THE NATIONAL UNIVERSITY
of SINGAPORE

School *of* Computing
Lower Kent Ridge Road, Singapore 119260

TR30/03

A Type-Based Approach to Parallelization

Dana N. XU, Siau Cheng KHOO,
Wei Ngan CHIN and Zhenjiang HU

October 2003

Technical Report

Foreword

This technical report contains a research paper, development or tutorial article, which has been submitted for publication in a journal or for consideration by the commissioning organization. The report represents the ideas of its author, and should not be taken as the official views of the School or the University. Any discussion of the content of the report should be sent to the author, at the address shown on the cover.

JAFFAR, Joxan
Dean of School

A Type-Based Approach to Parallelization Technical Report

Dana N. Xu[1], Siau-Cheng Khoo[1], Wei-Ngan Chin[1], and Zhenjiang Hu[2,3]

School of Computing
National University of Singapore[1]
{xun,khoosc,chinwn}@comp.nus.edu.sg
University of Tokyo[2]
PRESTO 21, Japan Science and Technology Corporation[3]
hu@mist.i.u-tokyo.ac.jp

Abstract. Parallel functional programming plays an important role in parallel programming [16]. Type system has significant impact on program analysis [23]. In this paper, we show how to automatically and correctly synthesize parallel programs from sequential functional program based on the concept of a type system. Our type system captures the parallelizability of a program, in a modular fashion, by exploring the ring structures of the program's operators. It handles programs defined by self-recursive functions with accumulating parameters, as well as a limited form of non-linear mutual-recursive functions. In contrast to the Damas-Milner type system (the typical type system) that is constructed from the evaluation rules of the underlying language, our type system is constructed from a set of meta-rules that are used to transform sequential programs into a special normal form suitable for parallelization. The idea of this paper has been implemented and used to generate parallel code of a form, called *mutumorphism*, a general parallel computation model. Transforming into such a form is an important step towards constructing efficient data parallel programs.

1 Introduction

Many computational or data-intensive applications require performance level attainable only on parallel architectures. As multiprocessor systems have become increasingly available and their price/performance ratio continues to improve, interest has grown in parallel programming. While sequential programming is already a challenging task for programmers, parallel programming is much, much harder as there are many more things to consider, including available parallelism, task distribution, communication overheads, and debugging. A desirable approach for parallel program development is to start with a sequential program, test and debug the sequential program and then systematically transform the program to parallel counterpart.

However, systematic parallelization of sequential programs still remains a major challenge in parallel computing. Particularly challenging is the automatic

restructuring of programs which makes use of distributive and associative operators to obtain divide-and-conquer style parallelism.

A traditional approach to this problem is to identify a set of useful algorithmic skeletons with properties that allow parallelism to be harnessed. These skeletons are predefined higher-order functions such as *map*, *reduce*, etc. We call them *higher-order skeletons*. As an example, Blelloch's NESL language [3] supports two important parallel skeletons, namely *scan* and *segmented scan*, that together can cover a wide range of parallel programs.

A known problem with higher-order skeletons is that they are non-trivial for programmers to use, as they require the relevant properties, such as associativity, to be present in the combining (conquering) operators used. Often, this task is difficult as it may require the combining operators to return multiple results, including auxiliary computations that may not be present earlier.

For example, consider the polynomial function definition (in Haskell[20] syntax) :

$$\begin{array}{l} \mathit{poly}\ [a]\ c\ =\ a \\ \mathit{poly}\ (a : x)\ c\ =\ a + c \times (\mathit{poly}\ x\ c) \end{array}$$

To align it with higher order skeleton, we need to introduce a combining operator *comb*2. Thus, the new definition of *poly* is

$$\begin{array}{l} \mathit{poly}\ \mathit{xs}\ c\ =\ \mathit{fst}\ (\mathit{polytup}\ \mathit{xs}\ c) \\ \mathit{polytup}\ [a]\ c\ =\ (a, c) \\ \mathit{polytup}\ (a : x)\ c\ =\ (a, c)\ `\mathit{comb}2`\ (\mathit{polytup}\ x\ c) \\ \ \text{where}\ \mathit{comb}2\ (p_1, u_1)\ (p_2, u_2)\ = (p_1 + p_2 \times\ u_1,\ u_2 \times\ u_1) \end{array}$$

Note that *comb*2 has to compute two values, including an auxiliary result, in order to be associative. Using it, we are able to match the above definition to suitable higher-order skeletons, as shown below. In general, coming up with suitable associative combining forms can be challenging for parallel programmers.

$$\mathit{poly}\ \mathit{xs}\ c\ =\ \mathit{fst}\ (\mathit{reduce}\ \mathit{comb}2\ (\mathit{map}\ (\backslash\ x\ \rightarrow\ (x, c))\ \mathit{xs}))$$

In this paper, we show that parallelization of sequential code can be automated to a great extent through automatic program analysis and transformation. We view parallelization as a meta-level transformation from sequential programs to parallel programs. As there is a big difference in the control structure of these two kinds of programs, we perform parallelization in two phases:(1) transforming parallelizable sequential program to a special form (which is still a sequential program), and (2) providing a direct mapping from this special form to parallel program. We name this special form *skeleton value* (defined in Section 2.3). The challenge of this parallelization process is to provide an automatic mechanism to the first phase, which is the focus of this paper. (Details of the second phase can be found in [31].)

Transformation to skeleton values are defined by a set of *normalization rules*, and parallelizability of a program is identified with its normalizability. A type

system is designed to detect parallelizability of a program. We call this new type system the PType system, where PType denotes *Parallelizable Type*. Operationally, the PType system can accept any first-order functional programs. If a recursive function defined over a group operators exhibit a ring structure (defined in Section 2.1), it is well-PTyped and can be normalized to an s-value, which can be automatically converted to parallel code. Otherwise, the function definition will remain as it is.

Underlying the PType system is a new way of reasoning about expressions, not according to the usual semantic evaluation rules, but rather in accordance with our normalization rules. This is distinguished from the conventional type-directed compilation approach to program transformation [29], in which the correctness of types is shown with respect to the original program semantics.

Our approach is complementary to the traditional higher-order skeleton approach, and can be used to enhance the usability of the latter. Specifically, traditional approach encourages users to write *non*-recursive functions using higher-order skeletons, while we allow the parallelization of functions defined in their natural recursive setting.

In the case of function *poly* defined earlier, our PType system infers that the expression $(a + c \times (f\ x))$ has the type $R_{[+,\times]}$. Here, $+$ and $\times$ in $R_{[+,\times]}$ are the operators that enable context preservation through their associativity and distributivity properties. More specifically, the present PType system aims to discover a set of binary operators, within an expression, that obeys an *extended-ring property* (defined in Section 2.1). Such discovery guarantees the parallelization of the sequential programs.

As another example, consider the following variant function,

$$f_1\ [(a, y)]\ c = a + c \times y$$
$$f_1\ ((a, y) : x)\ c = a + (c \times (y + (f_1\ x\ c)))$$

Even though the variant appears to be quite different from the *poly* definition, the PType system is still able to classify f_1 as having the same PType as that for *poly*, namely, $R_{[+,\times]}$ yielding a similar parallel code as *poly*. Parallel codes for both *poly* and f_1 are shown in Figure 1. The only difference between their pair of parallel code lies in their base cases. Note that parallel codes produced by our system can be automatically transformed to more efficient codes in mutumorphism form [13], and even to efficient homomorphisms (i.e. divide-and-conquer algorithm), by the tupling calculation [5, 18, 17].

Our type-based approach to parallelization provides a high-level user interface to programmers. It frees them from the operational detail (such as context preservation testing, normalization and even the concept of the type system) and enables them to focus on the (extended ring) properties of the operators involved.

The main contributions of this paper are as follows:

1. We propose a novel type system for parallelization and prove its soundness. We believe this is the first work on capturing parallelization in type format.

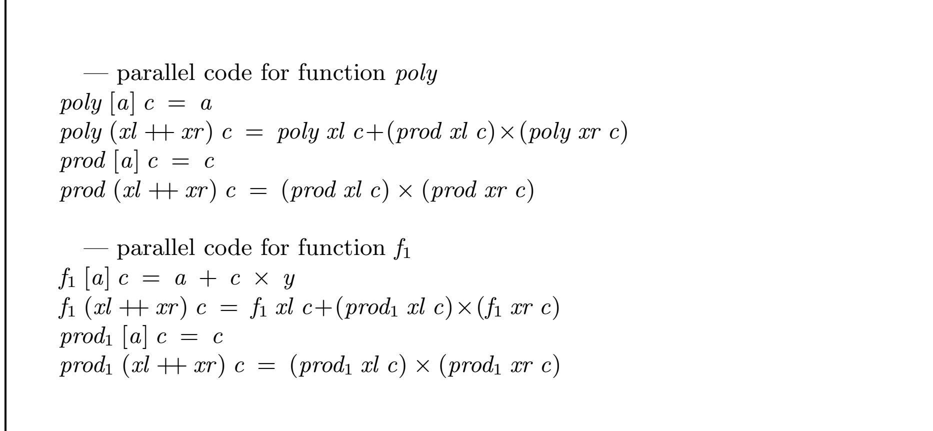

Fig. 1. Parallel Codes for *poly* and f_1

2. We propose a systematic and automatable normalization process for turning sequential programs to a form susceptible to parallelization. Our process relies on a general property about a group of binary operators, called extended ring property.
3. We construct a type (`PType`) system at the meta level, over a set of program transformation rules. This approach can be distinguished from the conventional type-directed compilation techniques, but akin to the approach taken by Cousot & Cousot in reasoning program transformation through abstract interpretation [10].
4. We provide an inference algorithm to reconstruct the parallelizability of functions. Furthermore, we automatically derive its parallel counterpart.

Our `PType` system handles first-order functional programs. It is amenable to recursively-defined functions with accumulating parameters and non-linear recursion. For the clarity of presentation, we illustrate our system without these two features in the main section and discuss them separately in the later section.

The outline for the paper is as follows. In the next section, we describe the syntax of the language used in the paper and give an overview of the `PType` system. Section 3 gives a set of typing rules and provides a corresponding inference algorithm. We illustrate the working of the `PType` system through examples in Section 5. Section 6 describes two important extensions (namely, recursive functions with accumulating parameters and non-linear recursive functions) to our `PType` system. Section 7 gives experiment results. Related works are discussed in Section 8. Finally, we conclude the paper in Section 9.

2 Overview

Analogous to Damas-Milner (DM) type system [11], our `PType` system asserts some properties about the subject program – their parallelizability. However,

this property is *not* directly related to the program's underlying semantics. It is possible for two function definitions with the same denotational semantics (e.g. different sorting algorithm) to exhibit different `PType`s in our type system, as one may be parallelizable but not the other.

In order to correctly reason about the `PType` of a program, we depart from the usual practice of type construction, and define a program's `PType` from a set of normalization rules, instead of from a set of evaluation rules. Under these normalization rules, a term in a program may be normalized to a special value, which we call *skeletal value*, or *s-value*. This s-value enables automatic derivation of parallel code for the original term. The goal of the `PType` system is thus to identify as many terms as possible that can be normalized to *s-values*.

Table 1 gives an analogy between the DM type system (which is at object level) and the `PType` system (which is at meta level). Suppose the recursive part

Table 1. Object Level vs. Meta Level

	DM Type System	`PType` System
Subject program	functions	functions
Basic Values	semantic value	s-value
Reduction Rules	evaluation rules	normalization rules
Type	data type	parallelizability

of a sequential function definition is $f\ (a : x)\ =\ e$ where e involves recursive call $(f\ x)$. The well-typedness of e asserts that e can be normalized to an s-value. Consequently, the subject-reduction property of `PType` system is proven with respect to the set of normalization rules.

2.1 Context Preservation

In [8], a program restructuring technique, known as *context preservation* was introduced to determine if parallelization is feasible. The term "context" here refers to a contextual expression where the recursive sub-terms have been extracted.

Consider the polynomial function definition again. As context preservation is done primarily for the recursive equation of *poly*:

$$poly\ (a : x)\ c = a + c \times (poly\ x\ c)$$

the *contextual function* (which will be called "*context*" for the rest of this paper) which extracts away the recursive subterm of the RHS of this equation can be written as $\lambda(\underline{\bullet})\ .\ \alpha\ +\ \beta\ \times\ (\underline{\bullet})$. Here, the symbol $\underline{\bullet}$ denotes an occurrence of a self-recursive call, while α and β denote subterms that do not contain any recursive call. Such a context is said to be *context preserving modulo replication* (or *context preservation* for short) if after composing the context with itself, we

can still obtain (by transformation) a resulting context that has the same form as the original context. For the case of function *poly*, we compose the context with a renamed copy of itself, as follows:

$$(\lambda\,(\underline{\bullet})\ .\ \alpha_1\ +\ \beta_1\ \times(\underline{\bullet}))\ \circ\ (\lambda\,(\underline{\bullet})\ .\ \alpha_2\ +\ \beta_2\ \times\ (\underline{\bullet}))$$

This composition is simplified through a sequence of transformation steps. If the simplified form matches the original context, we will have achieved context preservation, as illustrated below.

