JFP **31**, e7, 40 pages, 2021. © The Author(s), 2021. Published by Cambridge University Press. This is an Open Access article, distributed under the terms of the Creative Commons Attribution licence (http://creativecommons.org/licenses/by/4.0/), which permits unrestricted re-use, distribution, and reproduction in any medium, provided the original work is properly cited.

doi:10.1017/S0956796821000058

Lambda calculus with algebraic simplification for reduction parallelisation: Extended study

AKIMASA MORIHATA

University of Tokyo, 3-8-1, Komaba, Meguro-ku, Tokyo, Japan
(*e-mail:* morihata@graco.c.u-tokyo.ac.jp)

Abstract

Parallel reduction is a major component of parallel programming and widely used for summarisation and aggregation. It is not well understood, however, what sorts of non-trivial summarisations 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 a parallelisation 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 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 conditional jumps, prefix sum patterns and even tree manipulations.

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:

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

it is syntactically clear that the two recursive calls, $\mathit{fib}\ (n-1)$ and $\mathit{fib}\ (n-2)$, 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:

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

https://doi.org/10.1017/S0956796821000058 Published online by Cambridge University Press

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

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

Such equational reasoning is useful for not only certification but also for the *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:

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

It is not difficult to generalise 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 summarisation 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 MPI,[1] OpenMP,[2] Intel Threading Building Blocks,[3] MapReduce (Dean & Ghemawat, 2004), Cilk++ (Frigo *et al.*, 2009), Manticore (Fluet *et al.*, 2008), Repa (REgular PArallel arrays) for Haskell (Keller *et al.*, 2010) and Futhark (Henriksen *et al.*, 2017).

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, typically loops (or singly recursive function) specified by using an associative operator. To see the problem, consider the following *poly* function, which calculates the value of a polynomial represented by a list of coefficients. Its formal definition is shown in Figure 1(a):

$$poly\ x\ [a_0, a_1, \ldots, a_n] = a_0 + a_1 x + \cdots + a_n x^n$$

[1] http://mpi-forum.org/.
[2] http://openmp.org/wp/.
[3] https://www.threadingbuildingblocks.org/.

https://doi.org/10.1017/S0956796821000058 Published online by Cambridge University Press

$$\begin{aligned} poly\ x\ [\,] &= 0 \\ poly\ x\ (a:y) &= a + x \times poly\ x\ y \end{aligned}$$

(a) Polynomial computation

$$\begin{aligned} sumB1\ e\ [\,] &= e \\ sumB1\ e\ (a:x) &= \textbf{if}\ a < 0\ \textbf{then}\ e \\ &\qquad\qquad \textbf{else}\ sumB1\ (e+a)\ x \end{aligned}$$

(b) Summation with `break`

$$\begin{aligned} psum\ [\,]\ e &= [\,] \\ psum\ (a:x)\ e &= e : (psum\ x\ (e+a)) \end{aligned}$$

(c) Prefix sum

$$\begin{aligned} rd\ (\text{Assign}\ v\ e)\ y &= (remove\ v\ y) \cup \{(v, e)\} \\ rd\ (\text{Seq}\ s_1\ s_2)\ y &= rd\ s_2\ (rd\ s_1\ y) \\ rd\ (\text{If}\ e\ s_1\ s_2)\ y &= rd\ s_1\ y \cup rd\ s_2\ y \\ rd\ (\text{While}\ e\ s)\ y &= y \cup rd\ s\ y \end{aligned}$$

(d) Reaching definition analysis

Fig. 1. Examples of non-trivial reductions. (a) Polynomial computation. (b) Summation with `break`. (c) Prefix sum. (d) Reaching definition analysis.

Although *poly* is a modest generalisation of *sum* (note that $poly\ 1 = sum$), 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 $poly\ x\ (l +\!\!+ r) = poly\ x\ l \oplus poly\ x\ r$. Therefore, its parallel implementation is non-trivial. A known parallel implementation uses the powers of x in addition to the value of *poly*. More formally, the parallel implementation is specified by the following $pl\ x\ y = (poly\ x\ y, x^{length\ y})$:

$$\begin{aligned} pl\ x\ (l +\!\!+ r) = \textbf{let}\ &(lp, lx) = pl\ x\ l \\ &(rp, rx) = pl\ x\ r \\ \textbf{in}\ &(lp + lx \times rp, lx \times rx) \end{aligned}$$

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

We hope for parallel programming environments to support a wide variety of non-trivial reductions that real programs 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 non-linear structures such as trees (Figure 1(d)). Although there have been many studies on systematically developing parallel reductions (Fisher & Ghuloum, 1994; Suganuma *et al.*, 1996; Hu *et al.*, 1997, 1998; Chin *et al.*, 1998; Gorlatch, 1999; Xu *et al.*, 2004; Matsuzaki *et al.*, 2005, 2006; Deitz *et al.*, 2006; Morita *et al.*, 2007; Morihata *et al.*, 2009; Morihata & Matsuzaki, 2010; Emoto *et al.*, 2010; Morihata & Matsuzaki, 2011; Sato & Iwasaki, 2011; Chi & Mu, 2011; Emoto *et al.*, 2012; Raychev *et al.*, 2015; Fedyukovich *et al.*, 2017; Farzan & Nicolet, 2017; Jiang *et al.*, 2018; Farzan & Nicolet, 2019), 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 using algebraic properties such

https://doi.org/10.1017/S0956796821000058 Published online by Cambridge University Press

as associativity and commutativity. For example, consider calculating *sum* $[a_0, \ldots, a_n]$. The sequential evaluation essentially corresponds to the following expression:

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

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 $(+)$:

$$\begin{array}{ll} & a_0 + (\cdots (a_{n-1} + (a_n + 0)) \cdots) \\ = & \{ \text{ inserting a lambda abstraction } \} \\ & (\lambda x.\ a_0 + (\cdots (a_{k-1} + (a_k + x)) \cdots))\ (a_{k+1} + (\cdots (a_{n-1} + (a_n + 0))) \cdots) \\ \rightrightarrows & \{ \text{ parallel evaluation } \} \\ & (\lambda x.\ a_0^k + x)\ a_{k+1}^n \qquad \text{where } a_0^k = \textstyle\sum_{0 \le i \le k} a_i \text{ and } a_{k+1}^n = \sum_{k+1 \le i \le n} a_i \end{array}$$

This understanding of parallel reduction is not new. It has been used for developing parallel reduction loops (Callahan, 1992; Fisher & Ghuloum, 1994; Sato & Iwasaki, 2011; Raychev *et al.*, 2015; Farzan & Nicolet, 2017; Jiang *et al.*, 2018), parallel list/tree reductions (Hu *et al.*, 1998; Chin *et al.*, 1998; Xu *et al.*, 2004; Matsuzaki *et al.*, 2005, 2006; Morihata & Matsuzaki, 2010) and parallel querying on semi-structured 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-abstracted term, $\lambda x.\ e$, is a value; in other words, the body e is not evaluated until its argument is passed. A δ-abstracted 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

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

It is non-trivial 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 $\delta x.\ x^3 + 3 \times x^2 + 3 \times x + 1$

https://doi.org/10.1017/S0956796821000058 Published online by Cambridge University Press

can be simplified to $\lambda x.\ (x+1)^3$ by factorisation. Even worse, an inappropriate simplification strategy may significantly decrease efficiency. For instance,

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

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

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

To provide a simple and effective simplification strategy, λ^{AS} requires that simplifications must result in *linear* polynomials.[4] 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 (Xu *et al.*, 2004; Matsuzaki *et al.*, 2006; Emoto *et al.*, 2010; Sato & Iwasaki, 2011; Emoto *et al.*, 2012) 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.

Formalising a new lambda calculus, λ^{AS} , should be an important step in developing a powerful reduction parallelisation method for practical programming languages. The existing studies on parallel reduction suggest the following hypothesis: parallel reductions rely on the algebraic properties and simplifications of the operators used, and are nearly independent of control structures or programming patterns. If this hypothesis is correct, 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 operators (for base-type values) are clearly distinguished from higher-order features.

This paper contains the following three major contributions:

- Systematic development of a wide variety of parallel reductions using λ^{AS} (Section 2): the paper discusses reduction patterns including all examples in Figure 1, and others, as well.
- Design of λ^{AS}, a lambda calculus with algebraic simplification (Section 3): the type system of λ^{AS} guarantees progress, that is, 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.
- Extensive studies for strengthening λ^{AS} (Section 4): in particular, the paper discusses the possibilities of combining λ^{AS} with the fixed-point operator, algebraic structures other than a commutative semiring and control operators.

This paper extends the preliminary report (Morihata, 2019). In particular, Sections 2.4, 4.2, and 4.5, Theorem 1, Corollary 6, and Example 9 are new.

[4] This paper specifically uses the term 'linear' to refer to the linearity of polynomials, and not to the 'single-use' of variables.

https://doi.org/10.1017/S0956796821000058 Published online by Cambridge University Press

$$\begin{array}{lll} \mathit{foldr}\ f\ e\ [\,] & = e & \mathit{foldl}\ f\ e\ [\,] & = e \\ \mathit{foldr}\ f\ e\ (a : x) & = f\ a\ (\mathit{foldr}\ f\ x) & \mathit{foldl}\ f\ e\ (a : x) & = \mathit{foldl}\ (f\ e\ a)\ x \end{array}$$

Fig. 2. Definitions of standard functions.

2 Developing complex parallel reductions by λ^{AS}

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

2.1 Flavour 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.

$$e ::= x \mid \lambda x^{\tau}.\, e \mid e\ e \mid c \mid e \oplus e \mid e \otimes e \mid \delta x^{\mathscr{R}}.\, e \mid \cdots$$

λ^{AS} extends the simply typed lambda calculus via the semiring operators, $\oplus$ and $\otimes$, on the carrier set $\mathscr{R}$ ($c \in \mathscr{R}$), and a δ abstraction, $\delta x^{\mathscr{R}}.\, e$. 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 $\mathscr{R}$. 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 (commutative) semiring of addition and multiplication on integers, that is, $(\oplus) = (+)$, $(\otimes) = \times$, and $\mathscr{R} = \mathbb{Z}$. 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, that is, $\delta x^{\mathbb{Z}}.\, e \equiv \lambda x^{\mathbb{Z}}.\, e$. 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, $\mathscr{R}$. 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 $\rightrightarrows$ is used instead to emphasise possibilities for parallel evaluations:

$$\begin{array}{ll} & (\delta x.\, x + 2 \times x)\ (2 + 5) \\ \rightrightarrows & (\lambda x.\, 3 \times x)\ 7 \\ \rightarrow & 3 \times 7 \\ \rightarrow & 21 \end{array}$$

https://doi.org/10.1017/S0956796821000058 Published online by Cambridge University Press

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 using the algebraic properties of the semiring.

So long as a δ abstraction is not involved, the evaluation is carried out as the usual call-by-value reduction. 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 providing a simple and effective simplification strategy for polynomials, 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 non-linear: it involves a multiplication of indeterminates, x and y.

