebraic Interpretation of the $\lambda\beta K$ -Calculus; and an Application of a Labelled λ -Calculus. *Theor. Comput. Sci.* 2, 1 (1976), 97–114.
- Simon Marlow, Patrick Maier, Hans-Wolfgang Loidl, Mustafa Aswad, and Philip W. Trinder. 2010. Seq no more: better strategies for parallel Haskell. In *Proceedings of the 3rd ACM SIGPLAN Symposium on Haskell, Haskell 2010, Baltimore, MD, USA, 30 September 2010*. ACM, 91–102.
- Kiminori Matsuzaki, Zhenjiang Hu, Kazuhiko Kakehi, and Masato Takeichi. 2005. Systematic Derivation of Tree Contraction Algorithms. *Parallel Processing Letters* 15, 3 (2005), 321–336.
- Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi. 2006. Towards automatic parallelization of tree reductions in dynamic programming. In *SPAA 2006: Proceedings of the 18th Annual ACM Symposium on Parallel Algorithms and Architectures, Cambridge, Massachusetts, USA, July 30 - August 2, 2006*. ACM, 39–48.
- Yasuhiko Minamide. 1998. A Functional Representation of Data Structures with a Hole. In *POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA, January 19-21, 1998*. ACM, 75–84.
- Akimasa Morihata and Kiminori Matsuzaki. 2010. Automatic Parallelization of Recursive Functions Using Quantifier Elimination. In *Functional and Logic Programming, 10th International Symposium, FLOPS 2010, Sendai, Japan, April 19-21, 2010. Proceedings (Lecture Notes in Computer Science)*, Vol. 6009. Springer, 321–336.
- Akimasa Morihata and Kiminori Matsuzaki. 2011. Balanced trees inhabiting functional parallel programming. In *Proceeding of the 16th ACM SIGPLAN International Conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*. ACM, 117–128.
- Akimasa Morihata, Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi. 2009. The Third Homomorphism Theorem on Trees: Downward & Upward Lead to Divide-and-Conquer. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, Georgia, USA, January 21-23, 2009*. ACM, 177–185.
- Kazutaka Morita, Akimasa Morihata, Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi. 2007. Automatic inversion generates divide-and-conquer parallel programs. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*. ACM, 146–155.
- Susumu Nishimura and Atsushi Ohori. 1999. Parallel Functional Programming on Recursively Defined Data via Data-Parallel Recursion. *J. Funct. Program.* 9, 4 (1999), 427–462.
- Veselin Raychev, Madanlal Musuvathi, and Todd Mytkowicz. 2015. Parallelizing user-defined aggregations using symbolic execution. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*. ACM, 153–167.
- Shigeyuki Sato and Hideya Iwasaki. 2011. Automatic parallelization via matrix multiplication. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*. ACM, 470–479.
- Toshio Suganuma, Hideaki Komatsu, and Toshio Nakatani. 1996. Detection and Global Optimization of Reduction Operations for Distributed Parallel Machines. In *ICS '96: Proceedings of the 1996 International Conference on Supercomputing, May 25-28, 1996, Philadelphia, PA, USA*. ACM, 18–25.
- Val Tannen. 1988. Combining Algebra and Higher-Order Types. In *Proceedings of the Third Annual Symposium on Logic in Computer Science (LICS '88), Edinburgh, Scotland, UK, July 5-8, 1988*. IEEE Computer Society, 82–90.
- Val Tannen and Jean H. Gallier. 1991. Polymorphic Rewriting Conserves Algebraic Strong Normalization. *Theor. Comput. Sci.* 83, 1 (1991), 3–28.
- Kazushige Terui. 2012. Semantic Evaluation, Intersection Types and Complexity of Simply Typed Lambda Calculus. In *23rd International Conference on Rewriting Techniques and Applications (RTA'12), RTA 2012, May 28 - June 2, 2012, Nagoya, Japan (LIPIcs)*, Vol. 15. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 323–338.
- Dana N. Xu, Siau-Cheng Khoo, and Zhenjiang Hu. 2004. PType System: A Featherweight Parallelizability Detector. In *Programming Languages and Systems: Second Asian Symposium, APLAS 2004, Taipei, Taiwan, November 4-6, 2004. Proceedings (Lecture Notes in Computer Science)*, Vol. 3302. Springer, 197–212.