$(\lambda\,(\underline{\bullet})\ .\ \alpha_1\ +\ \beta_1\ \times(\underline{\bullet}))\ \circ\ (\lambda(\underline{\bullet})\ .\ \alpha_2\ +\ \beta_2\ \times\ (\underline{\bullet}))$
— function composition
$=\ \lambda\,(\underline{\bullet})\ .\ \alpha_1\ +\ \beta_1\ \times(\alpha_2\ +\ \beta_2\ \times\ (\underline{\bullet}))$
— $\times$ is distributive over $+$
$=\lambda\,(\underline{\bullet})\ .\ \alpha_1\ +\ (\beta_1\ \times\alpha_2\ +\ \beta_1\ \times\ (\beta_2\ \times\ (\underline{\bullet})))$
— $+$, $\times$ being associative
$=\ \lambda\,(\underline{\bullet})\ .\ (\alpha_1\ +\ \beta_1\ \times\alpha_2)\ +\ (\beta_1\ \times\ \beta_2)\ \times(\underline{\bullet})$
— it matches the original form
— as we can write it in the following form
$=\ \lambda\,(\underline{\bullet})\ .\ \alpha\ +\ \beta\ \times(\underline{\bullet})$
— where $\alpha = \alpha_1 + \beta_1 \times \alpha_2$ and $\beta = \beta_1 \times \beta_2$

This transformation process is called *normalization*. Currently, the normalization process described in [6] is ad-hoc, and relied on some heuristics related to the associativity and distributivity of binary operators involved.

Our first contribution in this paper is to introduce an *extended ring property* of the operators which guarantees automatic detection of context preservation.

Definition 1. *Let $S = [\oplus_1, \ldots, \oplus_n]$ be a sequence of n binary operators. We say that S possesses an* extended-ring property *iff*

1. *all operators are semi-associative;*
2. *each operator $\oplus$ has an identity, denoted by $\iota_\oplus$;*
3. *$\oplus_j$ is distributive over $\oplus_i$ $\forall\, 1 \leq\ i\ <\ j\ \leq\ n$.*

The *semi-associative* law states that $e_1\ \oplus\ (e_2\ \oplus\ e_3)\ =\ (e_1\ \oplus'\ e_2)\ \oplus\ e_3$ where $\oplus'$ is the *associative dual* of $\oplus$. Note that associativity is a special case of semi-associativity whereby $\oplus' = \oplus$. Furthermore, the identity of each operator $\oplus$ satisfies: for all possible operand v, we have: $\iota_\oplus\ \oplus\ v = v\ \oplus\ \iota_\oplus = v$.

For example, in the domain of non-negative integers, operators *max*, $+$ and $\times$ in that order form an extended ring. Their identities are 0, 0 and 1 respectively.

2.2 Language Syntax

We apply our technique to a first-order typed functional language with strict semantics. The syntax of our source language is given in Figure 2. We require programmers to annotate properties of binary operators used in the program. For example, annotation $\#(Int, [+, \times], [0, 1])$ is needed for the function definition *poly*. The annotation tells the system that, for all integers, operators $+$ and $\times$

satisfy the extended-ring property with 0 and 1 as their respective identities. (Note: annotations for system-defined operators are pre-stored in a library. Only user-defined operators' properties are required to be annotated by programmers in the implemented system.)

$$
\begin{array}{rcll}
\tau & \in & \texttt{Typ} & \textbf{Type} \\
n & \in & \texttt{Cons} & \textbf{Constants} \\
c & \in & \texttt{Con} & \textbf{Data Constructors} \\
v & \in & \texttt{Var} & \textbf{Variables} \\
\oplus & \in & \texttt{Op} & \textbf{Binary Primitive Operators} \\
\gamma & \in & \texttt{Ann} & \textbf{Annotations} \\
\gamma & ::= & \multicolumn{2}{l}{\#(\tau, [\oplus_1, \ldots, \oplus_n], [\iota_{\oplus_1}, \ldots, \iota_{\oplus_n}])} \\
e, t & \in & \texttt{Exp} & \textbf{Expressions} \\
e, t & ::= & \multicolumn{2}{l}{n \mid v \mid c\, e_1 \ \ldots\ e_n \mid e_1 \oplus e_2} \\
& & \multicolumn{2}{l}{\mid f\, e_0\, e_1 \ \ldots\ e_n \mid \ \textbf{let}\, v = e_1 \ \textbf{in}\, e_2} \\
& & \multicolumn{2}{l}{\mid \ \textbf{if}\ e_0 \ \textbf{then}\, e_1 \ \textbf{else}\, e_2} \\
p & \in & \texttt{Pat} & \textbf{Pattern} \\
p & ::= & \multicolumn{2}{l}{v \mid c\, v_1 \ \ldots\ v_n} \\
\sigma & \in & \texttt{Prog} & \textbf{Programs} \\
\sigma & ::= & \multicolumn{2}{l}{\gamma_i^*, (f_i\, p_0\, p_1 \ \ldots\ p_n = e)^* \ \forall\, i.\, i \geq 1} \\
& & \multicolumn{2}{l}{\text{where } f_1 \text{ is the main function.}}
\end{array}
$$

Fig. 2. Syntax of the source language.

Function definitions in this paper are written in Haskell syntax [20]. For the rest of the paper, we shall discuss detection of parallelism for recursive functions of the form

$$f(a : x) = E[\langle t_i \rangle_{i=1}^{m}, \langle q_j\ x \rangle_{j=1}^{n}, \langle f x \rangle]$$

where f is inductively defined on a list. This form was first described in [19]. $E[\,]$ denotes an *expression context* with three groups of holes $\langle\rangle$. It itself contains no occurrence of references to a, x and f. $\langle t_i \rangle_{i=1}^{m}$ is a group of m terms, each of which is allowed to contain occurrences of a, but not those of references to $(f\ x)$. $\langle q_i\ x \rangle_{j=1}^{n}$ denotes a group of n function applications, each of which is a mutumorphism (*aka.*, parallelized function, *c.f.* [19]). Lastly, $\langle f\ x \rangle$ is the self-recursive call.

As our analysis focuses on the syntactic expressions consisting of recursive calls, all variables directly or indirectly denoting an expression consisting of recursive call(s) need to be traced. We call such variable a *reference to recursive call* whose detailed definition is given below.

Definition 2 (Reference to Recursive Call). *A variable v is said to be a reference to recursive call(s) if the evaluation of v leads to an invocation of a recursive call.*

Consider the following function definitions

$$\begin{aligned} f_2\ (a : x) &= \textbf{let}\ v_2\ =\ 1\ +\ f_2\ x\ \textbf{in}\ a\ +\ v_2 \\ f_3\ (a : x) &= \textbf{let}\ v_3\ =\ 1\ +\ f_3\ x \\ &\quad\ \textbf{in}\ \textbf{let}\ u\ =\ 2\ +\ v_3\ \textbf{in}\ a\ +\ u, \end{aligned}$$

variable v_2 is a reference to the recursive call $(f_2\ x)$ as it names an expression which encloses a recursive call. In f_3, variables u and v_3 are references to recursive call. Variable u indirectly depends on the recursive call since it contains v_3.

For ease of presentation, we consider the following simplifications to our language that can be overcome in our full implementation.

- Each recursive function has only one recursion parameter located at position p_0. (We shall discuss how to deal with multiple recursion parameters and accumulating parameters in Section 6.1 and Section 6.2 respectively).
- There is only one occurrence of parallelizable auxiliary function, denoted by $(q\ x)$. (Users are allowed to use $(q_1 x), \ldots, (q_n\ x)$ in their program because we can *tuple* $(q_1 x), \ldots, (q_n\ x)$ to obtain a single $(q\ x)$ with the technique in [5, 18] as they are known to be homomorphism before normalization.)
- The recursive call $(f\ x)$, or its references, only appears in the right operand of any associative operator. (If it appears in the left operand, symmetric typing/normalization rules can be easily introduced.)
- The recursive call $(f\ x)$, or its references, does not occur in the test of **if** expression. (As context preservation of such functions require complex use of invariants [7] that are not presently captured by our system.)
- Recursive functions are all *linear self-recursive*. (We show how to parallelized non-linear recursive function in Section 6.3).

A function definition is said to be *linear self-recursive* if every execution path represented in its RHS expression contains references to at most one self-recursive call.

For example, the function f_4 is linear self-recursive:

$$\begin{aligned} f_4\ [\,] &= 0 \\ f_4\ (a : x) &= \textbf{if}\ a \leq\ 0\ \textbf{then}\ f_4\ x\ \textbf{else}\ a\ +\ (f_4\ x) \end{aligned}$$

whereas the function f_5 is not, because it will invoke two references to $(f_5\ x)$ during the execution of the **let**-body:

$$\begin{aligned} f_5\ [\,] &= 0 \\ f_5\ (a : x) &= \textbf{let}\ v\ =\ a\ +\ (f_5\ x)\ \ \textbf{in}\ v\ +\ v \end{aligned}$$

2.3 Skeletal Values

S-values (defined in Figure 3) are possible (also desirable) final results of normalization as they can be directly translated into parallel codes. S-values belong to a class of expressions conforming to a fix set of patterns which are parallelizable. In the figure, we use $\underline{\bullet}$ (the same as in Section 2.1) to denote a self-recursive call in a function definition.

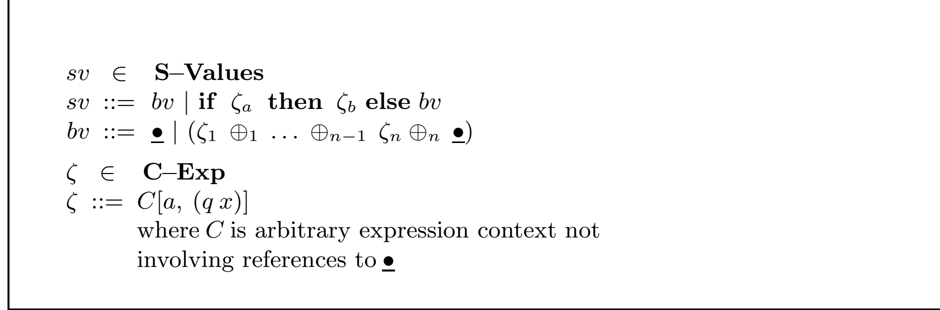

Fig. 3. Skeletal Values

An s-value of the form $(\zeta_1 \oplus_1 \ldots \oplus_{n-1} \zeta_n \oplus_n \underline{\bullet})$[1] is said to be *composed directly by* the sequence of operators $[\oplus_1, \ldots, \oplus_n]$ with extended-ring property (defined in Section **??**). An s-value of the form **if** ζ_0 **then** ζ_1 **else** lv is said to be in *conditional form*. Its self-recursive call occurs *only* in its alternate branch.

From [8, 19], we know that given the recursive part of a function body e, if e is context preserved, the function is parallelizable. In the following lemma, we show that all s-values are context preserved. Consequently, any expression e that can be normalized to an s-value is context preserved.

Lemma 1 (S-Values are Context Preserved).
Given a recursive part of a function definition $f\ (a : x) = e$, if e is an s-value, then e can be context preserved.

Proof. The proof is based on the definition of context preservation described in [8, 19]; *ie.*, an expression $e = E[\langle\, t_i \rangle_{i=1}^{m}, \langle\, q\ x \rangle, \underline{\bullet}]$ is context preserved if the following holds:

$$E[\langle\, A_i \rangle_{i=1}^{m}, \langle q\ (y +\!\!+ x) \rangle,\ E[\langle\, B_i \rangle_{i=1}^{m}, \langle q\ x \rangle, \underline{\bullet}]] \ = E[\langle\, t_i' \rangle_{i=1}^{m}, \langle\, q\ x \rangle, \underline{\bullet}].$$

It is easy to check that any recursive function with an s-value as its RHS can also be expressed in the form:

$$f(a : x) = E[\langle t_i \rangle_{i=1}^{m}, \langle q_j\ x \rangle_{j=1}^{n}, \langle f x \rangle]$$

[1] It is equivalent to $(\zeta_1 \oplus_1 (\cdots \oplus_{n-1} (\zeta_n \oplus_n \underline{\bullet}) \ldots))$. We omit brackets in the expression for simplicity.

Note: we let $C_i[a, (q\ x)] = t_i\ (q\ x)$ and $t_i = \lambda\, z.((g_i\ a)\ z)\ \forall\, 1 \le i \le n$ for arbitrary g_i. Operator $\oplus_q$ refers to the first operator of the sequence S in R_S where R_S is the `PType` of the function q.

We prove the lemma by analysing the structure of s-values. The first two cases prove that bv is context preserved. Case 3a and 3b prove that conditional form is context preserved when lv is bv. Case 4 proves that local abstraction form is context preserved. Case 5 proves that conditinal form is context preserved when lv is in local abstract form.