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:

$$\textbf{let}\ x_1 = e_1 \ [\!]\ x_2 = e_2 \ [\!]\ \cdots \ [\!]\ x_k = e_k\ \textbf{in}\ e_{k+1} \equiv (\lambda x_1.\ \lambda x_2. \cdots \lambda x_k.\ e_{k+1})\ e_1\ e_2 \cdots e_k$$

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

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

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

https://doi.org/10.1017/S0956796821000058 Published online by Cambridge University Press

As $x + x \times (7 + 3)$ depends on x, its parallel evaluation appears to be impossible. In λ^{AS}, by replacing the first **let** with **plet**, defined by:

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

a parallel evaluation becomes possible:

$$\begin{array}{ll} & \textbf{plet}\ x = 3 + 5\ \textbf{in}\ x + x \times (7 + 3) \\ = & (\delta x.\ x + x \times (7 + 3))\ (3 + 5) \\ \rightrightarrows & (\lambda x.\ x \times 11)\ 8 \\ \to & 88 \end{array}$$

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:

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

This program cannot gain any parallel speedup even 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:

$$\begin{array}{ll} & \textbf{let}\ x_0 = a_0\ \textbf{in let}\ x_1 = a_1 + x_0\ \textbf{in let} \cdots \textbf{in let}\ x_n = \ a_n + x_{n-1}\ \textbf{in}\ x_n \\ = & \textbf{plet}\ z = (\textbf{let}\ x_0 = a_0\ \textbf{in let}\ x_1 = a_1 + x_0\ \textbf{in let} \cdots \textbf{in let}\ x_k = \ a_k + x_{k-1}\ \textbf{in}\ x_k) \\ & \textbf{in}\ (\textbf{let}\ x_{k+1} = a_{k+1} + z\ \textbf{in let}\ \cdots \textbf{in let}\ x_n = \ a_n + x_{n-1}\ \textbf{in}\ x_n) \\ \equiv & (\delta z.\ (\lambda x_{k+1}.\ (\lambda x_{k+2}.\ (\cdots (\lambda x_n.\ x_n) \cdots))\ (a_{k+2} + x_{k+1}))\ (a_{k+1} + z)) \\ & ((\lambda x_0.\ (\lambda x_1.\ (\cdots (\lambda x_k.\ x_k) \cdots))\ (a_1 + x_0))\ a_0) \\ \rightrightarrows & (\lambda x.\ a^n_{k+1} + x)\ a^k_0 \qquad \text{where } a^k_0 = \textstyle\sum_{0 \le i \le k} a_i \text{ and } a^n_{k+1} = \sum_{k+1 \le i \le n} a_i \end{array}$$

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 $sum\ (l +\!\!+ r)$ by processing l and r independently. This can be achieved by inserting a δ abstraction:

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

https://doi.org/10.1017/S0956796821000058 Published online by Cambridge University Press

Since the left-hand side expression contains $l +\!\!+ r$, which is not a pattern, the derived equation is not a valid function definition in Haskell. Yet, this equational reasoning suggests an implementation that processes l and r in parallel and guarantees the correctness of the implementation regardless of the strategy of splitting the given list into two sublists, l and r. Moreover, the function part, $\delta x.\ \mathit{foldr}\ (+)\ x\ l$, 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 generalised 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:

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

The function part can be effectively simplified because its body forms a linear expression. For example, *poly* 10 ([2, 1, 3] +\!\!+ [5, 9]) is evaluated as follows:

$$\begin{array}{ll} & \mathit{poly}\ 10\ ([2,1,3] +\!\!+ [5,9]) \\ \rightarrow & (\delta z.\ \mathit{foldr}\ (\lambda a.\ \lambda y.\ a + 10 \times y)\ z\ [2,1,3])\ (\mathit{poly}\ 10\ [5,9]) \\ \rightrightarrows & (\delta z.\ (2 + 10 \times (1 + 10 \times (3 + 10 \times z))))\ (5 + 10 \times (9 + 10 \times 0)) \\ \rightrightarrows & (\lambda z.\ 312 + 1{,}000 \times z)\ 95 \\ \rightarrow & 95{,}312 \end{array}$$

Recall that the parallel implementation of *poly x* discussed in the introduction additionally uses the powers of x. 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 powers. Therefore, λ^{AS} enables us to discover the parallel implementation of *poly* without using the expert knowledge.

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:

$$\begin{array}{ll} & \mathit{poly}\ x\ (y_0 +\!\!+ y_1 +\!\!+ \cdots +\!\!+ y_n) \\ = & ((\delta z.\ \mathit{poly}'_x\ z\ y_0) \circ (\delta z.\ \mathit{poly}'_x\ z\ y_1) \circ \cdots \circ (\delta z.\ \mathit{poly}'_x\ z\ y_n))\ (\mathit{poly}\ x\ [\,]) \\ & \quad \textbf{where}\ \mathit{poly}'_x\ e\ w = \mathit{foldr}\ (\lambda a.\ \lambda y.\ a + x \times y)\ e\ w \end{array}$$

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:

$$\begin{array}{rl} \mathit{mps}\ [\,] & = 0 \\ \mathit{mps}\ (a : x) & = 0 \uparrow (a + \mathit{mps}\ x) \end{array}$$

https://doi.org/10.1017/S0956796821000058 Published online by Cambridge University Press

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*:

$$mps\ (l +\!\!+ r) = (\delta z.\ foldr\ (\lambda a.\ \lambda y.\ 0 \uparrow (a + y))\ z\ l)\ (mps\ r)$$

2.4 Conditional

Conditional branches often interfere with parallelisations. λ^{AS} provides a guideline for parallelising programs with conditionals.

As an example, consider the following *sumP*, which calculates the summation of all positive elements:

$$\begin{aligned} sumP\ [\,] &= 0 \\ sumP\ (a : x) &= \textbf{if}\ a > 0\ \textbf{then}\ a + sumP\ x\ \textbf{else}\ sumP\ x \end{aligned}$$

A reasoning very similar to the case of *sum* leads to the following divide-and-conquer implementation:

$$\begin{aligned} sumP\ (l +\!\!+ r) &= (\delta x.\ foldr\ f\ x\ l)\ (sumP\ r) \\ &\quad \textbf{where}\ f\ a\ v = \textbf{if}\ a > 0\ \textbf{then}\ a + v\ \textbf{else}\ v \end{aligned}$$

This divide-and-conquer implementation passes the typechecking[5] of λ^{AS}, which means that the conditional expression is harmless. The comparison operator, $<$, does not access polynomials, in particular, the indeterminate generated by the δ abstraction; therefore, the conditional does not interfere with the algebraic simplification.

Note that this situation is different from that of the following mps', which is equivalent to *mps* discussed in Section 2.3 but uses a conditional branch instead of the binary maximum operator:

$$\begin{aligned} mps'\ [\,] &= 0 \\ mps'\ (a : x) &= \textbf{if}\ a + mps'\ x > 0\ \textbf{then}\ a + mps'\ x\ \textbf{else}\ 0 \end{aligned}$$

If we try to derive divide-and-conquer implementation of mps', the comparison operator, $>$, will compare polynomials that may contain indeterminates. This is not allowed because it is impossible to effectively simplify expressions that contain comparisons between polynomials. Therefore, for parallelising mps', we should replace the conditional branch by a semiring operator. Fortunately, in this case, the conditional can be replaced by a binary maximum operator, which forms a semiring with $+$. In this way, λ^{AS} enables us to distinguish harmful conditionals from harmless ones and thereby provides a guideline for reduction parallelisation.

[5] Strictly speaking, the program is not typeable because each list element, a, is accessed by $<$ and a should have the semiring type. We can avoid this problem using a function that lifts usual values to semiring values. For example, using $lift_{\mathbb{Z}}\ \mathbb{Z} \to \mathscr{R}$, the recursive case of *sumP* could be $sumP\ (a : x) = \textbf{if}\ a > 0\ \textbf{then}\ (lift_{\mathbb{Z}}\ a) + sumP\ x\ \textbf{else}\ sumP\ x$. For simplicity of presentation, this paper neglects this issue.

https://doi.org/10.1017/S0956796821000058 Published online by Cambridge University Press

2.5 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:

$$sumL = foldl\ (+)\ 0$$

The function can be reasoned as follows:

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

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

Because of the associativity and commutativity of the addition, *sumL* is observationally equivalent to *sum*. Though this equivalence enables us to parallelise *sumL*, we did not use it because the use of case-specific properties makes the reasoning less scalable. Next, we will demonstrate that the same approach can deal with a more complicated example with jumps.

Example 5 (Loop with Jump)**.** 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 parallelisation is possible:

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

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,

https://doi.org/10.1017/S0956796821000058 Published online by Cambridge University Press

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:

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

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

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

This parallel implementation works correctly, as the following example shows

$$
\begin{aligned}
& \mathit{sumB1}\ 0\ [3, 1, -5, 4, 2, -3, 7, 1] \\
\to \quad & (\mathit{sumB1}'\ 0\ [3, 1, -5, 4])\ (\delta x.\ \mathit{sumB1}\ x\ [2, -3, 7, 1]) \\
\rightrightarrows \quad & (\lambda f.\ 4)\ (\lambda x.\ x + 2) \\
\to \quad & 4
\end{aligned}
$$

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

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

As in the case of mps', λ^{AS} does not allow this program because the semiring value stored in e cannot be manipulated by $<$. Moreover, the conditional branch can be replaced by neither the maximum nor minimum operator. In fact, *sumB2* cannot be parallelised by a simple divide-and-conquer approach as in the case of *sumB1*.

2.6 Prefix sum

Prefix sums, also known as scans (Ladner & Fischer, 1980; Blelloch, 1993), 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:

$$\mathit{psum}\ [a_0, a_1, \ldots, a_n]\ e = [e, e + a_0, e + a_0 + a_1, \ldots, e + a_0 + a_1 + \cdots + a_{n-1}]$$

https://doi.org/10.1017/S0956796821000058 Published online by Cambridge University Press

The following equational reasoning leads to a parallel implementation:[6]

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

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

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

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:

$$
\begin{aligned}
\mathit{psum}\ (l +\!\!+ r) \ = \ & \mathbf{let}\ s = \delta e.\ \mathit{sumL}\ e\ l \\
& \quad [\!]\ \ r' = \delta e.\ \mathit{psum}\ r\ e \\
& \mathbf{in}\ \lambda e.\ \mathit{psum}\ l\ e +\!\!+ r'\ (s\ e)
\end{aligned}
$$

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:

[6] Strictly 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. This is not problematic, as we will later discuss in Section 4.3.

https://doi.org/10.1017/S0956796821000058 Published online by Cambridge University Press

$$
\begin{array}{ll}
 & \mathit{psum}\ (l +\!\!+ r) \\
= & \quad \{ \text{ the first parallel implementation } \} \\
 & \lambda e.\ \mathit{psum}\ l\ e +\!\!+ (\delta e.\ \mathit{psum}\ r\ e)\ (\mathit{sumL}\ e\ l) \\
= & \quad \{ \text{ introducing } \delta \text{ abstraction } \} \\
 & \lambda e.\ \mathit{psum}\ l\ e +\!\!+ (\delta e.\ \mathit{psum}\ r\ e)\ ((\delta e.\ \mathit{sumL}\ e\ l)\ e) \\
= & \quad \{ \text{ introducing } \mathbf{let} \} \\
 & \lambda e.\ (\mathbf{let}\ s = \delta e.\ \mathit{sumL}\ e\ l \ [\!]\ r' = \delta e.\ \mathit{psum}\ r\ e\ \mathbf{in}\ \mathit{psum}\ l\ e +\!\!+ r'\ (s\ e)) \\
= & \quad \{ \text{ swapping } \mathbf{let} \text{ and the outermost } \lambda \text{ abstraction } \} \\
 & \mathbf{let}\ s = \delta e.\ \mathit{sumL}\ e\ l \ [\!]\ r' = \delta e.\ \mathit{psum}\ r\ e\ \mathbf{in}\ \lambda e.\ \mathit{psum}\ l\ e +\!\!+ r'\ (s\ e)
\end{array}
$$