Case 1: $e = \underline{\bullet}$ It is vicciously true.

Case 2: $e = (t_1\ (q\ x)\ \oplus_1\ (t_2\ (q\ x) \oplus_2\ \ldots\ \oplus_{n-1}\ (t_n\ (q\ x) \oplus_n \underline{\bullet})))$

This context is preserved since

$$
\begin{aligned}
& E[\langle\, A_i \rangle_{i=1}^n, \langle\, q\ (y +\!\!+\ x) \rangle\rangle,\ E[\langle B_i \rangle_{i=1}^n, \langle\, q\ \ x \rangle, \underline{\bullet}]] \\
& = E[\langle\, t_i' \rangle_{i=1}^n, \langle\, q\ \ x \rangle, \underline{\bullet}] \\
& \text{where} \\
& \forall\, 1 \le\ i \le\ n. \\
& t_i' = \lambda\, z.\, t_i\ (q\ y\ \oplus_q\ z) \oplus_i\ \ldots\ \oplus_{n-1}\ t_n\ (q\ y\ \oplus_q\ z)\ \oplus\ t_i\ z
\end{aligned}
$$

Case 3a: $e = \textbf{if}\ t_1\ (q\ x)\ \textbf{then}\ \ t_2\ (q\ x)\ \textbf{else}\ \underline{\bullet}$

This context is preserved since

$$
\begin{aligned}
& E[\langle\, A_i \rangle_{i=1}^2, \langle\, q\ (y +\!\!+\ x) \rangle\rangle,\ E[\langle B_i \rangle_{i=1}^2, \langle\, q\ \ x \rangle, \underline{\bullet}]] \\
& = E[\langle\, t_i' \rangle_{i=1}^2, \langle q\ \ x \rangle, \underline{\bullet}] \\
& \text{where} \\
& t_1' = \lambda\, z.\, A_1\ (q\ \oplus_q\ \ z)\ \vee\ B_1\ z \\
& t_2' = \lambda\, z.\, \textbf{if}\ A_1\ (q\ \oplus_q\ \ z)\ \textbf{then}\ A_2\ (q\ \oplus_q\ \ z)\ \textbf{else}\ B_2\ z
\end{aligned}
$$

Case 3b: $e = \textbf{if}\ \ t_{n+1}\ (q\ x)\ \textbf{then}\ \ t_{n+2}\ (q\ x)$
$\qquad\qquad \textbf{else}\ t_1\ (q\ x)\ \oplus_1 \ldots\ \oplus_{n-1}\ \ t_n\ (q\ x) \oplus_n\ \underline{\bullet}$

For simplicity of presentation, we only prove context
$E[\langle\, t_i \rangle_{i=1}^4, (q\ x), \underline{\bullet}] :: R_{[\oplus_1, \oplus_2]} =$
$\textbf{if}\ \ t_3\ (q\ x)\ \ \textbf{then}\ \ t_4\ (q\ x)\ \textbf{else}\ t_1\ (q\ x)\ \oplus_1\ \ t_2\ (q\ x)\ \oplus_2 \underline{\bullet}$

It is not difficult to generalize the proof to get a complete proof.

$$
\begin{aligned}
& \quad \text{— checking context preservation} \\
& \textbf{if}\ \ A_3\ (q\ (y +\!\!+\ x))\ \textbf{then}\ \ A_4\ (q\ (y +\!\!+\ x)) \\
& \textbf{else}\ A_1\ (q\ (y +\!\!+\ x))\ \oplus_1\ \ A_2\ (q\ (y +\!\!+\ x)) \oplus_2 \\
& \qquad (\textbf{if}\ \ B_3\ (q\ x)\ \textbf{then}\ \ B_4\ (q\ x) \\
& \qquad\ \ \textbf{else}\ \ B_1\ (q\ x)\ \oplus_1\ \ B_2\ (q\ x)\ \oplus_2\ \underline{\bullet}) \\
& \quad \text{— lifting } \textbf{if} \text{ out} \\
& = \textbf{if}\ \ A_3\ (q\ (y +\!\!+\ x))\ \textbf{then}\ \ A_4\ (q\ (y +\!\!+\ x)) \\
& \quad \textbf{else}\ \ (\textbf{if}\ \ B_3\ (q\ x) \\
& \qquad\quad \textbf{then}\ \ A_1\ (q\ (y +\!\!+\ x)\ \oplus_1\ \ A_2\ (q\ (y +\!\!+\ x)) \\
& \qquad\qquad\quad \oplus_2\ \ B_4\ (q\ x) \\
& \qquad\quad \textbf{else}\ A_1\ (q\ (y +\!\!+\ x))\ \oplus_1\ \ A_2\ (q\ (y +\!\!+\ x)) \\
& \qquad\qquad\quad \oplus_2\ B_1\ (q\ x)\ \oplus_1\ \ B_2\ (q\ x)\ \oplus_2\ \underline{\bullet}) \\
& \quad \text{— merging two } \textbf{if}\text{s}
\end{aligned}
$$

$$
\begin{aligned}
= & \ \mathbf{if}\ (A_3\ (q\ (y +\!\!+\ x)) \vee\ B_3\ (q\ x)) \\
& \ \mathbf{then}\ \ (\mathbf{if}\ (A_3\ (q\ (y +\!\!+\ x))\mathbf{then}\ A_4\ (q\ (y +\!\!+\ x)) \\
& \qquad\quad \mathbf{else}\ A_1\ (q\ (y +\!\!+\ x))\ \oplus_1\ A_2\ (q\ (y +\!\!+\ x)) \\
& \qquad\qquad\quad \oplus_2\ B_4\ (q\ x) \\
& \ \mathbf{else}\ A_1\ (q\ (y +\!\!+\ x))\ \oplus_1\ A_2\ (q\ (y +\!\!+\ x))\ \oplus_2\ B_1\ (q\ x) \\
& \qquad \oplus_1\ B_2\ (q\ x)\ \oplus_2\ \underline{\bullet}) \\
= & \ \mathbf{if}\ (\lambda\, z.\ A_3\ (q\ y\ \oplus_q\ z)) \vee\ B_3\ z)\ (q\ x)) \\
& \ \mathbf{then}\ \ (\lambda\, z.\ \mathbf{if}\ (A_3\ (q\ y \oplus_q\ z))\ \mathbf{then}\ \ A_4\ (q\ y \oplus_q\ z) \\
& \qquad\quad \mathbf{else}\ A_1\ (q\ y \oplus_q\ z)\ \oplus_1\ A_2\ (q\ y \oplus_q\ z) \\
& \qquad\qquad\quad \oplus_2\ B_4\ z)\ (q\ x) \\
& \ \mathbf{else}\ (\lambda\, z.\ A_1\ (q\ y \oplus_q\ z))\ \oplus_1\ A_2\ (q\ y \oplus_q\ z))\ \oplus_2\ B_1\ z \\
& \qquad \oplus_1\ B_2\ z)\ (q\ x)\ \oplus_2\ \underline{\bullet}
\end{aligned}
$$

□

2.4 Normalization Rules and Normal Forms

Figure 4 defines some normalization rules on expressions. They preserve the denotational semantics of those expressions of which the standard evaluation terminates.

The relation $e \rightsquigarrow e'$ defines a one-step normalization of expression e to e'. Its transitive closure defines a normalization process. The symbol sv represents an s-value, whereas bv and lv represent s-values of syntactic categories bv and lv respectively.

All normalization rules observe the following property: *an expression may be subject to normalization only when it encompasses some self-recursive calls, denoted by* $\underline{\bullet}$. Application of the normalization rules is deterministic.

When an expression e cannot be normalized by any of the rules, we say e is in *normal form*, and denote it by "$e \not\rightsquigarrow$".

The rules in Figure 4 deal with binary operations and **let** expressions. We assume that all binary operators are strict on both arguments. In `N-op`, we assume that the self-recursive call only appears in the right operand, and attempt to normalize the right operand first. Extending the normalization to handle calls appearing in either operand can be easily handled by introducing symmetrical normalization rules. `N-distr` and `N-semiassoc` try to normalize the entire binary operation *after* the right operand has been normalized to an s-value composed directly by a sequence of binary operations. `N-liftIf` lifts **if** to the top of the expression. The normalization is correct with respect to the strict semantics. `N-sv` and `N-grpN` ensure that the self-recursive call occurs only in the alternate branch of the top-level conditional. `N-let` attempts to normalize e_1 to an s-value (if e_1 contains recursive call) while `N-in` attempts to normalize e_2 to an s-value before v in e_2 is substituted with e_1. `N-sub` is applied when both e_1 and e_2 are in normal form. Such unfolding of local definition may cause code duplication, and it can be compensated by a common-subexpression elimination phase after the parallelization transformation, in order to maintain efficiency.

The rules in Figure 5 and 6 handle normalization of conditionals. `N-else` and `N-then` attempt to normalize the branches of the conditional. `N-grpR1`,

$$\frac{e \leadsto e'}{\zeta \oplus_i e \leadsto \zeta \oplus_i e'} \quad (\mathtt{N-op})$$

$$\frac{bv = \zeta_2 \oplus_i bv_1 \quad i < j}{\zeta_1 \oplus_j bv \leadsto (\zeta_1 \oplus_j \zeta_2) \oplus_i (\zeta_1 \oplus_j bv_1)} \quad (\mathtt{N-distr})$$

$$\frac{bv = \zeta_2 \oplus bv_1}{\zeta_1 \oplus bv \leadsto (\zeta_1 \oplus' \zeta_2) \oplus bv_1} \quad (\mathtt{N-semiassoc})$$

$$\zeta_1 \oplus \textbf{if } \zeta_0 \textbf{ then } \zeta_2 \textbf{ else } bv \leadsto \textbf{if } \zeta_0 \textbf{ then } \zeta_1 \oplus \zeta_2 \textbf{ else } \zeta_1 \oplus bv \quad (\mathtt{N-liftIf})$$

$$\textbf{if } \zeta_0 \textbf{ then } sv \textbf{ else } \zeta_2 \leadsto \textbf{if } \neg \zeta_0 \textbf{ then } \zeta_2 \textbf{ else } sv \quad (\mathtt{N-sv})$$

$$\begin{array}{l}\textbf{if } \zeta_0 \textbf{ then } \zeta_1 \\ \quad \textbf{else } (\textbf{if } \zeta_2 \textbf{ then } \zeta_3 \textbf{ else } bv)\end{array} \leadsto \begin{array}{l}\textbf{if } (\zeta_0 \vee \zeta_2) \\ \quad \textbf{then } (\textbf{if } \zeta_0 \textbf{ then } \zeta_1 \textbf{ else } \zeta_3) \\ \quad \textbf{else } bv\end{array} \quad (\mathtt{N-grpN})$$

$$\frac{e_1 \leadsto e_1'}{\textbf{let } v = e_1 \textbf{ in } e_2 \leadsto \textbf{let } v = e_1' \textbf{ in } e_2} \quad (\mathtt{N-let})$$

$$\frac{e_1 \not\leadsto \quad e_2 \leadsto e_2'}{\textbf{let } v = e_1 \textbf{ in } e_2 \leadsto \textbf{let } v = e_1 \textbf{ in } e_2'} \quad (\mathtt{N-in})$$

$$\frac{e_1 \not\leadsto \quad e_2 \not\leadsto}{\textbf{let } v = e_1 \textbf{ in } e_2 \leadsto e_2[v \mapsto e_1]} \quad (\mathtt{N-sub})$$

Fig. 4. Normalization Rules I

`N-grpR2` and `N-grpR3` consider three cases of nested conditionals in which the self-recursive calls appear in more than one branches. They aim to contain these calls within the alternate branch of the top-level conditional. `N-rmTest` eliminates redundant conditional test, whereas `N-pushIf1` to `N-pushIf5` transform "conditionals of binary operations" into "binary operations over conditionals". This is accomplished by introducing identity elements of the respective binary operators.

Property 1 (Normal Form). All s-values are in normal form.

$$\frac{e_1 \rightsquigarrow e_1' \quad e_2 \not\rightsquigarrow}{\textbf{if } \zeta \textbf{ then } e_1 \textbf{ else } e_2 \rightsquigarrow \textbf{if } \zeta \textbf{ then } e_1' \textbf{ else } e_2} \quad (\texttt{N}-\texttt{then})$$

$$\frac{e_2 \rightsquigarrow e_2'}{\textbf{if } \zeta \textbf{ then } e_1 \textbf{ else } e_2 \rightsquigarrow \textbf{if } \zeta \textbf{ then } e_1 \textbf{ else } e_2'} \quad (\texttt{N}-\texttt{else})$$

$$\frac{sv_2 = \textbf{if } \zeta_1 \textbf{ then } \zeta_2 \textbf{ else } bv_2}{\textbf{if } \zeta_0 \textbf{then } bv_1 \textbf{ else } sv_2 \rightsquigarrow \textbf{if } (\neg\, \zeta_0 \wedge \zeta_1) \textbf{ then } \zeta_2 \; \textbf{else } (\textbf{if } \zeta_0 \textbf{ then } bv_1 \textbf{ else } bv_2)} \quad (\texttt{N}-\texttt{grpR1})$$

$$\frac{sv_1 = \textbf{if } \zeta_1 \textbf{ then } \zeta_2 \textbf{ else } bv_1}{\textbf{if } \zeta_0 \textbf{then } sv_1 \textbf{ else } bv_2 \rightsquigarrow \textbf{if } (\zeta_0 \wedge \zeta_1) \textbf{ then } \zeta_2 \; \textbf{else } (\textbf{if } \zeta_0 \textbf{ then } bv_1 \textbf{ else } bv_2)} \quad (\texttt{N}-\texttt{grpR2})$$

$$\frac{sv_1 = \textbf{if } \zeta_1 \textbf{ then } \zeta_2 \textbf{ else } bv_1 \quad sv_2 = \textbf{if } \zeta_3 \textbf{ then } \zeta_4 \textbf{ else } bv_2}{\textbf{if } \zeta_0 \textbf{ then } sv_1 \textbf{ else } sv_2 \rightsquigarrow \textbf{if } (\zeta_0 \wedge \zeta_1) \textbf{ then } \zeta_2 \; \textbf{else } (\textbf{if } \zeta_0 \textbf{ then } bv_1 \textbf{ else } sv_2)} \quad (\texttt{N}-\texttt{grpR3})$$

Fig. 5. Normalization Rules II

Normalization of an expression always terminates. If the normal form is an s-value, the generation of parallel code is guaranteed. Otherwise, the expression cannot be parallelized by our `PType` system.

3 PType System

Normalization of an expression terminates with three possible kinds of normal forms: (1) one that does not contain any self-recursive calls; (2) one that is an s-value; and (3) one that contains self-recursive calls *but* is not an s-value. The objective of `PType` system is to classify an expression *symbolically* according to the kind of normal form it can normalize to.

`PType` expressions are defined in Figure 7. `PType` consists of `NType`s and `RType`s. An `NType` is a collection of *syntactic* expressions that do not contain any occurrences of self-recursive call $\underline{\bullet}$ or its reference. On the other hand, an `RType` is a collection of *syntactic* expressions that contain occurrences of $\underline{\bullet}$, *and* can be normalized to s-values. The S in R_S is a sequence of n binary operators satisfying extended-ring property. For example, for the self-recursive equation of the following function definition

$$f_6\ (a : x) = 5\ `max`\ (a + 2 \times (f_6\ x)),$$

$$\textbf{if } \zeta \textbf{ then } \underline{\bullet} \textbf{ else } \underline{\bullet} \rightsquigarrow \underline{\bullet} \qquad (\mathtt{N-rmTest})$$

$$\frac{bv_1 = \zeta_1 \oplus_i bv_3 \quad bv_2 = \zeta_2 \oplus_i bv_4}{\begin{array}{l}\textbf{if } \zeta_0 \textbf{then } bv_1 \textbf{ else } bv_2 \rightsquigarrow (\textbf{if } \zeta_0 \textbf{ then } \zeta_1 \textbf{ else } \zeta_2) \oplus_i \\ \qquad (\textbf{if } \zeta_0 \textbf{ then } bv_3 \textbf{ else } bv_4)\end{array}} \qquad (\mathtt{N-pushIf1})$$

$$\frac{\begin{array}{c} i < j \\ bv_1 = \zeta_1 \oplus_i bv_3 \quad bv_2 = \zeta_2 \oplus_j bv_4\end{array}}{\begin{array}{l}\textbf{if } \zeta_0 \textbf{then } bv_1 \textbf{ else } bv_2 \rightsquigarrow (\textbf{if } \zeta_0 \textbf{ then } \zeta_1 \textbf{ else } \iota_i) \oplus_i \\ \qquad (\textbf{if } \zeta_0 \textbf{ then } bv_3 \textbf{ else } bv_2)\end{array}} \qquad (\mathtt{N-pushIf2})$$

$$\frac{\begin{array}{c} i > j \\ bv_1 = \zeta_1 \oplus_i bv_3 \quad bv_2 = \zeta_2 \oplus_j bv_4\end{array}}{\begin{array}{l}\textbf{if } \zeta_0 \textbf{ then } bv_1 \textbf{ else } bv_2 \rightsquigarrow (\textbf{if } \zeta_0 \textbf{ then } \iota_j \textbf{ else } \zeta_2) \oplus_j \\ \qquad (\textbf{if } \zeta_0 \textbf{ then } bv_1 \textbf{ else } bv_4)\end{array}} \qquad (\mathtt{N-pushIf3})$$

$$\frac{bv_1 = \underline{\bullet} \quad bv_2 = \zeta_2 \oplus_i \underline{\bullet}}{\begin{array}{l}\textbf{if } \zeta_0 \textbf{ then } bv_1 \textbf{ else } bv_2 \rightsquigarrow (\textbf{if } \zeta_0 \textbf{ then } \iota_i \textbf{ else } \zeta_2) \oplus_i \\ \qquad (\textbf{if } \zeta_0 \textbf{ then } \underline{\bullet} \textbf{ else } \underline{\bullet})\end{array}} \qquad (\mathtt{N-pushIf4})$$

$$\frac{bv_1 = \zeta_1 \oplus_i \underline{\bullet} \quad bv_2 = \underline{\bullet}}{\begin{array}{l}\textbf{if } \zeta_0 \textbf{ then } bv_1 \textbf{ else } bv_2 \rightsquigarrow (\textbf{if } \zeta_0 \textbf{ then } \zeta_1 \textbf{ else } \iota_i) \oplus_i \\ \qquad (\textbf{if } \zeta_0 \textbf{ then } \underline{\bullet} \textbf{ else } \underline{\bullet})\end{array}} \qquad (\mathtt{N-pushIf5})$$

Fig. 6. Normalization Rules III

its RHS has type $R_{[max,+,\times]}$.

$$\begin{array}{lll}
\rho & \in & \mathtt{PType} \\
\rho & ::= & \psi \mid \phi \\
\psi & \in & \mathtt{NType} \\
\psi & ::= & N \\
\phi & \in & \mathtt{RType} \\
\phi & ::= & R_S \\
\end{array}$$

where S is a sequence of operators

Fig. 7. Type Expressions

Any expression containing $\underline{\bullet}$ but *cannot* be normalized to an s-value is considered ill-typed in our PType system.

We write $[\![\rho]\!]$ to denote the semantics of PType ρ. Thus,

$$[\![N]\!] \;=\; \textbf{C–Exp},$$

where **C–Exp** is defined in Figure 3.

Given $S = [op_1, \ldots, op_n]$ with extended-ring property, then

$$\begin{array}{rcl}[\![R_S]\!] & = & \{e \mid e \rightsquigarrow^* e' \;\wedge\; e' \text{ is an } \textit{s-value} \\ & & \quad\quad \wedge\; e' \text{ is composable by operators in } S\},\end{array}$$

where $\rightsquigarrow^*$ represents a normalization process.

Note that we say the expression e' is composable, rather than composed directly, by a set of operators. There are two reasons for saying that:

1. e' need not simply be an s-value of *bv* category; it can also include conditional and local abstraction, but its set of operators must be limited to S.
2. As operators in S have identities, we allow e' to contain just a subset of operators in S. We can always extend e' to contain all operators in S using their respective identities.

The last point implies that the semantics of RType enjoys the following subset relation:

Lemma 2. *Given two sequences of operators S_1 and S_2, both with extended-ring property. If S_1 is a subsequence of S_2, then $[\![R_{S_1}]\!] \subseteq [\![R_{S_2}]\!]$.*

The above lemma induces the following type subtyping relation:

Definition 3 (Subtyping of RType). *Given two sequences of operators S_1 and S_2, both with extended-ring property. We say R_{S_1}* is a subtype of *R_{S_2}, denoted by $R_{S_1} <: R_{S_2}$, if and only if $S_1 \ll S_2$ (where "$S_1 \ll S_2$" means "S_1 is a subsequence of S_2").*

A *type assumption* Γ binds program variables to their PTypes. A judgment of the PType has the form

$$\Gamma \vdash_\kappa e :: \rho$$

This states that the expression e has PType ρ assuming that any free variable in it has PType given by Γ and κ is an expression that may occur in e. κ is either a self-recursive call or a *reference* to such call. It represents the *currently active reference* (the detail can be seen in the type-checking rule for **let**.) Before type checking the RHS of a recursive definition of f, we initiate κ to be the term $(f\ x)$. we also assign in, Γ, PType N to the recursive parameters of f.

Analogous to the well known Damas-Milner type system, we can now state the objective of PType system through the notion of well-PTypedness.

Definition 4 (Well-PTypedness). *Given a recursive equation of f defined by $f\ (a : x) = e$. The expression e is said to be* well-PTyped *if there is some* PType *ρ such that $\Gamma \vdash_{(f\ x)} e :: \rho$, where Γ assigns a to N and x to N.*

$$\frac{}{\Gamma \vdash_\kappa n :: N} \quad \texttt{(con)}$$

$$\frac{v \neq \kappa}{\Gamma \cup \{v :: N\} \vdash_\kappa v :: N} \quad \texttt{(var - N)}$$

$$\frac{v = \kappa}{\Gamma \cup \{v :: R_S\} \vdash_\kappa v :: R_S} \quad \texttt{(var - R)}$$

$$\frac{}{\Gamma \vdash_{(f\,x)} (f\,x) :: R_S} \quad \texttt{(rec)}$$

$$\frac{\Gamma \vdash_\kappa e_1 :: N \quad \Gamma \vdash_\kappa e_2 :: \rho \quad (\rho = N) \vee (\rho = R_S \wedge \oplus \in S)}{\Gamma \vdash_\kappa (e_1 \oplus e_2) :: \rho} \quad \texttt{(op)}$$

$$\frac{\Gamma \vdash_\kappa e_0 :: N \quad \Gamma \vdash_\kappa e_1 :: \rho_1 \quad \Gamma \vdash_\kappa e_2 :: \rho_2 \quad \bigtriangledown_{\texttt{if}} (\rho, \rho_1, \rho_2)}{\Gamma \vdash_\kappa (\textbf{if}\, e_0\, \textbf{then}\, e_1 \textbf{else}\, e_2) :: \rho} \quad \texttt{(if)}$$

$$\frac{\Gamma \vdash_\kappa e_1 :: N \quad \Gamma \cup \{v :: N\} \vdash_\kappa e_2 :: \rho}{\Gamma \vdash_\kappa (\textbf{let}\, v = e_1\, \textbf{in}\, e_2) :: \rho} \quad \texttt{(let - N)}$$

$$\frac{\Gamma \vdash_\kappa e_1 :: R_S \quad \Gamma \cup \{v :: R_S\} \vdash_v e_2 :: R_S}{\Gamma \vdash_\kappa (\textbf{let}\, v = e_1\, \textbf{in}\, e_2) :: R_S} \quad \texttt{(let - R)}$$

$$\frac{\Gamma \vdash_\kappa e :: N \quad g \notin FV(\kappa)}{\Gamma \vdash_\kappa (g\,e) :: N} \quad \texttt{(g)}$$

$$\frac{\Gamma \vdash_\kappa e : \rho \quad \rho <: \rho'}{\Gamma \vdash_\kappa e :: \rho'} \quad \texttt{(sub)}$$

Fig. 8. Type-Checking Rules

3.1 Type-Checking Rules

The `PType` of a function f is defined as the `PType` of the RHS of its recursive equation. Figure 8 lists the type-checking rules which are explained below.

Since any expression not enclosing any references of the recursive call $(f\ x)$ will be given `NType`. Thus, both constants and variables not referencing the recursive call are given `NType`, as shown in the rules (`var-N`) and (`con`).

Use of variable referencing self-recursive call will be given an `RType` if it is the currently active references; *ie.*, it is the same as κ. The self-recursive call $(f\ x)$ will also be given an `RType`. We note that any use of inactive references are ill-`PType`, as there is no corresponding rule for it. This is reasonable, since such use is non-linear.

In rule (op), binary operations over expressions of `NType` yield expressions of `NType`. Again, we restrict our discussion to the case where the self-recursive call occurs in the right operand. Thus, the operation yields an `RType` if the right operand is already so, and the operator under investigation is part of the sequence S.