Nevertheless, this implementation shows a different behaviour from that of the previous one:

$$
\begin{array}{ll}
 & \mathit{psum}\ [3,1,7,5,2,6,9,4]\ 0 \\
\to & (\mathbf{let}\ s = \delta e.\ \mathit{sumL}\ e\ [3,1] \ [\!]\ r' = \delta e.\ \mathit{psum}\ [7,5,2,6,9,4]\ e \\
 & \ \mathbf{in}\ \lambda e.\ \mathit{psum}\ [3,1]\ 0 +\!\!+ r'\ (s\ e))\ 0 \\
\to & (\mathbf{let}\ s = \delta e.\ \mathit{sumL}\ e\ [3,1] \\
 & \quad [\!]\ r' = \delta e.\ (\mathbf{let}\ s = \delta e.\ \mathit{sumL}\ e\ [7,5] \\
 & \qquad\qquad [\!]\ r' = \delta e.\ (\mathbf{let}\ s = \delta e.\ \mathit{sumL}\ e\ [2,6] \ [\!]\ r' = \delta e.\ \mathit{psum}\ [9,4]\ e \\
 & \qquad\qquad\qquad\qquad \mathbf{in}\ \lambda e.\ \mathit{psum}\ [2,6]\ e +\!\!+ r'\ (s\ e))\ e \\
 & \qquad\qquad \mathbf{in}\ \lambda e.\ \mathit{psum}\ [7,5]\ e +\!\!+ r'\ (s\ e))\ e \\
 & \ \mathbf{in}\ \lambda e.\ \mathit{psum}\ [3,1]\ e +\!\!+ r'\ (s\ e))\ 0 \\
\rightrightarrows & (\lambda e.\ \mathit{psum}\ [3,1]\ e +\!\!+ \\
 & \quad (\lambda e.\ \mathit{psum}\ [7,5]\ e +\!\!+ \\
 & \qquad (\lambda e.\ \mathit{psum}\ [2,6]\ e +\!\!+ \\
 & \qquad\quad (\lambda e.\ [e, e+9])\ ((\lambda e.\ e+8)\ e))\ ((\lambda e.\ e+12)\ e))\ ((\lambda e.\ e+4)\ e))\ 0 \\
\to & \mathit{psum}\ [3,1]\ 0 +\!\!+ \mathit{psum}\ [7,5]\ 4 +\!\!+ \mathit{psum}\ [2,6]\ 16 +\!\!+ (\lambda e.\ [e, e+9])\ 24 \\
\rightrightarrows & [0,3] +\!\!+ [4,11] +\!\!+ [16,18] +\!\!+ [24,33]
\end{array}
$$

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 development for *psum* is not specific to summation. The same can be applied to any computation that records the intermediate results of a reduction expressed by a linear expression over a semiring. For example, similar algorithms can calculate all partial summations of a power series.

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

https://doi.org/10.1017/S0956796821000058 Published online by Cambridge University Press

2.7 Beyond list processing

Existing reduction parallelisation 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:

$$\begin{aligned} sum_{\mathrm{Tree}}\ (\mathrm{Nd}\ n\ l\ r) &= n + sum_{\mathrm{Tree}}\ l + sum_{\mathrm{Tree}}\ r \\ sum_{\mathrm{Tree}}\ (\mathrm{Lf}\ n) &= n \end{aligned}$$

This kind of bottom-up approach has been commonly used for parallel tree processing. While it is suitable for processing balanced trees, it cannot achieve sufficient parallel speedup 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 sum'_{Tree} that satisfies the following equation:

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

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

$$\begin{aligned} & sum'_{\mathrm{Tree}}\ c\ x \\ = \quad & \{\ \text{the characteristic equation above}\ \} \\ & sum_{\mathrm{Tree}}\ (c[t'])\quad \textbf{where}\ x = sum_{\mathrm{Tree}}\ t' \\ = \quad & \{\ \text{suppose}\ c[t'] = t'\ (\text{i.e., } c = \bullet)\ \} \\ & sum_{\mathrm{Tree}}\ t'\quad \textbf{where}\ x = sum_{\mathrm{Tree}}\ t' \\ = \quad & \{\ \text{definition of}\ x\ \} \\ & x \end{aligned}$$

$$\begin{aligned} & sum'_{\mathrm{Tree}}\ c\ x \\ = \quad & \{\ \text{the characteristic equation above}\ \} \\ & sum_{\mathrm{Tree}}\ (c[t'])\quad \textbf{where}\ x = sum_{\mathrm{Tree}}\ t' \\ = \quad & \{\ \text{suppose}\ c[t'] = \mathrm{Nd}\ n\ (l[t'])\ r\ \} \\ & n + sum_{\mathrm{Tree}}\ (l[t']) + sum_{\mathrm{Tree}}\ r\quad \textbf{where}\ x = sum_{\mathrm{Tree}}\ t' \\ = \quad & \{\ \text{induction}\ \} \\ & n + sum'_{\mathrm{Tree}}\ l\ x + sum_{\mathrm{Tree}}\ r \end{aligned}$$

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

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

https://doi.org/10.1017/S0956796821000058 Published online by Cambridge University Press

Because the computation of sum'_{Tree} consists of additions, $(\delta x.\ sum'_{\mathrm{Tree}}\ c\ x)$ can be computed independently with $sum_{\mathrm{Tree}}\ t'$.

The definition of sum'_{Tree} shows the possibility of processing independent subtrees in parallel, and 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:

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

Note on tree division strategy

The approach, namely dividing a tree into a subtree and a context, is influenced by the theory of parallel tree contraction (Reid-Miller *et al.*, 1993; Morihata *et al.*, 2009; Morihata & Matsuzaki, 2011) that enables efficient parallel tree reduction regardless of the tree shape. It is a generalisation of the divide-and-conquer approach for parallel list processing. If x is a list, then $x = c[x']$ is equivalent to $x = c +\!\!+ x'$. Besides, the approach subsumes the bottom-up tree processing, which divides Nd $n\ l\ r$ into Nd $n\ [\bullet]\ r$ and l.

As similar to the case of parallel list processing, the equations about sum_{Tree} developed so far do not correspond to valid function definitions in Haskell. Instead, they show the correctness of *any* parallel tree reduction in which each task corresponds to a subtree or (one-hole) context. In other words, they correspond to several concrete implementations, including the reduction to parallel list processing (Reid-Miller *et al.*, 1993) and transformation to balanced tree processing (Morihata & Matsuzaki, 2011).

The approach is somewhat similar to lazy tree splitting (Bergstrom *et al.*, 2012) that generates tasks in a by-need manner during tree traversal expressed by Huet's zippers (Huet, 1997). However, each task generated by lazy tree splitting corresponds to a subtree, and hence, the lazy tree splitting approach cannot achieve sufficient parallel speed-up for list-like trees.

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` $(e)\ s_1$ `else` s_2 and a loop statement like

https://doi.org/10.1017/S0956796821000058 Published online by Cambridge University Press

`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:

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

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: $\cup$ and *remove* v y can be regarded as $\vee$ 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. We regard $\vee$ and $\wedge$ as the addition and multiplication, respectively; then, *rd* does not involve a non-linear multiplication because a left operand of a multiplication is always a constant, $\neg v$. 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. To resolve the dependency, we require the computation for $c[\bullet]$ to return two values: an accumulation parameter y' passed to t, and a function f_c that takes the result of t:

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

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

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

The reasoning to this point leads to the following equation:

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

https://doi.org/10.1017/S0956796821000058 Published online by Cambridge University Press

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

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

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 generalises 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:

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

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

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

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:

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

https://doi.org/10.1017/S0956796821000058 Published online by Cambridge University Press

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

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

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 $\mathscr{R} = (S, \oplus, \otimes, \overline{0}, \overline{1})$, we may use $\mathscr{R}$ and S interchangeably if the meaning is apparent from the context. For example, we may write $s \in \mathscr{R}$, that is, 's is an element of $\mathscr{R}$', instead of $s \in S$.

Given a set of indeterminates X (X should be disjoint from $\mathscr{R}$) and a semiring $\mathscr{R} = (S, \oplus, \otimes, \overline{0}, \overline{1})$, a polynomial of the following form, where $c_0, c_1, \ldots, c_m \in S$ and $x_1, x_2, \ldots, x_m \in X$:

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

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

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

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 4 discusses further extensions:

https://doi.org/10.1017/S0956796821000058 Published online by Cambridge University Press

$$
\begin{array}{ll}
(\lambda x^{\tau}.\, e)\ v \to e[v/x] & \\
e_1\ e_2 \to e_1'\ e_2' & \text{if } e_1 \to e_1' \text{ and } e_2 \to e_2', \text{ or, } e_i = e_i' \text{ and } e_j \to e_j'\ (i,j \in \{1,2\}, i \neq j) \\
e_1 \oplus e_2 \to e_1' \oplus e_2' & \text{if } e_1 \to e_1' \text{ and } e_2 \to e_2', \text{ or, } e_i = e_i' \text{ and } e_j \to e_j'\ (i,j \in \{1,2\}, i \neq j) \\
e_1 \otimes e_2 \to e_1' \otimes e_2' & \text{if } e_1 \to e_1' \text{ and } e_2 \to e_2', \text{ or, } e_i = e_i' \text{ and } e_j \to e_j'\ (i,j \in \{1,2\}, i \neq j) \\
\delta x.\, e \to \delta x.\, e' & \text{if } e \to e' \\
\delta x.\, v \to \lambda x.\, v & \\
(c_0 \oplus (c_1 \otimes x_1) \oplus \cdots \oplus (c_m \otimes x_m)) \oplus (c_0' \oplus (c_1' \otimes x_1) \oplus \cdots \oplus (c_m' \otimes x_m)) & \\
\quad \to c_0'' \oplus (c_1'' \otimes x_1) \oplus \cdots \oplus (c_m'' \otimes x_m) & \text{where } \mathscr{R} \ni c_i'' = c_i \oplus c_i'\ (0 \le i \le m) \\
c_0 \otimes (c_0' \oplus (c_1' \otimes x_1) \oplus \cdots \oplus (c_m' \otimes x_m)) & \\
\quad \to c_0'' \oplus (c_1'' \otimes x_1) \oplus \cdots \oplus (c_m'' \otimes x_m) & \text{where } \mathscr{R} \ni c_i'' = c_0 \otimes c_i'\ (0 \le i \le m) \\
(c_0 \oplus (c_1 \otimes x_1) \oplus \cdots \oplus (c_m \otimes x_m)) \otimes c_0' & \\
\quad \to c_0'' \oplus (c_1'' \otimes x_1) \oplus \cdots \oplus (c_m'' \otimes x_m) & \text{where } \mathscr{R} \ni c_i'' = c_i \otimes c_0'\ (0 \le i \le m)
\end{array}
$$

Fig. 3. Reduction rules for λ^{AS}.

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

A metavariable x is used to denote a variable (or indeterminate). $\mathscr{R}$ denotes the underlying semiring, and c is a value in $\mathscr{R}$. Each base type, $\mathscr{R}$, 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. Later, Section 4.1 extends the theory developed here to non-commutative semirings"

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

Values are functions and linear polynomials. 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. For simplicity, 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 $x:\tau$, that is, $\Gamma\{x:\tau\}(x) = \tau$, and

https://doi.org/10.1017/S0956796821000058 Published online by Cambridge University Press

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x^{\tau} : \tau} \qquad \frac{\Gamma\{x : \tau\} \vdash e : \tau'}{\Gamma \vdash \lambda x^{\tau}.\, e : \tau \to \tau'} \qquad \frac{\Gamma \vdash e_1 : \tau \to \tau' \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1\ e_2 : \tau'}$$

$$\frac{\Gamma\{x : \mathscr{R}_{\alpha}\} \vdash e : \tau' \qquad \Gamma\{x : \mathscr{R}_{\mathrm{P}}\} \vdash e : \tau''}{\Gamma \vdash \delta x^{\mathscr{R}}.\, e : \mathscr{R}_{\alpha} \to \tau'} \qquad \Gamma \vdash c : \mathscr{R}_{\alpha}$$

$$\frac{\Gamma \vdash e_1 : \mathscr{R}_{\alpha} \quad \Gamma \vdash e_2 : \mathscr{R}_{\alpha}}{\Gamma \vdash e_1 \oplus e_2 : \mathscr{R}_{\alpha}} \qquad \frac{\Gamma \vdash e_1 : \mathscr{R}_{\mathrm{C}} \quad \Gamma \vdash e_2 : \mathscr{R}_{\alpha}}{\Gamma \vdash e_1 \otimes e_2 : \mathscr{R}_{\alpha}} \qquad \frac{\Gamma \vdash e_1 : \mathscr{R}_{\alpha} \quad \Gamma \vdash e_2 : \mathscr{R}_{\mathrm{C}}}{\Gamma \vdash e_1 \otimes e_2 : \mathscr{R}_{\alpha}}$$

Fig. 4. Typing rules for λ^{AS}.

$\Gamma\{x : \tau\}(y) = \Gamma(y)$ 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 $\mathscr{R}_{\mathrm{C}}$ 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 $\delta x^{\mathscr{R}}.\, e$. Because e is to be simplified before the argument is passed, e should be typeable even if x is an indeterminate and therefore of $\mathscr{R}_{\mathrm{P}}$ type. Moreover, because $\delta x^{\mathscr{R}}.\, e$ is regarded as a usual function after the simplification of e, $\delta x^{\mathscr{R}}.\, e$ should have the same type as $\lambda x^{\mathscr{R}}.\, e$. Accordingly, the body, e, is typechecked twice. Note that the following simpler rule is safe but too restrictive:

$$\frac{\Gamma\{x : \mathscr{R}_{\mathrm{P}}\} \vdash e : \tau'}{\Gamma \vdash \delta x^{\mathscr{R}}.\, e : \mathscr{R}_{\mathrm{P}} \to \tau'}$$

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

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 call-by-value simply typed lambda calculus extended by a speculative simplification. In the following, we show that the speculative simplification of λ^{AS} is not problematic.

The formalisation 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:

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

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

https://doi.org/10.1017/S0956796821000058 Published online by Cambridge University Press

First, evaluation of λ^{AS} is terminating. Let $\to^*$ be the reflective transitive closure of $\to$.

Theorem 1. *For any λ^{AS} term e, there exists e' such that $e \to^* e'$ and no e'' satisfies $e' \to e''$.*

Proof. Consider another reduction relation $\twoheadrightarrow$, defined as follows:

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

Because $\twoheadrightarrow$ enables more reductions than $\to$, the termination of reductions by $\to$ follows from that by $\twoheadrightarrow$.

Note that $\twoheadrightarrow$ expresses β reductions with algebraic simplifications using semiring properties, and the algebraic simplification can be regarded as a convergent (i.e., confluent and terminating) rewriting system. Because a rewriting system obtained by combining convergent rules with β reductions of the simply typed lambda calculus is convergent (Tannen, 1988; Okada, 1989; Tannen & Gallier, 1991), $\twoheadrightarrow$ is terminating. □

Second, the following theorem states that a δ abstraction is observationally equivalent to a λ abstraction.[7]

Theorem 2. *For any λ^{AS} term e and λ^{AS} context C, if $C[\delta x^{\mathscr{R}}.\ e] \to^* c_1$ and $C[\lambda x^{\mathscr{R}}.\ e] \to^* c_2$, where $c_1, c_2 \in \mathscr{R}$, then $c_1 = c_2$.*

Proof. Consider the reduction relation $\twoheadrightarrow$ defined in the proof of Theorem 1. Then, from the definition of $\twoheadrightarrow$ and the assumption, $C[\delta x^{\mathscr{R}}.\ e] \twoheadrightarrow^* c_1$ and $C[\delta x^{\mathscr{R}}.\ e] \twoheadrightarrow C[\lambda x^{\mathscr{R}}.\ e] \twoheadrightarrow^* c_2$ hold. Recall that $\twoheadrightarrow$ is confluent (Tannen, 1988; Okada, 1989; Tannen & Gallier, 1991); hence, there should exists $c^\dagger$ such that $c_1 \twoheadrightarrow^* c^\dagger$ and $c_2 \twoheadrightarrow^* c^\dagger$. Because $c_1, c_2 \in \mathscr{R}$, this is possible only if $c_1 = c_2$. □

Theorem 2 only considers successful evaluations that yield values. This is not a serious limitation if we consider only typeable terms. As Theorem 1 shows, evaluations of λ^{AS} terms terminate. Moreover, 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 $fvs(e)$ denote the set of all free variables in e, that is, variables not bound by any λ abstraction or δ abstraction. We assume that every free variable is an indeterminate and thus has type $\mathscr{R}_{\mathrm{P}}$ because free variables except for indeterminates cannot be values in λ^{AS}.

Lemma 3. *Assume that $\Gamma \vdash v : \tau$, and that $\Gamma(x) = \mathscr{R}_{\mathrm{P}}$ for any $x \in fvs(v)$; then, the following equations hold*

[7] The theorem only considers evaluations that lead to base-type values. This is essential. $\delta x^{\mathscr{R}}.\ e$ and $\lambda x^{\mathscr{R}}.e$ may lead to syntactically different functions because the former performs speculative simplifications.

https://doi.org/10.1017/S0956796821000058 Published online by Cambridge University Press

- $v = c$ *if* $\tau = \mathscr{R}_C$.
- $v = c_0 \oplus (c_1 \otimes x_1) \oplus \cdots \oplus (c_m \otimes x_m)$, *where* $\{x_1, \ldots, x_m\} = fvs(v)$, *if* $\tau = \mathscr{R}_P$.
- $v = \lambda x^{\tau_1}.\ e'$ *if* $\tau = \tau_1 \to \tau_2$.

Proof. The proof follows immediately from the typing rules of λ^{AS}. □

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

Theorem 4. *If* $\Gamma \vdash e : \tau$ *and* $e \to e'$, *then* $\Gamma \vdash e' : \tau$.

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

Theorem 5. *If* $\Gamma \vdash e : \tau$ *and* $\Gamma(x) = \mathscr{R}_P$ *for any* $x \in fvs(e)$, *then there exists* e' *such that* $e \to e'$ *unless* e *is a value.*

Proof. The proof uses an induction over the structure of e. Every case is easily proved using Lemma 3. Note that it is safe to regard any $x \in fvs(e)$ as an indeterminate of a polynomial because $\Gamma(x) = \mathscr{R}_P$. □

The following corollary summarises Theorems 1, 4 and 5.

Corollary 6. *For any* λ^{AS} *term* e *and* λ^{AS} *context* C *such that* $\vdash C[\delta x^{\mathscr{R}}.\ e] : \mathscr{R}_C$, *there exists* $c \in \mathscr{R}$ *such that* $C[\delta x^{\mathscr{R}}.\ e] \to^* c$ *and* $C[\lambda x^{\mathscr{R}}.\ e] \to^* c$.

3.5 Encoding by Hindley–Milner typing

The type system of λ^{AS} is not satisfactory from a practical perspective. It is difficult to assign $\mathscr{R}_P$ and $\mathscr{R}_C$ appropriately.

As an simple example, consider $dbl = \lambda x.\ x + x$ (type annotations are omitted because no annotation is appropriate). The possible type of *dbl* is either $\mathbb{Z}_P \to \mathbb{Z}_P$ or $\mathbb{Z}_C \to \mathbb{Z}_C$. In fact, both are inappropriate: if *dbl* is of the former type, $\delta x.\ (dbl\ 1) \times x$ cannot be typechecked; if *dbl* is of the latter type, $\delta x.\ dbl\ x$ cannot.

This problem is not specific to *dbl*. Most of the expressions can calculate both polynomials and constants, depending on the inputs (or free variables) passed. This observation is not expressed in the type system. Instead, the type system blindly generates all possibilities and tried to find a possible assignment. In particular, the rule for a multiplication requires examining two possibilities, and the rule for a δ abstraction requires typechecking the body twice.

A possible solution to this problem is to introduce polymorphism, such as $dbl : \forall \alpha \in \{\mathbb{Z}_C, \mathbb{Z}_P\}.\ \alpha \to \alpha$. In particular, a promising approach is to encode the type system of λ^{AS} using the standard Hindley–Milner type system. This approach is beneficial from several perspectives.

https://doi.org/10.1017/S0956796821000058 Published online by Cambridge University Press

- It allows polymorphic types not only for base types.
- It enables us to encode the type system of λ^{AS} in widely used languages such as Haskell and OCaml.
- It avoids costly brute-force searches and makes it possible to use existing efficient type inference algorithms.
- It avoids inferring $\mathscr{R}_{\mathrm{P}}$ for every base-type expressions because the Hindley–Milner type system can infer the principle type schema.

The idea of using the Hindley–Milner type system is based on the following observation. Recall that *dbl* can take an argument of either $\mathbb{Z}_{\mathrm{C}}$ or $\mathbb{Z}_{\mathrm{P}}$. This situation can be naturally expressed by the **let** polymorphism. Moreover, we can similarly solve inefficiency in type-checking δ abstractions. The body of a δ abstraction is essentially evaluated twice, and that the two evaluations may take arguments of different types. Using a **let** expression, we can informally express this situation by the following equation. Here, $\lfloor x \rfloor$ denotes an indeterminate rather than a variable:

$$\delta x^{\mathscr{R}}.\ e \equiv \mathbf{let}\ y = (\lambda x^{\mathscr{R}}.\ e)\ \lfloor x \rfloor\ \mathbf{in}\ \lambda x^{\mathscr{R}}.\ y$$

That is, $\delta x^{\mathscr{R}}.\ e$ first takes $\lfloor x \rfloor$ as its argument, and after that, additionally takes the actual argument. Based on this observation, we can check the following expression instead of directly typechecking $\delta x^{\mathscr{R}}.\ e$:

$$\mathbf{let}\ f = \lambda x^{\mathscr{R}}.\ e\ \ \mathbf{in}\ \mathbf{let}\ \_ = f\ \lfloor x \rfloor\ \mathbf{in}\ f$$

That is, $\delta x^{\mathscr{R}}.\ e$ is essentially regarded as $\lambda x^{\mathscr{R}}.\ e$, but in addition, its applicability to an indeterminate $\lfloor x \rfloor :: \mathscr{R}_{\mathrm{P}}$ is checked.[8]

The polymorphism of the Hindley–Milner type system can encode subtyping (Fluet & 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 $\forall\alpha.\ \mathscr{R}_\alpha$ and $\forall\alpha.\ \mathscr{R}_\alpha \to \mathscr{R}_\alpha \to \mathscr{R}_\alpha$, 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 $\lambda x^{\mathscr{R}}.\ \lambda y^{\mathscr{R}}.\ x \otimes y$ is either $\forall\alpha.\ \mathscr{R}_\alpha \to \mathscr{R}_{\mathrm{C}} \to \mathscr{R}_\alpha$ or $\forall\alpha.\ \mathscr{R}_{\mathrm{C}} \to \mathscr{R}_\alpha \to \mathscr{R}_\alpha$, and these two types are incomparable. A natural workaround is to use two kinds of type-annotated multiplications, $(\otimes_{\mathrm{L}}) :: \forall\alpha.\ \mathscr{R}_{\mathrm{C}} \to \mathscr{R}_\alpha \to \mathscr{R}_\alpha$ and $(\otimes_{\mathrm{R}}) :: \forall\alpha.\ \mathscr{R}_\alpha \to \mathscr{R}_{\mathrm{C}} \to \mathscr{R}_\alpha$. This modification may, however, make some typeable terms not typeable. For instance, although $\mathbf{let}\ f = \lambda x^{\mathscr{R}}.\ \lambda y^{\mathscr{R}}.\ x \otimes y\ \mathbf{in}\ \delta x^{\mathscr{R}}.\ f\ 1\ x \oplus f\ x\ 1$ is typeable if $\otimes$ is commutative, neither $\otimes_{\mathrm{L}}$ nor $\otimes_{\mathrm{R}}$ can be used instead of $\otimes$.