We restrict any references to the self-recursive call from appearing in conditional-test position. Thus, in rule `(if)`, a conditional expression is of `NType` if both its branches are of `NType`. On the other hand, it is of `RType` if one of its branches is of `RType`. When both branches are of `RType`, the conditional will be of `RType` provided both branches can be normalized to s-values composable by operators in S (*aka.* R_S.) These inferences are expressed by the operator $\bigtriangledown_{\texttt{if}}$, which is declared as follows:

$$\frac{}{\bigtriangledown_{\texttt{if}}(\rho,\rho,\rho)} \qquad \frac{}{\bigtriangledown_{\texttt{if}}(R_S,N,R_S)} \qquad \frac{}{\bigtriangledown_{\texttt{if}}(R_S,R_S,N)}$$

There are two rules for **let**-expression. Rule `let-N` applies to expressions with no recursive-call references in e_1. Thus, the resulting type depends on the type of e_2. Rule `let-R` applies to expressions with recursive-call references occurring in e_1 and variable v is used in the expression e_2.

Note that in the rule (let-R), the deductive operator has changed from $\vdash_\kappa$ to $\vdash_v$. This means that in e_2, v is the sole active reference to the recursive function. Thus, the following two expressions will fail the `PType` check: In the first expression, the recursive call is non-linear; in the second expression, the use of v is non-linear.

$\textbf{let}\ v\ =\ f\ x\ \ \textbf{in}\ f\ x$
$\textbf{let}\ v\ =\ f\ x\ \ \textbf{in let}\ \ u\ =\ v\ \ \textbf{in}\ \ v$

In rule `(g)`, the application of an auxiliary function g is of `NType` if its argument e is of `NType` too. Otherwise, such application may not be effectively parallelized [12], and the application will be deemed ill-`PTyped`.

Note that while we consider the presence of mutumorphisms, such as $(q\ x)$, during normalization, we need not formulate a separate typing rule for such application. In fact, the distinction between $(q\ x)$ and ordinary auxiliary function call is only required at parallelization phase.

The `(sub)` rule provides a linkage between the subtyping and typing relations.

3.2 Soundness of `PType` System and Strong Normalization

In this section, we provide the soundness of our type-checking rules with respect to normalization process by proving the progress and preservation theorems. Note: For Theorem 1, Theorem 2 and Theorem 3, expressions that do not contain recursive call or references to recursive calls are not in the scope of these proofs.

Soundness of PType System For the proof of the progress theorem, it is convenient to record a couple of facts about the possible shapes of the canonical forms of RType.

Lemma 3 (Canonical Forms). *If e is an s-value of* PType $R_{[\oplus_1,\,\ldots,\oplus_n]}$ *where* $\forall\, n.\ n \geq 0$ *(when $n = 0$, it is $R_{[\,]}$), then e is either* $(\zeta_1\ \oplus_1\ \ldots\ \oplus_n\ \underline{\bullet})$ *or* (**if** ζ_a **then** ζ_b **else** $(\zeta_1 \oplus_1\ \ldots\ \oplus_n\ \underline{\bullet})$).

Proof. The grammar in Figure 3 gives the desired result immediately.

Theorem 1 (Progress). *If $\Gamma\ \vdash_\kappa\ e :: R_S$, then either e is an s-value or $e \rightsquigarrow\ e'$.*

Proof. By induction on the normalization of $e :: R_S$. The `self-rec` case is immediate since e is an s-value. For the other cases, we argue as follows.
Case `(var-R)`: Since v is in Γ, by induction hypothesis, the result is immediate.

Case `(op)` where $(\rho = R_S\ \wedge \oplus\ \in S)$: By induction hypothesis, either e_2 is an s-value or else there is some e_2' such that $e_2 \rightsquigarrow\ e_2'$. If e_2 is an s-value, either e is an s-value or the canonical forms lemma assures us that e_2 must be one of the canonical forms in which case either `N-distr`, `N-semiassoc` or `N-liftIf1` applies to $(e_1 \oplus\ e_2)$. On the other hand, if $e_2 \rightsquigarrow\ e_2'$, then `N-op` applies.

Case `(if)` where $\bigtriangledown_{\texttt{if}}\,(R_S, N, R_S)$: By induction hypothesis, either e_2 is a s-value or else there is some e_2' such that $e_2 \rightsquigarrow\ e_2'$. If e_2 is an s-value, e is either an s-value or the canonical forms lemma assures us that e_2 must be the third form in which case `N-grpN` applies. On the other hand, if $e_2 \rightsquigarrow\ e_2'$, then `N-else` applies.

Case `(if)` where $\bigtriangledown_{\texttt{if}}\,(R_S, R_S, N)$: By induction hypothesis, either e_1 is a s-value or else there is some e_1' such that $e_1 \rightsquigarrow\ e_1'$. If e_1 is an s-value, rule `N-sv` applies. On the other hand, if $e_1 \rightsquigarrow\ e_1'$, then `N-then` applies.

Case `(if)` where $\bigtriangledown_{\texttt{if}}\,(R_S, R_S, R_S)$: If both e_1 and e_2 are s-values either rule `N-grpR1`, `N-grpR2`, `N-grpR3`, `N-rmTest`, `N-pushIf1-5` applies. If e_1 is not an s-value but e_2 is an s-value, rule `N-then` applies. If e_2 is not an s-value, `N-else` applies.

Case `(let-N)` where $e_2 :: R_S$: If e_2 is an s-value, `N-sub` applies. If it is not, `N-in` applies.

Case `(let-R)`: If e_1 is not an s-value, `N-let` applies. If e_1 is an s-value and e_2 is not an s-value, `N-in` applies. If both e_1 and e_2 are s-values, `N-sub` applies. □

Theorem 2 (Preservation). *If $e :: R_S$ and $e\ \rightsquigarrow e'$, then $e' :: R_S$*

Proof. By induction on a derivation of $e :: R_S$. At each step of the induction, we assume that the desired property holds for all subderivations and proceed by

case analysis on the final rule in the derivation.
Case (var-R): Since v is in Γ, by induction hypothesis, the result is immediate.

Case (rec): If the last rule in the derivation is (rec), then we know from the form of this rule that e must be $\underline{\bullet}$. Since e is an s-value, it cannot be the case that $e \rightsquigarrow e'$ for any e', and the theorem is vacuously satisfied.

Case (op) where ($\rho = R_S \wedge \oplus \in S$): If this is the last rule in the derivation, we know from the form of this rule that e may have the form $\zeta \oplus e_2$, for some ζ and e_2. We must also have subderivations with conclusions $\zeta :: N$, $e_2 :: R_S$ and $\oplus \in S$. Now, looking at the normalization rules with such form on the left-hand side in Figure 4, we find that there are four rules by which $e \rightsquigarrow e'$ can be derived. We consider each case separately.

Subcase N-op: Applying induction hypothesis to subderivation $e_2 \rightsquigarrow e_2'$, we get $e_2' :: R_S$. We can apply rule (op) to get $(\zeta \oplus e_2') :: R_S$.

Subcase N-distr: If $e \rightsquigarrow e'$ is derived using N-distr, then from the form of this rule we can see that $\zeta_1 :: N$, $(\zeta_2 \oplus_i bv_1) :: R_S$ and $\oplus_j \in S$. we also know $\zeta_1 :: N$, $\zeta_2 :: N$, $bv_1 :: R_S$ and $\oplus_i \in S$ by hypothesis. With rule (op), we can obtain $(\zeta_1 \oplus_j \zeta_2) :: N$, $(\zeta_1 \oplus_j bv_1) :: R_S$ and $((\zeta_1 \oplus_j \zeta_2) \oplus_i (\zeta_1 \oplus_j bv_1)) :: R_S$.

Subcase N-semiassoc: If $e \rightsquigarrow e'$ is derived using N-semiassoc, then from the form of this rule we can see that $\zeta_1 :: N$, $(\zeta_2 \oplus bv_1) :: R_S$ and $\oplus_j \in S$. we also know $\zeta_1 :: N$, $\zeta_2 :: N$, $bv_1 :: R_S$ and $\oplus \in S$ by hypothesis. Applying rule (op), we obtain $(\zeta_1 \oplus' \zeta_2) :: N$ and $((\zeta_1 \oplus' \zeta_2) \oplus bv_1) :: R_S$.

Subcase N-liftIf: Applying induction hypothesis to subderivations, we get (**if** $\zeta_0 :: N$ **then** $\zeta_2 :: N$ **else** $lv :: R_S) :: R_S$. Applying rule (op), we get $(\zeta_1 \oplus_i lv) :: R_S$ and $(\zeta_1 \oplus_i \zeta_2) :: N$. Applying rule (if), we get (**if** ζ_0 **then** $\zeta_1 \oplus_i \zeta_2$ **else** $\zeta_1 \oplus_i lv) :: R_S$.

Case (if) where $\bigtriangledown_{\texttt{if}} (R_S, N, R_S)$: If this is the last rule in the derivation, we know from the form of this rule that (**if** $e_0 :: N$ **then** $e_1 :: N$ **else** $e_2 :: R_S) :: R_S$. We find that N-else and N-grpN can be applied. We consider each case separately.

Subcase N-else: Applying induction to subderivation $e_2 \rightsquigarrow e_2'$, we get $e_2' :: R_S$. Applying typing rule (if), we get (**if** ζ **then** e_1 **else** $e_2') :: R_S$

Subcase N-grpN: Applying rule (op), we have $(\zeta_0 \vee \zeta_2) :: N$ and applying rule (if), we have (**if** ζ_0**then** ζ_1 **else** $\zeta_3) :: N$. Applying induction to subderivation (**if** ζ_2 **then** ζ_3 **else** lv, we get $lv :: R_S$. Applying rule (if), we obtain (**if** $(\zeta_0 \vee \zeta_2)$ **then** (**if** ζ_0**then** ζ_1 **else** ζ_3) **else** $lv) :: R_S$.

Case `(if)` where $\bigtriangledown_{\texttt{if}}(R_S, R_S, N)$: Applying induction on subderivations, we get $\zeta_0 :: N$, $sv :: R_S$ and $\zeta_2 :: N$. We find that `N-then` and `N-sv` can be applied. We consider each case separately.

Subcase `N-then`: By induction hypothesis, $e_1' :: R_S$. Applying rule `if`, we have $(\textbf{if}\ \zeta_0\ \textbf{then}\ e_1'\ \textbf{else}\ \zeta_2) :: R_S$

Subcase `N-sv`: Applying rule `(g)`, $\neg\zeta_0 :: N$. Applying rule `(if)`, $(\textbf{if}\ \neg\zeta_0\ \textbf{then}\ \zeta_2\ \textbf{else}\ sv) :: R_S$.

Case `(if)` where $\bigtriangledown_{\texttt{if}}(R_S, R_S, R_S)$: If this is the last rule in the derivation, then we know from the form of this rule that e must have the form $(\textbf{if}\ e_0\ \textbf{then}\ e_1 :: R_S\ \textbf{else}\ e_2 :: R_S) :: R_S$. Now, looking at the normalization rules with such form on the left-hand side in Figure 5 and 5, we find that there are eleven rules by which $e \rightsquigarrow e'$ can be derived. We consider each case separately.

Subcase `N-then`: Applying induction to subderivations, we get $e_1' :: R_S$ and $e_2 :: R_S$. Applying rule `(if)`, we have $(\textbf{if}\ e_0\ \textbf{then}\ e_1'\ \textbf{else}\ e_2) :: R_S$.

Subcase `N-else`: Similar to that of Subcase `N-then`.

Subcase `N-grpR1`: Applying induction to subderivation, we get $\zeta_0, \zeta_1, \zeta_2 :: N$, $sv_1, lv_2 :: R_S$. Applying rule `(g)`, we have $\neg\zeta_0 :: N$. Applying rule `(if)`, we have $(\textbf{if}\ \zeta_0\ \textbf{then}\ sv_1\ \textbf{telse}\ lv_1) :: R_S$. Applying rule `(if)` again, we obtain $((\textbf{if}\ \neg\zeta_0 \wedge \zeta_1\ \textbf{then}\ \zeta_2\ \textbf{else}\ (\textbf{if}\ \zeta_0\ \textbf{then}\ sv_1\ \textbf{else}\ lv_1)) :: R_S$

Subcase `N-grpR2` & Subcase `N-grpR3`: Similar to that of Subcase `N-grpR1`.

Subcase `N-rmTest`: Apply induction to subderivation, we get $\underline{\bullet} :: R_S$.

Subcase `N-pushIf1`: Apply induction to subderivations, we get $\zeta_0, \zeta_1, \zeta_2 :: N$ and $bv_3, bv_4 :: R_S$. Applying rule `(if)`, we can get $(\textbf{if}\zeta_0\ \textbf{then}\ \zeta_1\ \textbf{else}\ \zeta_2) :: N$ and $(\textbf{if}\ \zeta_0\ \textbf{then}\ bv_3\ \textbf{else}\ bv_4) :: R_S$. Applying rule `(op)`, $((\textbf{if}\ \zeta_0\textbf{then}\ \zeta_1\ \textbf{else}\ \zeta_2) \oplus_i (\textbf{if}\ \zeta_0\ \textbf{then}\ bv_3\ \textbf{else}\ bv_4)) :: R_S$

Subcase `N-pushIf2` to Subcase `N-pushIf5`: Similar to that of Subcase `N-pushIf1`.