The discussion above shows a trade-off between advanced types (e.g., polymorphism) and the necessity of annotations. While advanced types are generally more expressive, because of the difficulty of their inference, the type system often requires programmers more type annotations. In the current situation of using the Hindley–Milner type system for λ^{AS}, the cost seems acceptable relative to the benefit.

We have introduced in Section 3.3 a monomorphic type system and did not regard the use of the Hindley–Milner type system as the default choice. It is because adoption of

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

https://doi.org/10.1017/S0956796821000058 Published online by Cambridge University Press

other kinds of type systems, including those equipped with structural subtyping, generalised algebraic data types, refinement types and full dependent types, could be beneficial. They enable us to count the number of indeterminates in the type level, and therefore, would be useful to infer not only possibilities of getting stuck but also overheads for simplifying polynomials. Nevertheless, as discussed above, their use may not come for free and require additional type annotations. In summary, there exists a design choice between type systems. Further study remains as future work.

4 Discussions

4.1 Non-commutative semiring

So far, we have considered commutative semirings. If $\otimes$ is not commutative, simplifications become more difficult. For instance, neither $c \otimes x \otimes c'$ nor $(c_1 \otimes x \otimes c'_1) \oplus (c_2 \otimes x \otimes c'_2)$ ($c_1 \neq c'_1$ and $c_2 \neq c'_2$) can be simpler.[9] 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 and multiplication between a left-linear polynomial and a right-linear polynomial should be prohibited.

We can refine the operational semantics of additions and multiplications as follows:

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

In the type system, $\mathscr{R}$ 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:

$$
\frac{\Gamma\{x : \mathscr{R}_\alpha\} \vdash e : \tau' \qquad \Gamma\{x : \mathscr{R}_\beta\} \vdash e : \tau'' \qquad \beta \in \{\mathrm{LP}, \mathrm{RP}\}}{\Gamma \vdash \delta x^{\mathscr{R}}.\ e : \mathscr{R}_\alpha \to \tau'}
$$

$$
\frac{\Gamma \vdash e_1 : \mathscr{R}_\mathrm{C} \qquad \Gamma \vdash e_2 : \mathscr{R}_\beta \qquad \beta \in \{\mathrm{LP}, \mathrm{C}\}}{\Gamma \vdash e_1 \otimes e_2 : \mathscr{R}_\beta} \qquad \frac{\Gamma \vdash e_1 : \mathscr{R}_\beta \qquad \beta \in \{\mathrm{RP}, \mathrm{C}\} \qquad \Gamma \vdash e_2 : \mathscr{R}_\mathrm{C}}{\Gamma \vdash e_1 \otimes e_2 : \mathscr{R}_\beta}
$$

The type system can be encoded by the Hindley–Milner type system. When using a non-commutative semiring $\mathscr{R}$, every base type has two kinds of annotations: left-linear L or right-linear R, and constant C or polynomial P. Accordingly, we can encode the types of semiring values, semiring operators and indeterminates as follows:

[9] The non-commutativity of the multiplication should not be confused with the type-annotated multiplications considered in Section 3.5. If $\otimes$ is commutative, $(c_1 \otimes_\mathrm{L} x) \otimes_\mathrm{R} c_2$ is a simplifiable (and typeable) expression.

https://doi.org/10.1017/S0956796821000058 Published online by Cambridge University Press

$$
\begin{aligned}
c &:: \forall \alpha, \beta.\ \mathscr{R}_{\alpha,\beta} \\
(\oplus) &:: \forall \alpha, \beta.\ \mathscr{R}_{\alpha,\beta} \to \mathscr{R}_{\alpha,\beta} \to \mathscr{R}_{\alpha,\beta} \\
(\otimes_{\mathrm{L}}) &:: \forall \alpha, \beta.\ \mathscr{R}_{\alpha,\mathrm{C}} \to \mathscr{R}_{\mathrm{L},\beta} \to \mathscr{R}_{\mathrm{L},\beta} \\
(\otimes_{\mathrm{R}}) &:: \forall \alpha, \beta.\ \mathscr{R}_{\mathrm{R},\beta} \to \mathscr{R}_{\alpha,\mathrm{C}} \to \mathscr{R}_{\mathrm{R},\beta} \\
\lfloor x \rfloor &:: \forall \alpha.\ \mathscr{R}_{\alpha,\mathrm{P}}
\end{aligned}
$$

As in the case of commutative semirings, this encoding rejects some typeable terms. For instance, $(1 \otimes_{\mathrm{L}} 1) \oplus (1 \otimes_{\mathrm{R}} 1)$ cannot pass the typechecking based on this encoding.

Although these refinements make the whole calculus more complicated, they maintain the major properties including Corollary 6.

Example 9 (list concatenations)**.** As an application of non-commutative semiring, consider a function, *pElem*, which gathers positive elements:

$$
\begin{aligned}
\mathit{pElem}\ [\,] &= [\,] \\
\mathit{pElem}\ (a : x) &= \mathbf{if}\ a > 0\ \mathbf{then}\ [a] +\!\!+ \mathit{pElem}\ x\ \mathbf{else}\ \mathit{pElem}\ x
\end{aligned}
$$

Reasoning similar to the case of *sumP* discussed in Section 2.4 leads to the following divide-and-conquer implementation:

$$
\begin{aligned}
\mathit{pElem}\ (l +\!\!+ r) &= (\delta x.\ \mathit{foldr}\ f\ x\ l)\ (\mathit{pElem}\ r) \\
&\quad \mathbf{where}\ f\ a\ v = \mathbf{if}\ a > 0\ \mathbf{then}\ [a] +\!\!+ v\ \mathbf{else}\ v
\end{aligned}
$$

By regarding $+\!\!+$ as the multiplication operator, we can see that this program calculates left-linear polynomials and hence parallelisable.

Note that $+\!\!+$ cannot be the addition operator because of its non-commutativity. Commutativity of the addition operator is essential. Otherwise, we cannot simplify expressions like $x_1 \oplus x_2 \oplus x_1$, and therefore, sizes of polynomials cannot be bound by the number of indeterminates.

4.2 Multiplication without associativity

Sometimes, we can drop even associativity for the multiplication.

As an example, consider the 'cons' operator, (:), for lists. While it does not satisfy associativity, a related associative operator, $(+\!\!+)$, enables simplification. For example, a linear expression $a : b : c : x$, where x is an indeterminate, can be simplified to $w +\!\!+ x$ where $w = [a, b, c]$. In general, any linear expression written using the cons operator can be simplified to the form of $w +\!\!+ x$, where w is a list and x is an indeterminate. In fact, we have implicitly used this simplification strategy in Example 9.

Many practical operators have related associative operators that enable simplification.

> **Division:** For an indeterminate x and numbers a_1, a_2, a_3, $x \,/\, a_1 \,/\, a_2 \,/\, a_3$ can be simplified to $x \,/\, a$ where $a = a_1 \times a_2 \times a_3$.
> **Matrix multiplications (Matsuzaki *et al.*, 2006; Sato & Iwasaki, 2011):** For an indeterminate vector x and matrices A_1, A_2, A_3, $A_1(A_2(A_3x))$ can be simplified to Ax where $A = A_1A_2A_3$.

https://doi.org/10.1017/S0956796821000058 Published online by Cambridge University Press

Algebraic data structures: For an indeterminate x and constructors of algebraic data structures $f_1, f_2, f_3, f_1\ (f_2\ (f_3\ x))$ can be simplified to $f\ x$, where $f = f_1\ (f_2\ (f_3\ \bullet))$ is a data structure with the unique hole, $\bullet$ (Minamide, 1998).

Formally, we can apply the simplification strategy if the following properties hold. Here we consider the case of left-linear expressions, and that of right-linear ones are analogous:

$$
\begin{array}{ll}
a \oplus (b \oplus c) = (a \oplus b) \oplus c & \{\text{ associativity of } \oplus \} \\
a \oplus b = b \oplus a & \{\text{ commutativity of } \oplus \} \\
a \oplus \overline{0} = \overline{0} \oplus a = a & \{\text{ additive identity }\} \\
a \otimes (b \otimes c) = (a \oslash b) \otimes c & \{\text{ simplification by } \oslash \} \\
\overline{1} \otimes a = a & \{\text{ left multiplicative identity }\} \\
a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c) & \{\text{ left distributivity }\}
\end{array}
$$

That is, $\otimes$ needs not to be associative; instead, $\oslash$ should be able to simplify a series of $\otimes$ applications. Note that $\oslash$ should be nearly associative, as the following reasoning shows

$$
\begin{array}{rcl}
(a \oslash (b \oslash c)) \otimes x & = & a \otimes ((b \oslash c) \otimes x) \\
& = & a \otimes (b \otimes (c \otimes x)) \\
& = & (a \oslash b) \otimes (c \otimes x) \\
& = & ((a \oslash b) \oslash c) \otimes x
\end{array}
$$

In particular, this generalises the case of a non-commutative semiring, in which $(\oslash) = (\otimes)$.

The simplification strategy can be implemented using the following reduction rule instead of the corresponding original one:

$$
\begin{array}{l}
c_0 \otimes (c'_0 \oplus (c'_1 \otimes x_1) \oplus \cdots \oplus (c'_m \otimes x_m)) \to c''_0 \oplus (c''_1 \otimes x_1) \oplus \cdots \oplus (c''_m \otimes x_m) \\
\quad \text{where } \mathscr{R} \ni c''_i = c_0 \oslash c'_i\ (0 \le i \le m)
\end{array}
$$

The type system is unnecessary to modify from the case of non-commutative semirings. This modification maintains the major properties including Corollary 6.