Case `(let-N)`: If the last rule in the derivation is `(let-N)`, we know that e must have the form $(\textbf{let}\ \ v = e_1 :: N\ \textbf{in} e_2 :: \rho) :: \rho$. It matches either `N-in` or `N-sub`. We consider each case separately.

Subcase `N-in`: By induction hypothesis,$e_2' :: \rho$. Applying rule `(let-N)`, we get $(\textbf{let}\ v = e_1\ \textbf{in}\ e_2') :: \rho$

Subcase `N-sub`: The result is immediate.

Case (let-R) where $\bigtriangledown_{\texttt{let}} (R_S, R_S, R_S)$: If this is the last rule in the derivation, then we know from the form of this rule that e must have the form (**let** $v = e_1 :: R_S$ **in** $e_2 :: R_S) :: R_S$. Now, looking at the normalization rules with such form on the left-hand side in Figure 4, we find that there are three rules by which $e \leadsto e'$ can be derived. We consider each case separately.

Subcase N-let:By induction hypothesis, e_1' and e_1 have the same type. Thus, applying rule (let-R), **let** $v = e_1'$ **in** e_2 has the same type as **let** $v = e_1$ **in** e_2.

Subcase N-in: By induction hypothesis, e_2' and e_2 have the same type. Thus, applying rule (let-R), **let** $v = e_1$ **in** e_2' has the same type as **let** $v = e_1$ **in** e_2.

Subcase N-sub: The result is immediate. □

Strong Normalization In this section, we prove that a well-PTyped expression is strong normalizing with respect to the normalization rules in Figure 4, 5 and 6.

We prove it by induction on the size of the syntax tree. Usually strong normalization property cannot be proven using the size of the syntax tree as a function application may increase the syntax tree to an arbitrary size. However, in our case, we work at meta-level so that we can employ this technique with the help of the definitions of *atomic expression* and *effective syntax tree.*

Definition 5 (Atomic Expression). *An atomic expression is either an expression of* NType *or a recursive call.*

All atomic expressions are in normal form as there is no normalization rule can be applied to them.

Definition 6 (Effective Syntax Tree). *An effective syntax tree is a syntax tree built from atomic expressions.*

An atomic expression in an effective syntax tree is considered as one node. Thus, the size of each atomic expression in the tree one.

For example, we have function $f\ (a : x) = (1 + a * 2) + (f\ x)$ the depth of its effective syntax tree is two though the size of its concrete syntax tree is four.

There are three cases that may lead to increasing of the size of the syntax tree in the proof of strong normalization in lambda calculus.

1. recursive call
2. function application (other than recursive call)
3. let-expression

In this section, we analyse each case to show that why they are no longer a problem in our strong normalization proof.

Case 1 (recursive call): Our system works at meta-level. Thus, a recursive call is treated as an atomic expression (i.e. $\underline{\bullet}$ in the earlier part of the thesis). This means no substitution takes place.

Case 2 (other function application): Similarly to the reason in Case 1, other function applications (i.e. $(q\ x)$ or $(g\ x)$ mentioned in the earlier part of the thesis) all have type of N during normalization. Thus, they are considered as atomic expressions as well.

Case 3 (let-expression): There are two subcases to consider. Given an let-expression of the form $(\textbf{let}\ v\ =\ e_1\ \textbf{in}\ e_2)$, one subcase is $e_1 :: N$ and $e_2 :: R_S$; the other subcase is $e_1 :: R_S$ and $e_2 :: R_S$.

For the first subcase, although there may be many copies of v in the expression e_2, according to the definition of atomic expression, they still form one atomic expression and thus the size of the effective syntax tree is reduced by one instead after substitution. For example, we have function definition as follows.

$$f\ (a : x)\ =\ \textbf{let}\ v\ =\ 2 + a\ \textbf{in}\ (v\ +\ a * v)\ +\ (f\ x)$$

After applying normalization rule `N-sub`, we have

$$f\ (a : x)\ =\ ((2 + a)\ +\ a * (2 + a))\ +\ (f\ x)$$

where according to the definition of atomic expression, expression $((2 + a)\ +\ a * (2 + a))$ is still considered as one atomic expression of size one though it is larger than the expression $(v\ +\ a * v)$.

For the second subcase, there is only one v appear in expression e_2 as e is well-`PType`d. Before applying normalization rule `N-sub`, both e_1 and e_2 are in normal form. So the rule `N-sub` does not increase the size of the effective syntax tree.

Theorem 3 **(Strong Normalization).** *If* $e :: \rho$*, then* $\exists\ e'$ *s.t.* $e \leadsto^{*}\ e' \not\leadsto$.

Proof. We prove it by induction on the size of the effective syntax tree (EST).
Case `N-op`: By induction hypothesis, $e_2 \leadsto\ e_2'$ decreases the size of the EST, it is obvious that $\zeta\ \oplus_i\ e_2\ \leadsto \zeta\ \oplus_i\ e_2'$ decreases the size of the EST.

Case `N-distr`: The part that will participate in further normalization is $\zeta_1\ \oplus_j\ bv_1$. Thus, the size of EST is decreased by one.

Case `N-semiassoc`: The RHS of the rule is an s-value.

Case `N-liftIf`: The part that will participate in further normalization is $\zeta_1\ \oplus\ bv$. Thus, the size of EST is decreased by one.

Case `N-let`: By induction hypothesis, $e_1\ \leadsto\ e_1'$ decreases the size of the EST, it is obvious that the normalization process below the line decreases the size of the EST.

Case `N-let`: By induction hypothesis, $e_2 \rightsquigarrow e_2'$ decreases the size of the EST, it is obvious that the normalization process below the line decreases the size of the EST.

Case `N-sub`: We see that the size of the effective syntax tree is reduced. $e_2[v \mapsto e_1]$ is either an s-value or maps to any of the normalization rules.

Case `N-then`: By induction hypothesis, $e_1 \rightsquigarrow e_1'$ decreases the size of the EST, it is obvious that the normalization process below the line decreases the size of the EST.

Case `N-else`: By induction hypothesis, $e_2 \rightsquigarrow e_2'$ decreases the size of the EST, it is obvious that the normalization process below the line decreases the size of the EST.

Case `N-sv`: The RHS of the rule is an s-value.

Case `N-grpN`: The RHS of the rule is an s-value.

Case `N-grpR1`: The part that will participate in further normalization is **if** ζ_0 **then** bv_1 **else** bv_2. Thus, the size of the effective syntax tree is reduced.

Case `N-grpR2` & Case `N-grpR3`: Similar reasoning as that of the Case `N-grpR1`.

Case `N-rmTest`: The RHS of the rule is an s-value.

Case `N-pushIf1`: The part that will participate in further normalization is **if** ζ_0 **then** bv_3 **else** bv_4. Thus, the size of the effective syntax tree is reduced.

Case `N-pushIf2` to Case `N-pushIf5`: Similar reasoning as that of the Case `N-pushIf1`.

From the above case analysis, we can see that each normalization rule either help reducing the size of the effective syntax tree or its RHS is an s-value. This proves the lemma. □

3.3 PType Inference Algorithm

The `PType` inference is done on an extension of the `PType` with *sequence variables*. The extension, called **EPtype**, is defined as follows:

$$
\begin{array}{rcll}
\rho & \in & \mathbf{EPType} & \text{— Extended } \texttt{PType} \\
& \rho & ::= \; N \mid R_\Pi & \\
\Pi & \in & \mathbf{ESeq} & \text{— Extended Sequences} \\
& \Pi & ::= \; S \mid \beta \mid S \bowtie \beta & \\
\beta & \in & \mathbf{SVar} & \text{— Sequence Variables} \\
S & \in & \mathbf{Seq} & \text{— Sequences} \\
& S & ::= \; [\,] \mid [\oplus_1, \ldots, \oplus_n] &
\end{array}
$$

Note that S denotes a (possibly empty) sequence of operators, satisfying the extended ring property. The extended sequence Π can either be a sequence, a sequence variable, or a joint between a sequence and a sequence variable. The latter enables a sequence to be extended to include new operators. We also identify $S \bowtie [\,]$ with S. This allows us to "terminate" the extended sequence by converting it into a normal sequence.

The subtyping relation for extended-`PType` system is extended from the original type subtyping with sequence variables. The subtyping rules are as follows:

$$
\beta \ll \beta \qquad \frac{S_1 \ll S_2}{(S_1 \bowtie \beta) \ll (S_2 \bowtie \beta)}
$$

$$
\rho <: \rho \qquad \frac{\rho_1 <: \rho_2 \quad \rho_2 <: \rho_3}{\rho_1 <: \rho_3} \qquad \frac{\Pi_1 \ll \Pi_2}{R_{\Pi_1} <: R_{\Pi_2}}
$$

The `EPType`-type inference algorithm, $\mathcal{W}_{\|\kappa}$, is defined in Figure 9. It is expressed as:

$$
\begin{array}{l}
\mathcal{W}_{\|\kappa} \; :: \; (\mathbf{Exp}, \mathbf{Env}) \rightarrow (\mathbf{EPType}, \mathbf{Sub}) \\
\mathcal{W}_{\|\kappa}(e, \Gamma) \; = \; (\rho, \theta)
\end{array}
$$

where $\Gamma \in \mathbf{Env}$ is type assumption containing mapping between program variables and `EPType`s, and $\theta \in \mathbf{Sub}$ is a substitution *mapping* sequence variables to extended sequences.

Before inferencing the RHS of a definition of a function f, all parameter variables of f will be kept in the initial environment Γ_{init} and are assigned the `EPType` N. Moreover, κ is set to the term representing the self-recursive call to f, such as $(f\ x)$.

Unification of `EPType` is performed by the function $\mathcal{U}$, the definition of which, as well as its associated functions, are defined in Figure 10 and Figure 11 respectively. Application of a substitution to an `EPType` (or an environment or an extended sequence) is performed by the overloaded function *app*. Similarly, composition of substitutions are defined by the operator ;. These are defined in Figure 11. Lastly, we define a *ground substitution*, $\theta_{[\,]}$ such that, $\forall\, \beta$, $app(\theta_{[\,]}, \beta) = [\,]$, the empty sequence. An application of $\theta_{[\,]}$ to an `EPType`-value will effectively eliminate any sequence variables occurring in it. A substitution θ is a *ground validation* of Γ and ρ if and only if it is a ground substitution that covers Γ and ρ. We can safely say $\theta_{[\,]}$ is a ground validation of Γ and ρ in all the cases.

$$\mathcal{W}_{\|\kappa}(n,\Gamma) \;=\; (N,\{\})$$

$$\begin{array}{l}
\mathcal{W}_{\|\kappa}(v,\Gamma) \;=\; \\
\quad \mathit{if}\ v \neq \kappa\ \ \mathit{then\ let}\ \rho = \Gamma(v) \\
\qquad\qquad\qquad \mathit{in\ if}\ (\rho == N)\ \mathit{then}\ (N,\{\}) \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{else\ Error} \\
\quad \mathit{else}\ \ (\Gamma(v),\{\})
\end{array}$$

$$\begin{array}{l}
\mathcal{W}_{\|\kappa}(f\ x,\Gamma) \;=\; \mathit{if}\ (f\ x) == \kappa\ \mathit{then}\ \ \mathit{let}\ \beta\ \text{ be fresh} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{in}\ \ (R_\beta,\{\}) \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{else\ Error}
\end{array}$$

$$\begin{array}{l}
\mathcal{W}_{\|\kappa}(e_1 \oplus e_2,\Gamma) \;=\; \\
\quad \mathit{let}\ (\rho_1,\theta_1) \;=\; \mathcal{W}_{\|\kappa}(e_1,\Gamma) \\
\qquad\ (\rho_2,\theta_2) \;=\; \mathcal{W}_{\|\kappa}(e_2, \mathit{app}(\theta_1,\Gamma)) \\
\quad \mathit{in}\ \ \mathit{case}\ (\mathit{app}(\theta_2,\rho_1),\rho_2)\ \mathit{of} \\
\qquad (N,N) \ \ \rightarrow (N,\theta_2\,;\theta_1) \\
\qquad (N,R_S) \rightarrow \mathit{let}\ \beta'\ \text{be fresh} \\
\qquad\qquad\qquad\qquad S' \;=\; [\oplus] \bowtie \beta' \\
\qquad\qquad\qquad\qquad \theta_3 \;=\; \mathcal{U}(R_S, R_{S'}) \\
\qquad\qquad\qquad \mathit{in}\ \ (\mathit{app}(\theta_3,\ R_S),\ \theta_3\,;\theta_2\,;\theta_1) \\
\qquad (_,_) \qquad \rightarrow \mathit{Error}
\end{array}$$