Example 10 (nondeterminisitc finite automata)**.** Let $\mathscr{A} = (Q, \Sigma, I, F, \tau)$ be a nondeterministic finite automaton, where Q is the set of the states, Σ is the alphabet, $I \subseteq Q$ is the set of the initial states, $F \subseteq Q$ is the set of the final states and $\tau \subseteq (Q \times \Sigma \times Q)$ is the transition relation. The following $run_{\mathscr{A}}$ expresses the state transition by $\mathscr{A}$:

$$
\begin{array}{c}
run_{\mathscr{A}} = foldl\ (\lambda a\ s.\ \{q' \mid q \in s,\ (q, q') \in \tau_a\})\ I \\
\textbf{where } \tau_a = \{(q, q') \mid (q, a, q') \in \delta\}
\end{array}
$$

One may expect that this program does not fit the parallelisation approach by λ^{AS} because there is no algebraic operator. In fact, it can be parallelised by reformulating the computation by a vector–matrix multiplication. Let $\{q_1, q_2, \ldots, q_m\} = Q$. First, $s \subseteq Q$ can be regarded as a bit vector of size m whose i-th bit is set 1 if $q_i \in s$. Second, each τ_a $(a \in \Sigma)$ can be regarded as a $m \times m$ matrix whose (i, j) element is set 1 if $(q_j, q_i) \in \tau_a$. Then, $\{q' \mid q \in s,\ (q, q') \in \tau_a\}$ can be calculated by a vector–matrix multiplication $\tau_a\ s$, where $\vee$

https://doi.org/10.1017/S0956796821000058 Published online by Cambridge University Press

and $\wedge$ are, respectively, the addition and multiplication. Because a vector–matrix multiplication has a related associative operator, that is, a matrix–matrix multiplication, $run_{\mathscr{A}}$ can be parallelised just like those examples discussed in Section 2.5.

Example 11 (tree transformation)**.** As mentioned above, constructions of algebraic data structures are equipped with an associative operator, which is the substitution operator for data structures with holes. This view enables us to parallelise computations that construct data structures. As a concrete example, consider the following simple tree transformation. The input is a tree consisting of four kinds of nodes: B, L, R and N. The output is obtained by eliminating the right and the left subtree of each L node and each R node, respectively. The following function tt does the transformation:

$$\begin{aligned} tt\ (\mathrm{B}\ t_1\ t_2) &= \mathrm{B}\ (tt\ t_1)\ (tt\ t_2) \\ tt\ (\mathrm{L}\ t_1\ t_2) &= tt\ t_1 \\ tt\ (\mathrm{R}\ t_1\ t_2) &= tt\ t_2 \\ tt\ \mathrm{N} &= \mathrm{N} \end{aligned}$$

As discussed in Section 2.7, splitting the input tree into a subtree and a context may improve parallelism. Formally, we would like to derive tt' that satisfies the following equation:

$$tt\ (c[t]) = (\delta x.\ tt'\ c\ x)\ (tt\ t)$$

It is not difficult to derive the following definition of tt':

$$\begin{aligned} tt'\ (\mathrm{B}\ (t_1[\bullet])\ t_2)\ x &= \mathrm{B}\ (tt'\ (t_1[\bullet])\ x)\ (tt\ t_2) \\ tt'\ (\mathrm{B}\ t_1\ (t_2[\bullet]))\ x &= \mathrm{B}\ (tt\ t_1)\ (tt'\ (t_2[\bullet])\ x) \\ tt'\ (\mathrm{L}\ (t_1[\bullet])\ t_2)\ x &= tt'\ (t_1[\bullet])\ x \\ tt'\ (\mathrm{L}\ t_1\ (t_2[\bullet]))\ x &= tt\ t_1 \\ tt'\ (\mathrm{R}\ (t_1[\bullet])\ t_2)\ x &= tt\ t_2 \\ tt'\ (\mathrm{R}\ t_1\ (t_2[\bullet]))\ x &= tt'\ (t_2[\bullet])\ x \\ tt'\ \bullet\ x &= x \end{aligned}$$

Because of the linearity of tt' and algebraic property of data structures with holes, this program can be evaluated in parallel just like sum_{Tree}.

While the formalism discussed so far is sufficient for capturing typical parallel reductions, there is a room for further generalisation. Any set of operators with a simplification strategy can be integrated to λ^{AS} if it satisfies desirable properties, such as termination, confluency and efficiency. Nevertheless, it is generally non-trivial to develop a type system that ensures the properties. For example, the linearity requirement of polynomials is not sufficient to guarantee the efficiency of tree transformations because using a tree with a hole more than once may require its duplication. Minamide (1998) provided a type system that guarantees the single use of each structure. While it seems possible to adapt this type system to λ^{AS}, a formal investigation is left for future work.

https://doi.org/10.1017/S0956796821000058 Published online by Cambridge University Press

4.3 Other programming constructs

λ^{AS} is so designed that it can be extended with standard program constructs such as conditionals, data structures and recursions. Note that Theorems 2, 4 and 5 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.

In Section 2.6, we have already used list constructors that store semiring values. They are not problematic. Data structures are merely containers, and hence, their application does not cause essential computation about semiring values. This intuition can be more formally justified by considering Church encoding for the structures.

It is also possible to add the fixed-point operator to λ^{AS}. Note that Theorems 2, 4 and 5 deal only with terminating evaluations. If an evaluation terminates in n steps, we can use an n-fold unfolding operator instead of the fixed-point operators.[10] 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, $\delta x^{\mathscr{R}}.\ e$ is observationally equivalent to $\lambda x^{\mathscr{R}}.\ e$ if evaluations of these two terms terminate. Note, however, that they may have different termination behaviors. For instance, when $\bot$ is non-terminating, $(\lambda x^{\tau}.\ 1)\ (\lambda z^{\mathscr{R}}.\ \bot)$ terminates, whereas $(\lambda x^{\tau}.\ 1)\ (\delta z^{\mathscr{R}}.\ \bot)$ does not.

4.4 More than one semiring

It is not difficult to deal with programs that use 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 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 if e is evaluated to a constant. Further investigation of this notion is left for future work.

4.5 Delimited continuation for introducing more δ abstractions

We have introduced parallelism by changing λ abstractions to δ abstractions. It is natural to consider introducing more parallelism by splitting expressions by inserting δ abstractions. Formally, given an expression $e_1[e_2]$, we would like to evaluate e_2 and its surrounding context $e_1[\bullet]$ in parallel by inserting a δ abstraction, that is, $(\delta x.\ e_1[x])\ e_2$. However, this approach is not always possible. For instance, consider $e = (\lambda x.\ x + ((\lambda y.\ y + x)\ 3) + x)\ 5 = e_1[e_2]$ where $e_1[\bullet] = (\lambda x.\ x + \bullet + x)\ 5$ and $e_2 = (\lambda y.\ y + x)\ 3$; then, e is not equivalent to

[10] This approach is known as Levy's labelled reduction technique (Lévy, 1976).

https://doi.org/10.1017/S0956796821000058 Published online by Cambridge University Press

$(\delta z.\ e_1[z])\ e_2$ because e_2 contains a free variable, x, whose actual value, 5, is specified in e_1.

A remedy to this situation is to use *delimited continuations*. In the following, we use the shift/reset operator (Danvy & Filinski, 1990).

A continuation represents the computation that will be performed later. A reset operator, $\langle e \rangle$, *delimits* a continuation. A shift operator, $\mathscr{S}k.\ e$, captures the current continuation up to the surrounding reset operator and binds it to the variable, k. For example, $\langle 3 + (\mathscr{S}k.\ k\ (k\ 2)) \rangle + 4$ is evaluated as follows:

$$\begin{array}{ll} & \langle 3 + (\mathscr{S}k.\ k\ (k\ 2)) \rangle + 4 \\ \rightarrow & \langle (\lambda k.\ k\ (k\ 2))\ (\lambda x.\ \langle 3 + x \rangle) \rangle + 4 \\ \rightarrow & \langle \langle 3 + \langle 3 + 2 \rangle \rangle \rangle + 4 \\ \rightarrow & 12 \end{array}$$

Usefulness of continuations for modelling concurrency and parallelism is well recognised (Giorgi & Métayer, 1990; Wand, 1999; Li *et al.*, 2007; Fluet *et al.*, 2008; Imam & Sarkar, 2014; Dolan *et al.*, 2017). We specifically focus on one-shot (Bruggeman *et al.*, 1996; Dolan *et al.*, 2017) delimited continuations. Each one-shot continuation can only be invoked at most once and hence corresponds to suspend/resume patterns. One-shot delimited continuations can express several concurrent/parallel programming constructs including coroutines (de Moura & Ierusalimschy, 2009) and Multilisp's futures (Imam & Sarkar, 2014).

It is non-trivial to integrate delimited continuations into the type system and operational semantics of λ^{AS}. Especially, the notion of *current continuation* is not well defined in parallel evaluations. For instance, when evaluating $\langle (\mathscr{S}k.\ k)\ (2 + 3) \rangle$, it is unclear which of $\lambda y.\ y\ (2 + 3)$ or $\lambda y.\ y\ 5$ is bound to k. Even worse, the result of $\langle (\mathscr{S}k.\ 1)\ (\mathscr{S}k.\ 3) \rangle$ can be either 1 or 3, depending on the evaluation order. To avoid such pathological cases, we strictly follow the suspend/resume patterns. Continuations are only used for suspending subcomputations, and suspended computations should be resumed later; neither duplication nor cancellation of suspended computations is allowed. In other words, we use delimited continuations only for controlling the evaluation strategy.

Shift/reset operators enable us to express a general method of splitting expressions for introducing parallelism. For an expression $e_1[e_2]$, the following transformation leads to a parallel evaluation of $e_1[\bullet]$ and e_2:[11]

$$e_1[e_2] \ \equiv\ \langle e_1[\mathscr{S}k.\ (\delta x.\ k\ x)\ e_2] \rangle$$

The context, e_1, is dynamically captured without making the variables in e_2 free. Applying this transformation to $(\lambda x.\ x + ((\lambda y.\ y + x)\ 3) + x)\ 5$ discussed above results in the following parallel evaluation:

[11] We assume that this rule introduces a corresponding shift/reset pair and their correspondence is kept. This is possible without extending the language. For example, a context up to the n-th nearest reset can be obtained by $\mathscr{S}k_1 \cdots \mathscr{S}k_n.\ \lambda x.\ k_n\ (k_{n-1}\ (\cdots (k_1\ x) \cdots))$.

https://doi.org/10.1017/S0956796821000058 Published online by Cambridge University Press

$$
\begin{array}{ll}
 & (\lambda x.\ x + ((\lambda y.\ y + x)\ 3) + x)\ 5 \\
\equiv & \langle(\lambda x.\ x + (\mathscr{S} k.\ (\delta x.\ k\ x)\ ((\lambda y.\ y + x)\ 3)) + x)\ 5\rangle \\
\to & \langle 5 + (\mathscr{S} k.\ (\delta z.\ k\ z)\ ((\lambda y.\ y + 5)\ 3)) + 5\rangle \\
\to & \langle((\delta z.\ (\lambda w.\ \langle 5 + w + 5\rangle)\ z)\ ((\lambda y.\ y + 5)\ 3)\rangle \\
\rightrightarrows & \langle(\lambda z.\ \langle 10 + z\rangle)\ 8\rangle \\
\to & 18
\end{array}
$$

Example 12 (list reductions, revisited)**.** So far, we have regarded lists as primitives and assumed that lists can be divided at a middle. For example, $sum\ (l +\!\!+ r) = sum\ l + sum\ r$ requires dividing the input list, $l +\!\!+ r$, into sublists l and r. One may hope to more formally express a list division by combining Church encoding and δ abstractions:

$$
\begin{array}{rcl}
[\,] & \equiv & \lambda c.\ \lambda n.\ n \\
(:) & \equiv & \lambda h.\ \lambda t.\ \lambda c.\ \lambda n.\ c\ h\ (t\ c\ n).
\end{array}
$$