$$\begin{array}{l}
\mathcal{W}_{\|\kappa}(\mathbf{if}\ e_0\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2,\Gamma) \;=\; \\
\ \ \mathit{let}\ (\rho_0,\theta_0) \;=\; \mathcal{W}_{\|\kappa}(e_0,\Gamma) \\
\qquad (\rho_1,\theta_1) \;=\; \mathcal{W}_{\|\kappa}(e_1, \mathit{app}(\theta_0,\Gamma)) \\
\qquad (\rho_2,\theta_2) \;=\; \mathcal{W}_{\|\kappa}(e_2, \mathit{app}(\theta_1\,;\theta_0,\Gamma)) \\
\qquad \theta \qquad\quad = \theta_2\,;\theta_1\,;\theta_0 \\
\ \ \mathit{in\ if}\ (\mathit{app}(\theta_2\,;\theta_1,\rho_0)) \;=\; N \\
\quad \mathit{then}\ \ \mathit{case}\ (\mathit{app}(\theta_2,\rho_1),\rho_2)\ \mathit{of} \\
\qquad\quad (N,N) \qquad \rightarrow (N,\theta) \\
\qquad\quad (N,R_S) \quad\ \ \rightarrow (R_S,\theta) \\
\qquad\quad (R_S,\ N) \quad\ \ \rightarrow (R_S,\theta) \\
\qquad\quad (R_{S_1},R_{S_2}) \rightarrow \mathit{let}\ \theta' \;=\; \mathcal{U}(R_{S_1},R_{S_2}) \\
\qquad\qquad\qquad\qquad\quad \mathit{in}\ \ (\mathit{app}(\theta',R_{S_1}),\ \theta'\,;\theta) \\
\quad \mathit{else\ Error}
\end{array}$$

$$\begin{array}{l}
\mathcal{W}_{\|\kappa}(\mathbf{let}\ v \;=\; e_1\ \mathbf{in}\ e_2,\Gamma) \;=\; \\
\quad \mathit{let}\ (\rho_1,\theta_1) \;=\; \mathcal{W}_{\|\kappa}(e_1,\Gamma) \\
\quad \mathit{in\ if}\ (\rho_1 == N)\ \ \mathit{then} \\
\qquad\quad \mathcal{W}_{\|\kappa}(e_2,\{v :: N\} \cup \Gamma) \\
\qquad \mathit{else}\ \ \mathit{let}\ (\rho_2,\theta_2) \;=\; \mathcal{W}_{\|\ v}\,(e_2,\{v :: \rho_1\} \cup \Gamma) \\
\qquad\qquad\qquad \theta \;=\; \mathcal{U}(\mathit{app}(\theta_2,\rho_1),\rho_2) \\
\qquad\qquad \mathit{in}\ \ (\mathit{app}(\theta,\rho 2),\ \theta\,;\theta_2\,;\theta_1)
\end{array}$$

$$\begin{array}{l}
\mathcal{W}_{\|\kappa}(g\ e,\Gamma) \;=\; \\
\quad \mathit{let}\ (\rho,\theta) \;=\; \mathcal{W}_{\|\kappa}(e,\Gamma) \\
\quad \mathit{in\ if}\ (\rho == N\ \&\ g\ \notin\ FV(\kappa))\ \mathit{then}\ (N,\theta)\ \mathit{else\ Error}
\end{array}$$

Fig. 9. Type Inference Algorithm

$$
\begin{aligned}
&\mathcal{U} :: (\,\mathbf{EPType},\ \mathbf{EPType}) \rightarrow\ \mathbf{Sub}\\
&\mathcal{U}(N, N) \ = \ \{\}\\
&\mathcal{U}(R_{\Pi_1}, R_{\Pi_2}) \ = \ \mathcal{U}_S(\Pi_1, \Pi_2)\\
&\mathcal{U}(\rho_1, \rho_2) \ = \ \mathit{Error}\\
\\
&\mathcal{U}_S :: (\,\mathbf{ESeq},\ \mathbf{ESeq}) \rightarrow\ \mathbf{Sub}\\
&\mathcal{U}_S(\beta, \Pi) \ = \ \{\beta \mapsto \Pi\}\\
&\mathcal{U}_S(\Pi, \beta) \ = \ \{\beta \mapsto \Pi\}\\
&\mathcal{U}_S(S_1, S_2 \bowtie \beta) = \mathit{let}\ S \ = \ S_1 \uplus S_2\\
&\qquad\qquad\qquad\qquad \mathit{in}\ \mathit{if}\ (S \ == \ S_2\ \&\ S \neq S_1)\\
&\qquad\qquad\qquad\qquad\quad \mathit{then}\ \mathit{Error}\\
&\qquad\qquad\qquad\qquad\quad \mathit{else}\ \mathit{let}\ T_2 \ = \ \mathit{diff}(S, S_2)\\
&\qquad\qquad\qquad\qquad\qquad \mathit{in}\ \{\beta \mapsto T2\}\\
&\mathcal{U}_S(S_1 \bowtie \beta, S2) \ = \mathcal{U}_S(S_2, S_1 \bowtie \beta)\\
&\mathcal{U}_S(S_1 \bowtie \beta_1, S_2 \bowtie \beta_2) \ =\\
&\qquad \mathit{let}\ S \ = \ S_1 \uplus S_2\\
&\qquad\quad T_1 \ = \ \mathit{diff}(S, S_1)\\
&\qquad\quad T_2 \ = \ \mathit{diff}(S, S_2)\\
&\qquad\quad \beta \text{ be fresh}\\
&\qquad \mathit{in}\ \{\beta_1 \mapsto (T_1 \bowtie \beta),\ \beta_2 \mapsto (T_2 \bowtie \beta)\}
\end{aligned}
$$

Fig. 10. Unify Function

The algorithm resembles a typical type inference algorithm. The detail can be found in standard program analysis textbook [22].

The correctness of $\mathcal{W}_{\|\kappa}$ can be expressed as follows:

Theorem 4 (Soundness of $\mathcal{W}_{\|\kappa}$). *Given a type environment Γ and an expression e. If $\mathcal{W}_{\|\kappa}(e, \Gamma) = (\rho, \theta)$ for some ρ and θ, then $app(\theta_{[\,]}; \theta, \Gamma) \vdash_\kappa e :: app(\theta_{[\,]}, \rho)$.*

Proof. By the definition of $\theta_{[\,]}$, $\theta_{[\,]}$ is a ground validation for all θ. If θ is a ground validation of $app(\theta 1; \theta 2, \Gamma)$, then $(\theta; \theta 1)$ is a ground validation of $app(\theta 2, \Gamma)$.

The proof proceeds by structural induction on e (because $\mathcal{W}_{\|\kappa}$ is defined by structural induction on e).

Case n: We have $\mathcal{W}_{\|\kappa}(n, \Gamma) = (N, \{\})$. From rule `(con)` in Figure 8, it is immediate that $\Gamma \vdash_\kappa n :: N$.

Case v: There are two subcases we need to consider.

Subcase $v \neq \kappa$: We have $\mathcal{W}_{\|\kappa}(v, \Gamma) = (N, \{\})$ provided $\Gamma(v) = N$. From rule `(var-N)` in Figure 8, it is immediate that $\Gamma \cup \{v :: N\} \vdash_\kappa v :: N$.

Subcase $v == \kappa$: We have $\mathcal{W}_{\|\kappa}(v, \Gamma) = (\Gamma(v), \{\})$. From rule `(var-R)` in Figure 8, it is immediate that $app(\theta_{[\,]}, \Gamma \cup \{v :: R_S\}) \vdash_\kappa v :: app(\theta_{[\,]}, \Gamma(v))$.

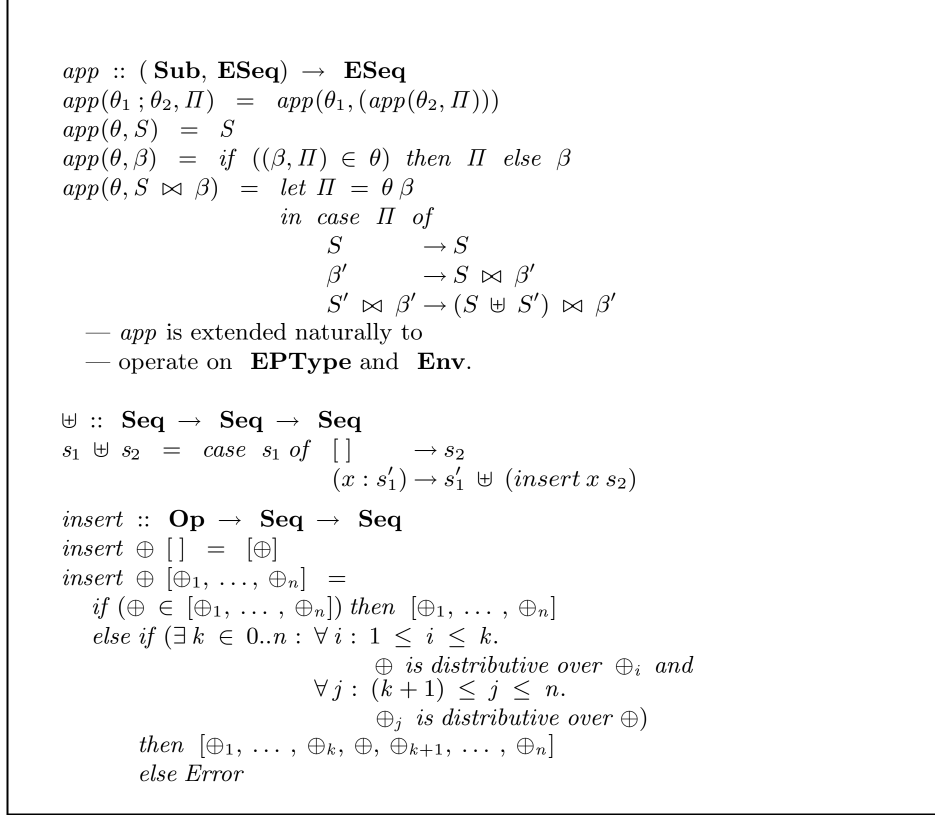

Fig. 11. Associate Functions to Type Inference Algorithm

Case $(f\ x)$: We have $\mathcal{W}_{\|\kappa}(f\ x, \Gamma) = (R_\beta, \{\})$. From rule `(rec)` in Figure 8, it is immediate that $app(\theta_{[\,]}, \Gamma) \vdash_\kappa (f\ x) :: R_{[\,]}$ where $app(\theta_{[\,]}, R_\beta) = R_{[\,]}$.

Case $e_1 \oplus e_2$: We shall use the notion established in the clause for $\mathcal{W}_{\|\kappa}(e_1 \oplus e_2, \Gamma)$. There are two subcases to consider.

Subcase $(app(\theta_2, \rho_1) == N, \rho_2 == N)$: $\theta_{[\,]}$ is a ground validation of $app(\theta_2; \theta_1, \Gamma)$. Then $\theta_{[\,]} ; \theta_2$ is an ground validation of $app(\theta_1, \Gamma)$. Hence by the induction hypothesis, we get $app(\theta_{[\,]}; \theta_2; \theta_1, \Gamma) \vdash_\kappa e_1 :: N$ and $app(\theta_{[\,]};\theta_2 ;\theta_1, \Gamma) \vdash_\kappa e_2 :: N$. We can apply rule `(op)` and get $app(\theta_{[\,]}; \theta_2;\theta_1, \Gamma) \vdash_\kappa (e_1 \oplus e_2) :: N$ which is the desired result.

Subcase $(app(\theta_2, \rho_1) == N, \rho_2 == R_S)$: $\theta_{[\,]}$ is a ground validation of $app(\theta_3; \theta_2; \theta_1, \Gamma)$ and $app(\theta_3, R_S)$. Then $\theta_{[\,]} ;\theta_3$ is a ground validation of $app(\theta_2 ;\theta_1, \Gamma)$. $\theta_{[\,]}; \theta_3; \theta_2$ is a ground validation of $app(\theta_1, \Gamma)$. Hence by the induction hypothesis, we get

$app(\theta_{[\,]}; \theta_3 ;\theta_2 ; \theta_1,\ \Gamma) \vdash_\kappa\ e_1\ ::\ N$ and $app(\theta_{[\,]}; \theta_3 ; \theta_2 ; \theta_1, \Gamma) \vdash_\kappa\ e_2\ :: app(\theta_{[\,]} ; \theta_3;\ \theta_2; \theta_1 , R_S)$. Since we have $\mathcal{W}_{\|\kappa}((e_1\ \oplus\ \ e_2, \Gamma)\ = (app(\theta_3,\ R_S),\ \theta_3; \theta_2 ;\ \theta_1)$, we get $\oplus\ \in\ S$. We can apply rule (op) and get $app(\theta_{[\,]}; \theta_3; \theta_2 ; \theta_1,\ \Gamma) \vdash_\kappa (e_1 \oplus e_2) ::\ app\ (\theta_{[\,]};\ \theta_{[\,]} ;\theta_3;\theta_2; \theta_1,\ R_S)$ which is the desired result.