However, we cannot divide Church-encoded lists by simply inserting a δ abstraction. For instance, a list $[a_0, a_1, a_2]$ is expressed by the following lambda expression:

$$\lambda c.\ \lambda n.\ c\ a_0\ ((\lambda c_1.\ \lambda n_1.\ c_1\ a_1\ ((\lambda c_2.\ \lambda n_2.\ c_2\ a_2\ ((\lambda c_3.\ \lambda n_3.\ n_3)\ c_2\ n_2))\ c_1\ n_1))\ c\ n).$$

Then, it appears to be impossible to insert a δ abstraction into this expression. For instance, the following is invalid because c_1 and n_1 become unbound:

$$\lambda c.\ \lambda n.\ (\delta x.\ c\ a_0\ ((\lambda c_1.\ \lambda n_1.\ c_1\ a_1\ x)\ c\ n))\ ((\lambda c_2.\ \lambda n_2.\ c_2\ a_2\ ((\lambda c_3.\ \lambda n_3.\ n_3)\ c_2\ n_2))\ c_1\ n_1)$$

Moreover, the following is also invalid because the type of δ-abstracted variable x has a function type:

$$\lambda c.\ \lambda n.\ (\delta x.\ c\ a_0\ ((\lambda c_1.\ \lambda n_1.\ c_1\ a_1\ (x\ c_1\ n_1))\ c\ n))\ (\lambda c_2.\ \lambda n_2.\ c_2\ a_2\ ((\lambda c_3.\ \lambda n_3.\ n_3)\ c_2\ n_2))$$

The general splitting rule using shift/reset operators resolves this problem:

$$
\begin{array}{ll}
 & (\lambda c.\ \lambda n.\ c\ a_0\ ((\lambda c_1.\ \lambda n_1.\ c_1\ a_1\ ((\lambda c_2.\ \lambda n_2.\ e)\ c_1\ n_1))\ c\ n))\ (+)\ 0 \\
 & \textbf{where}\ e = c_2\ a_2\ ((\lambda c_3.\ \lambda n_3.\ n_3)\ c_2\ n_2) \\
\equiv & (\lambda c.\ \lambda n.\ \langle c\ a_0\ ((\lambda c_1.\ \lambda n_1.\ c_1\ a_1\ ((\lambda c_2.\ \lambda n_2.\ e')\ c_1\ n_1))\ c\ n)\rangle)\ (+)\ 0 \\
 & \textbf{where}\ e' = \mathscr{S} k.\ (\delta x.\ k\ x)\ (c_2\ a_2\ ((\lambda c_3.\ \lambda n_3.\ n_3)\ c_2\ n_2)) \\
\to & \langle a_0 + (a_1 + (\mathscr{S} k.\ (\delta x.\ k\ x)\ (a_2 + ((\lambda c_3.\ \lambda n_3.\ n_3)\ (+)\ 0))))\rangle \\
\to & \langle((\delta x.\ (\lambda y.\ a_0 + (a_1 + y))\ x)\ (a_2 + ((\lambda c_3.\ \lambda n_3.\ n_3)\ (+)\ 0))\rangle
\end{array}
$$

As expected, in the last expression, the function and argument can be evaluated in parallel.

4.6 Efficiency

We have discussed the usefulness of the type system of λ^{AS} for understanding the overhead introduced by the simplification. However, what the type system exactly guarantees

https://doi.org/10.1017/S0956796821000058 Published online by Cambridge University Press

is not efficiency but the linearity of polynomials. The linearity rules out apparent inefficiency including exponential blow-up. Nevertheless, it does not guarantee that the parallel evaluation will improve efficiency.

First, the parallel evaluation may be useless because of the insufficient or ill-balanced independent tasks. 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:

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

Second, the speculative nature of the simplification in λ^{AS} may provoke computations that are unnecessary in the sequential evaluation. Recall the following example discussed in Section 4.3, in which the δ abstraction forces to evaluate $\bot$:

$$(\lambda x^{\tau}.\ 1)\ (\delta z^{\mathscr{R}}.\ \bot)$$

Third, the simplification of polynomials is slower than the usual evaluation. For instance, the parallel implementation of *poly* calculates two coefficients of a linear polynomial, and therefore, does about twice as much work as the sequential implementation. In general, if a calculated linear polynomial contains k indeterminates, its simplification is about $k + 1$ times as slow as the usual evaluation. This overhead is often essential for reduction parallelisation, as in the case of *poly*.

The number of indeterminates is commonly at most a constant, and hence, the overhead of simplifying polynomials does not affect asymptotic complexity. Note, however, a small program can be evaluated to a linear polynomial that contains many indeterminates. For example, consider the following reduction process:

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

The underlined linear polynomial consists of three indeterminates, y_0, y_1 and y_2, all of which originate from the same δ abstraction in the function g. We can obtain more indeterminates by composing more gs in the same manner: $g\ (g\ (\cdots (g\ (g\ (\lambda v.\ \lambda w.\ v + w)))\cdots))\ 1$. The type system of λ^{AS} does not reject such programs that involve many indeterminates.

In usual partial evaluations and simplifications, a result of evaluation/simplification can be large, and therefore, its duplication commonly introduces serious overheads. In λ^{AS}, however, a simplification only results in a linear polynomial whose size is bounded by the number of indeterminates. Therefore, if the number of indeterminates is at most constant, the cost of duplicating linear polynomials instead of constants does not affect asymptotic complexity.

https://doi.org/10.1017/S0956796821000058 Published online by Cambridge University Press

5 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 specialised for the static inputs without knowing the other inputs, called dynamic. On one hand, online 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) online 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 & Danvy (1992). While λ^{AS} makes use of algebraic properties and requires linearity over indeterminates to model efficient parallel reductions, their approach does not impose any requirements on programs and therefore provides no support for parallel programming, especially for developing efficient 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; Fisher & Ghuloum, 1994; Hu *et al.*, 1998; Chin *et al.*, 1998; Matsuzaki *et al.*, 2005; Morihata & Matsuzaki, 2010; Raychev *et al.*, 2015; Farzan & Nicolet, 2017; Jiang *et al.*, 2018; Farzan & Nicolet, 2019) and those on parallel querying of semi-structured databases (Buneman *et al.*, 2006; Cong *et al.*, 2007, 2012). They generally focus on specific reduction patterns to enable automation of reduction parallelisation. λ^{AS} is designed to be a foundation for exploring automatically parallelisable reduction patterns. As discussed in Section 2, the higher-order calculus enables us to express several reduction patterns and uniformly study their parallelisation. Instead of the generality, λ^{AS} makes less account of automatic parallelisation.

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 parallelisation system for list reductions. Their idea is to trace algebraic operators and the linearity condition using a type system. Matsuzaki *et al.* (2006) and Sato & Iwasaki (2011) developed similar systems for automatic parallelisation 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 uniformly study those parallelisation strategies. The basic idea of λ^{AS} is that their essence, that is, the use of algebraic properties for modelling complex reductions, is independent of the control structures that express iterations/recursions.

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.

https://doi.org/10.1017/S0956796821000058 Published online by Cambridge University Press

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 parallelisation 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 analyse 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, λ^{AS} can be used to parallelise programs whose divide-and-conquer implementations require breaking data dependencies using algebraic properties.

Nishimura & Ohori (1999) proposed a higher-order functional programming language that has a special construct, called a parallel map, for modelling 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.

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 (2012) used type-based semantic evaluations to perform computations on compressed data without decompression.

While λ^{AS} is based on call-by-value evaluation, δ abstraction introduces a different evaluation strategy. The call-by-push-value calculus (Levy, 2003) enables a close analysis of the effect of evaluation strategies by carefully distinguishing values and computations. Unfortunately, the call-by-push-value calculus seems insufficient for expressing λ^{AS}. In λ^{AS}, a δ abstraction generates a function, which is a *computation* in the call-by-push-value calculus, by capturing an indeterminate in a polynomial, which is a *value*. In the call-by-push-value calculus, the *thunk* construct can obtain a computation from a value; however, it cannot introduce a new binder that captures an indeterminate. Nevertheless, a similar calculus may be useful for providing a better understanding of λ^{AS}.

6 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

https://doi.org/10.1017/S0956796821000058 Published online by Cambridge University Press

using algebraic properties before its arguments' arrival. The operational semantics and type system of λ^{AS} were formalised. The type system guarantees that the simplification results in linear polynomials and, in turn, rules out the major possibility of unsuccessful parallelisation. The usefulness of λ^{AS} for modelling parallel reductions was demonstrated on several non-trivial examples.

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

6.1 Inferring evaluation costs

As discussed in Section 4.6, 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 non-linear polynomials as well because exponential blow-ups cannot occur.

6.2 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 may be useful: replace a λ abstraction to a δ abstraction if the introduction of δ abstraction improves the estimated parallel evaluation cost. 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.

6.3 Compilation to existing calculus

Although λ^{AS} is a theoretical model for studying reduction parallelisation, it is desirable to formulate a compilation of λ^{AS} to an existing calculus (or an abstract machine) that supports parallel evaluation. Such a compilation would lead to not only an understanding of λ^{AS} from a different perspective but also a better inference of parallel evaluation costs. However, providing such a compilation is non-trivial.

A major difficulty is the implementation of indeterminates. As discussed in Section 4.6, the evaluation of λ^{AS} may lead to unboundedly many free variables (indeterminates), which cannot be captured by usual variable environments. This situation has similarity to the case of lazy evaluations, which use a heap for managing unboundedly many thunks (Launchbury, 1993). However, a naive compilation using heaps is unsatisfactory because it threads computations and thereby prohibit exploiting parallelism.

https://doi.org/10.1017/S0956796821000058 Published online by Cambridge University Press

6.4 Parallelisation of practical programs

The original motivation in developing λ^{AS} is its application for reduction parallelisation 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.

Conflict of Interest

The author, Akimasa Morihata, is employed at the University of Tokyo. His recent collaborators are Kento Emoto (Kyushu Institute of Technology, Japan), Zhenjiang Hu (Peking University, China), Hideya Iwasaki (The University of Electro-Communications, Japan), Kiminori Matsuzaki (Kochi University of Technology, Japan), Shigeyuki Sato (The University of Tokyo, Japan) and Katsuhiro Ueno (Tohoku University, Japan).

Acknowledgement

The author is grateful to Makoto Hamana for his suggestion of related articles that were useful for proving Theorems 1 and 2, Sin'ya Katsumata for his introduction to related formalisms 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

Bergstrom, L., Fluet, M., Rainey, M., Reppy, J. H. & Shaw, A. (2012) Lazy tree splitting. *J. Funct. Program.* **22**(4–5), 382–438.

Blelloch, G. E. (1993) Prefix sums and their applications. In *Synthesis of Parallel Algorithms*, Chapter 1, Reif, J. H. (ed). Morgan Kaufmann Publishers.

Bruggeman, C., Waddell, O. & Dybvig, R. K. (1996) Representing control in the presence of one-shot continuations. In Proceedings of the ACM SIGPLAN'96 Conference on Programming Language Design and Implementation (PLDI), Philadephia, Pennsylvania, USA, May 21–24, 1996, Fischer, C. N. (ed). ACM, pp. 99–107.

Buneman, P., Cong, G., Fan, W. & Kementsietsidis, A. (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, Dayal, U., Whang, K.-Y., Lomet, D. B., Alonso, G., Lohman, G. M. , Kersten, M. L., Cha, S. K. & Kim, Y.-K. (eds). ACM, pp. 211–222.

Callahan, D. (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, Banerjee, U., Gelernter, D., Nicolau, A. & Padua, D. A. (eds), Lecture Notes in Computer Science, vol. 589. Springer, pp. 169–185.

Castro, D., Hammond, K. & Sarkar, S. (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, Garrigue, J., Keller, G. & Sumii, E. (eds). ACM, pp. 4–17.

Castro, D., Hammond, K., Sarkar, S. & Alguwaifli, Y. (2018) Automatically deriving cost models for structured parallel processes using hylomorphisms. *Future Gener. Comp. Syst.* **79**, 653–668.

https://doi.org/10.1017/S0956796821000058 Published online by Cambridge University Press

Chi, Y.-Y. & Mu, S.-C. (2011) Constructing list homomorphisms from proofs. In Programming Languages and Systems - 9th Asian Symposium, APLAS 2011, Kenting, Taiwan, December 5–7, 2011. Proceedings, Yang, H. (ed), Lecture Notes in Computer Science, vol. 7078. Springer, pp. 74–88.

Chin, W.-N., Takano, A. & Hu, Z. (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, pp. 153–162.

Cole, M. I. (1989) *Algorithmic Skeletons: Structural Management of Parallel Computation*. MIT Press.

Cong, G., Fan, W. & Kementsietsidis, A. (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, Chan, C. Y., Ooi, B. C. & Zhou, A. (eds). ACM, pp. 509–520.

Cong, G., Fan, W., Kementsietsidis, A., Li, J. & Liu, X. (2012) Partial evaluation for distributed XPath query processing and beyond. *ACM Trans. Database Syst.* **37**(4), 32:1–32:43.

Consel, C. & Danvy, O. (1992) Partial evaluation in parallel. *Lisp Symb. Comput.* **5**(4), 327–342.

Danvy, O. & Filinski, A. (1990) Abstracting control. In Proceedings of the 1990 ACM Conference on LISP and Functional Programming, LFP 1990, Nice, France, 27–29 June 1990. ACM, pp. 151–160.

Dean, J. & Ghemawat, S. (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, pp. 137–150.

de Moura, A. & Ierusalimschy, R. (2009) Revisiting coroutines. *ACM Trans. Program. Lang. Syst.* **31**(2), 6:1–6:31.

Deitz, S. J., Callahan, D., Chamberlain, B. L. & Snyder, L. (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, Torrellas, J. & Chatterjee, S. (eds). ACM, pp. 40–47.

Dolan, S., Eliopoulos, S., Hillerström, D., Madhavapeddy, A., Sivaramakrishnan, K. C. & White, L. (2017) Concurrent system programming with effect handlers. In Trends in Functional Programming - 18th International Symposium, TFP 2017, Canterbury, UK, June 19–21, 2017, Revised Selected Papers, Wang, M. & Owens, S. (eds), Lecture Notes in Computer Science, vol. 10788. Springer, pp. 98–117.

Emoto, K., Fischer, S. & Hu, Z. (2012) Filter-embedding semiring fusion for programming with mapreduce. *Formal Asp. Comput.* **24**(4–6), 623–645.

Emoto, K., Hu, Z., Kakehi, K., Matsuzaki, K. & Takeichi, M. (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, D'Ambra, P., Guarracino, M. R. & Talia, D. (eds), Lecture Notes in Computer Science, vol. 6272. Springer, pp. 26–37.

Farzan, A. & Nicolet, V. (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, Cohen, A. & Vechev, M. T. (eds). ACM, pp. 540–555.

Farzan, A. & Nicolet, V. (2019) Modular divide-and-conquer parallelization of nested loops. In Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22–26, 2019, McKinley, K. S. & Fisher, K. (eds). ACM, pp. 610–624.

Fedyukovich, G., Ahmad, M. B. S. & Bodík, R. (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, Cohen, A. & Vechev, M. T. (eds). ACM, pp. 572–585.

https://doi.org/10.1017/S0956796821000058 Published online by Cambridge University Press

Fisher, A. L. & Ghuloum, A. M. (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, pp. 135–146.

Fluet, M. & Pucella, R. (2006) Phantom types and subtyping. *J. Funct. Program.* **16**(6), 751–791.

Fluet, M., Rainey, M., Reppy, J. H. & Shaw, A. (2008) Implicitly threaded parallelism in Manticore. *J. Funct. Program.* **20**(5–6), 537–576.

Frigo, M., Halpern, P., Leiserson, C. E. & Lewin-Berlin, S. (2009) Reducers and other Cilk++ hyperobjects. In SPAA 2009: Proceedings of the 21st Annual ACM Symposium on Parallelism in Algorithms and Architectures, Calgary, Alberta, Canada, August 11–13, 2009, Meyer auf der Heide, F. & Bender, M. A., pp. 79–90.

Giorgi, J.-F. & Métayer, D. L. (1990) Continuation-based parallel implementation of functional programming languages. In Proceedings of the 1990 ACM Conference on LISP and Functional Programming, LFP 1990, Nice, France, 27–29 June 1990. ACM, pp. 209–217.

Gorlatch, S. (1999) Extracting and implementing list homomorphisms in parallel program development. *Sci. Comput. Program.* **33**(1), 1–27.

Henriksen, T., Serup, N. G. W., Elsman, M., Henglein, F. & Oancea, C. E. (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, Cohen, A. & Vechev, M. T. (eds). ACM, pp. 556–571.

Hu, Z., Iwasaki, H. & Takechi, M. (1997) Formal derivation of efficient parallel programs by construction of list homomorphisms. *ACM Trans. Program. Lang. Syst.* **19**(3), 444–461.

Hu, Z., Takeichi, M. & Chin, W.-N. (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, pp. 316–328.

Huet, G. P. (1997) The zipper. *J. Funct. Program.* **7**(5), 549–554.

Imam, S. M. & Sarkar, V. (2014) Cooperative scheduling of parallel tasks with general synchronization patterns. In ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28–August 1, 2014. Proceedings, Jones, R. E. (ed), Lecture Notes in Computer Science, vol. 8586. Springer, pp. 618–643.

Jiang, P., Chen, L. & Agrawal, G. (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, Evripidou, S., Stenström, P. & O'Boyle, M. F. P. (eds). ACM, pp. 10:1–10:13.

Jones, N. D. (1996) An introduction to partial evaluation. *ACM Comput. Surv.* **28**(3), 480–503.

Keller, G., Chakravarty, M. M. T., Leshchinskiy, R., Jones, S. L. P. & Lippmeier, B. (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, Hudak, P. & Weirich, S. (eds). ACM, pp. 261–272.

Kobayashi, N., Matsuda, K., Shinohara, A. & Yaguchi, K. (2012) Functional programs as compressed data. *Higher-Order Symb. Comput.* **25**(1), 39–84.

Ladner, R. E. & Fischer, M. J. (1980) Parallel prefix computation. *J. ACM* **27**(4), 831–838.

Launchbury, J. (1993) A natural semantics for lazy evaluation. In Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina, USA, January 1993, Deusen, M. S. V. & Lang, B. (eds). ACM, pp. 144–154.

Lévy, J.-J. (1976) An algebraic interpretation of the *lambda beta* k-calculus; and an application of a labelled *lambda*-calculus. *Theor. Comput. Sci.* **2**(1), 97–114.

Levy, P. B. (2003) *Call-by-Push-Value: A Functional/Imperative Synthesis*. Springer.

Li, P., Marlow, S., Peyton Jones, S. L. & Tolmach, A. P. (2007) Lightweight concurrency primitives for GHC. In Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2007, Freiburg, Germany, September 30, 2007, Keller, G. (ed). ACM, pp. 107–118.

https://doi.org/10.1017/S0956796821000058 Published online by Cambridge University Press

Marlow, S., Maier, P., Loidl, H.-W., Aswad, M. & Trinder, P. W. (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, Gibbons, J. (ed). ACM, pp. 91–102.

Matsuzaki, K., Hu, Z., Kakehi, K. & Takeichi, M. (2005) Systematic derivation of tree contraction algorithms. *Parallel Process. Lett.* **15**(3), 321–336.

Matsuzaki, K., Hu, Z. & Takeichi, M. (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, Gibbons, P. B. & Vishkin, U. (eds). ACM, pp. 39–48.

Minamide, Y. (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, MacQueen, D. B. & Cardelli, L. (eds). ACM, pp. 75–84.

Morihata, A. (2019) Lambda calculus with algebraic simplification for reduction parallelization by equational reasoning. *PACMPL* **3**(ICFP), 80:1–80:25.

Morihata, A. & Matsuzaki, K. (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, Blume, M., Kobayashi, N. & Vidal, G. (eds), Lecture Notes in Computer Science, vol. 6009. Springer, pp. 321–336.

Morihata, A. & Matsuzaki, K. (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, Chakravarty, M. M. T., Hu, Z. & Danvy, O. (eds). ACM, pp. 117–128.

Morihata, A., Matsuzaki, K., Hu, Z. & Takeichi, M. (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, pp. 177–185.

Morita, K., Morihata, A., Matsuzaki, K., Hu, Z. & Takeichi, M. (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, Ferrante, J. & McKinley, K. S. (eds). ACM, pp. 146–155.

Nishimura, S. & Ohori, A. (1999) Parallel functional programming on recursively defined data via data-parallel recursion. *J. Funct. Program.* **9**(4), 427–462.

Okada, M. (1989) Strong normalizability for the combined system of the typed lambda calculus and an arbitrary convergent term rewrite system. In Proceedings of the ACM-SIGSAM 1989 International Symposium on Symbolic and Algebraic Computation, ISSAC'89, Portland, Oregon, USA, July 17–19, 1989, Gonnet, G. H. (ed). ACM, pp. 357–363.

Raychev, V., Musuvathi, M. & Mytkowicz, T. (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, Miller, E. L. & Hand, S. (eds). ACM, pp. 153–167.

Reid-Miller, M., Miller, G. L. & Modugno, F. (1993) List ranking and parallel tree contraction. In *Synthesis of Parallel Algorithms*, Chapter 3, Reif, J. H. (ed). Morgan Kaufmann Publishers.

Sato, S. & Iwasaki, H. (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, Hall, M. W. & Padua, D. A. (eds). ACM, pp. 470–479.

Suganuma, T., Komatsu, H. & Nakatani, T. (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, pp. 18–25.

Tannen, V. (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, pp. 82–90.

https://doi.org/10.1017/S0956796821000058 Published online by Cambridge University Press

Tannen, V. & Gallier, J. H. (1991) Polymorphic rewriting conserves algebraic strong normalization. *Theor. Comput. Sci.* **83**(1), 3–28.

Terui, K. (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, Tiwari, A. (ed), LIPIcs, vol. 15. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, pp. 323–338.

Wand, M. (1999) Continuation-based multiprocessing. *Higher-Order Symb. Comput.* **12**(3), 285–299.

Xu, D. N., Khoo, S.-C. & Hu, Z. (2004) Ptype system: A featherweight parallelizability detector. In Programming Languages and Systems: Second Asian Symposium, APLAS 2004, Taipei, Taiwan, November 4–6, 2004. Proceedings, Chin, W.-N. (ed), Lecture Notes in Computer Science, vol. 3302. Springer, pp. 197–212.

https://doi.org/10.1017/S0956796821000058 Published online by Cambridge University Press