Case (**if** e_0 **then** e_1 **else** e_2): We shall use the notion established in the clause for $\mathcal{W}_{\|\kappa}$(**if** e_0 **then** e_1 **else** e_2, Γ). If e_0 has type N, there are four subcases to consider.

Subcase ($\rho_1 == N$ *and* $\rho_2 == N$): $\theta_{[\,]}$ is a ground validation of $app(\theta_{[\,]}; \theta, \Gamma)$ and ρ_2. Then $\theta_{[\,]}; \theta_2$ is a ground validation of $app(\theta_1; \theta_0, \Gamma)$ and ρ_1. By the induction hypothesis, we get $app(\theta_{[\,]}; \theta, \Gamma)\ \vdash_\kappa\ e_1 :: app(\theta_{[\,]};\theta, \rho_1)$ and $app(\theta_{[\,]}; \theta, \Gamma)\ \vdash_\kappa e_2 :: app(\theta_{[\,]};\ \theta, \rho_2)$. We apply rule (if) and get $\Gamma\ \vdash_\kappa$ (**if** e_0**then** e_1 **else** e_2) $:: N$ where $app(\theta_{[\,]};\theta, N) = N$.

Subcase ($\rho_1 == N$ *and* $\rho_2 == R_{S_1}$): $\theta_{[\,]}$ is a ground validation of $app(\theta_{[\,]}; \theta, \Gamma)$ and ρ_2. Then $\theta_{[\,]}; \theta_2$ is a ground validation of $app(\theta_1; \theta_0, \Gamma)$ and ρ_1. By the induction hypothesis, we get $app(\theta_{[\,]}; \theta, \Gamma)\ \vdash_\kappa\ e_1 :: app(\theta_{[\,]};\theta, \rho_1)$ and $app(\theta_{[\,]}; \theta, \Gamma)$ $\vdash_\kappa\ e_2 :: app\ (\theta_{[\,]}; \theta, \rho_2)$. We apply rule (if) and get $app(\theta_{[\,]}; \theta, \Gamma)\ \vdash_\kappa$ (**if** e_0 **then** e_1 **else** e_2) $:: app(\theta_{[\,]}\ ; \theta, \rho_2)$

Subcase ($\rho_1 == R_{S_1}$ *and* $\rho_2 == N$): Similar reasoning as Subcase ($\rho_1 == N$ *and* $\rho_2 == R_{S_1}$).

Subcase ($\rho_1 == R_{S_1}$ *and* $\rho_2 == R_{S_2}$): $\theta_{[\,]}$ is a ground validation of $app(\theta'; \theta, \Gamma)$ and if $U(R_{S_1}, R_{S_2})\ =\ \theta$ then $app(\theta_{[\,]};\ \theta, R_{S_1})\ =\ app(\theta_{[\,]};\ \ \theta, R_{S_2})$. Then $\theta_{[\,]}; \theta'$ is a ground validation of $app(\theta, \Gamma)$, R_{S_1} and R_{S_2}. By the induction hypothesis, we get $app(\theta_{[\,]}; \theta';\theta,\ \Gamma)\ \vdash_\kappa\ e_1 :: app(\theta_{[\,]}; \theta'; \theta, R_{S_1})$ and $app\ (\theta_{[\,]}; \theta';\theta,\ \Gamma)\ \vdash_\kappa\ e_2 :: app$ $(\theta_{[\,]}; \theta'; \theta, R_{S_2})$. Since θ' is the result of unifying R_{S_1} and R_{S_1}, $app(\theta_{[\,]}; \theta'; \theta, R_{S_1)}$ is as same as $app(\theta_{[\,]}; \theta'; \theta, R_{S_2)}$. Applying rule (if) and get $app(\theta_{[\,]}; \theta';\ \theta, \Gamma)$ $\vdash_\kappa$ (**if** e_0 **then**e_1 **else** e_2) $:: app(\theta_{[\,]}; \theta'\ ; \theta, R_{S_1})$

Case ($g\ e$): We shall use the notion established in the clause for $\mathcal{W}_{\|\kappa}((g\ e), \Gamma)$. Hence by the induction hypothesis we get: $app(\theta_{[\,]}, \Gamma)\ \vdash_\kappa\ e :: app(\theta_{[\,]}, \rho)$. If $\rho == N$, we can apply rule (g) and get $\Gamma \vdash_\kappa\ (g\ e) :: N$ which is the desired result.

Case **let** $v = e_1$ **in** e_2: There are two subcases to consider.

Subcase $\rho_1 == N$: $\theta_{[\,]}$ is a ground validation of Γ. By the induction hypothesis, we have $app(\theta_{[\,]}, \Gamma)\ \vdash_\kappa e_1\ ::\ \rho_1$ and $app(\theta_{[\,]}, \Gamma)\ \cup\ \{v :: \rho_1\} \vdash_\kappa e_2 :: app(\theta_{[\,]}, \rho_2)$. Applying rule (let-N), $app(\theta_{[\,]}, \Gamma)\ \cup\ \{v :: N\} \vdash_\kappa$ (**let** $v\ =\ e_1$ **in** e_2) $:: app(\theta_{[\,]}, \rho_2)$.

Subcase $\rho_1 == R_{S_1}$: $\theta_{[\,]}$ is a ground validation of $app(\theta; \theta_2;\ \theta_1, \Gamma)$. Then $\theta_{[\,]}; \theta$ is a ground validation of $app(\theta_2; \theta_1, \Gamma)$ and ρ_2. $\theta_{[\,]}; \theta; \theta_2$ is a ground validation

of $app(\theta_1, \Gamma)$ and ρ_1. By induction hypothesis, we have $app(\theta_{[\,]}; \theta; \theta_2; \theta_1, \Gamma) \vdash_\kappa e_1 :: app(\theta_{[\,]}; \theta; \theta_2; \theta_1, \rho_1)$ and $app(\theta_{[\,]}; \theta; \theta_2; \theta_1, \Gamma) \cup \{v :: \rho_1\} \vdash_v e_2 :: app(\theta_{[\,]}; \theta; \theta_2; \theta_1, \rho_2)$. Applying rule `(let-R)`, $app(\theta_{[\,]}; \theta; \theta_2; \theta_1, \Gamma) \cup \{v :: app(\theta_{[\,]}; \theta; \theta_2; \theta_1, \rho_1)\} \vdash_v (\textbf{let } v = e_1 \textbf{ in } e_2) :: app\ (\theta_{[\,]}; \theta; \theta_2; \theta_1, \rho_2)$. □

Theorem 5 (Completeness of $\mathcal{W}_{\|\kappa}$). *For any type environment Γ and expression e, if there exists a* `PType` *ρ' such that $\Gamma \vdash_\kappa e :: \rho'$, then there exists ρ such that $\mathcal{W}_{\|\kappa}(e, \Gamma) = (\rho, \theta)$ and $app(\theta_{[\,]}, \rho) <: \rho'$.*

Proof. The proof is by induction on the shape of the inference tree since the type-checking rules in Figure 8 are syntax directed.
Case n: We have $\Gamma \vdash_\kappa n :: N$. Clearly $\mathcal{W}_{\|\kappa}(n, \Gamma) = (N, \{\})$.

Case $v \neq \kappa$: We have $\Gamma \cup \{v :: N\} \vdash_\kappa v :: N$. Clearly $\mathcal{W}_{\|\kappa}(v, \Gamma') = (\Gamma'(v), \{\})$ where $\Gamma' = \Gamma \cup \{v :: N\}$.

Case $v = \kappa$: We have $\Gamma \cup \{v :: R_S\} \vdash_\kappa v :: R_S$. Clearly $\mathcal{W}_{\|\kappa}(v, \Gamma') = (\Gamma'(v), \{\})$ where $\Gamma' = \Gamma \cup \{v :: N\}$.

Case $(f\ x)$: We have $\Gamma \vdash_{(f\ x)} (f\ x) :: R_S$. Clearly $\mathcal{W}_{\|\kappa}(f\ x, \Gamma) = (R_\beta, \{\})$, $app(\theta_{[\,]}, R_\beta) = R_{[]}$ and $R_{[]} <: R_S$ for all S.

Case $(e_1 \oplus e_2)$: There are two type-checking rules we need to consider. We consider each case separately.

Subcase `(op)` where $\rho = N$: We have $\Gamma \vdash_\kappa (e_1 \oplus e_2) :: N$. If rule `(op)` is the last rule applied, we have $\Gamma \vdash_\kappa e_1 :: N$ and $\Gamma \vdash_\kappa e_2 :: N$. By induction hypothesis, we have $\mathcal{W}_{\|\kappa}(e_1, \Gamma) = (N, \{\})$ and $\mathcal{W}_{\|\kappa}(e_2, \Gamma) = (N, \{\})$ respectively. Clearly $\mathcal{W}_{\|\kappa}(e_1 \oplus e_2, \Gamma) = (N, \{\})$.

Subcase `(op)` where $\rho = R_S \wedge \oplus \in S$: We have $\Gamma \vdash_\kappa (e_1 \oplus e_2) :: R_S$. If rule `(op)` is the last rule applied, we have $\Gamma \vdash_\kappa e_1 :: N$, $\Gamma \vdash_\kappa e_2 :: R_S$. By induction hypothesis, we have $\mathcal{W}_{\|\kappa}(e_1, \Gamma) = (N, \theta_1)$ and $\mathcal{W}_{\|\kappa}(e_2, \Gamma) = (R_S, \theta_2)$. Since $\oplus \in S$, $\theta_3 = \{\}$. Clearly, $\mathcal{W}_{\|\kappa}(e_1 \oplus e_2, \Gamma) = (R_S, \theta_2; \theta_1)$ and $R_S <: R_S$.

Case (**if** e_0 **then** e_1 **else** e_2): There are four type-checking rules we need to consider. We consider each case separately.

Subcase `(if)` where $\bigtriangledown_{\texttt{if}} (N, N, N)$: We have $\Gamma \vdash_\kappa (\textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2) :: N$. If rule `if` is the last rule applied, we have $e_0 :: N$, $e_1 :: N$ and $e_2 :: N$. By induction hypothesis, we have $\mathcal{W}_{\|\kappa}(e_0, \Gamma) = (N, \{\})$, $\mathcal{W}_{\|\kappa}(e_1, \Gamma) = (N, \{\})$ and $\mathcal{W}_{\|\kappa}(e_2, \Gamma) = (N, \{\})$. Clearly $\mathcal{W}_{\|\kappa}(\textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2, \Gamma) = (N, \{\})$.

Subcase `(if)` where $\bigtriangledown_{\texttt{if}} (R_S, N, R_S)$: We have $\Gamma \vdash_\kappa (\textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2) :: R_S$. If rule `(if)` is the last rule applied, we have $\Gamma \vdash_\kappa e_0 :: N$, $\Gamma \vdash_\kappa e_1 :: N$ and $\Gamma \vdash_\kappa e_2 :: R_S$. By induction hypothesis, we have $\mathcal{W}_{\|\kappa}(e_0, \Gamma) = (N, \theta_0)$, $\mathcal{W}_{\|\kappa}(e_1, \Gamma)$

$= (N, \theta_1)$ and $\mathcal{W}_{\|\kappa}(e_2, \Gamma) = (R_S, \theta_2)$ where R_S <: R_S. Clearly $\mathcal{W}_{\|\kappa}(\textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2, \Gamma) = (R_S, \theta_2; \theta_1; \theta_0)$.

Subcase `(if)` where $\bigtriangledown_{\texttt{if}}\ (R_S, R_S, N)$: Similar to the Subcase `if` where $\bigtriangledown_{\texttt{if}}\ (R_S, N, R_S)$.

Subcase `(if)` where $\bigtriangledown_{\texttt{if}}\ (R_S, R_S, R_S)$: We have $\Gamma \vdash_\kappa (\textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2) :: R_S$. If rule `(if)` is the last rule applied, we have $\Gamma \vdash_\kappa e_0 :: N$, $\Gamma \vdash_\kappa e_1 :: R_{S'}$ and $\Gamma \vdash_\kappa e_2 :: R_{S'}$. By induction hypothesis, we have $\mathcal{W}_{\|\kappa}(e_0, \Gamma) = (N, \theta_0)$, $\mathcal{W}_{\|\kappa}(e_1, \Gamma) = (R_S, \theta_1)$ and $\mathcal{W}_{\|\kappa}(e_2, \Gamma) = (R_S, \theta_2)$ and $app(\theta_{[\,]}, R_S) <: R_{S'}$. Clearly we have $\mathcal{W}_{\|\kappa}(\textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2, \Gamma) = (app(\theta', R_S), \theta'; \theta_2; \theta_1; \theta_0)$. Since $U(R_S, R_S) = \{\}$, $\theta' = \{\}$. This is the desired result.

Case $(g\ e)$: We have $\Gamma \vdash_\kappa e :: \rho$. From rule `(g)` in Figure 8, we know that $e :: N$ and $g \notin FV(\kappa)$. By induction hypothesis, we have $\mathcal{W}_{\|\kappa}(e, \Gamma) = (N, \{\})$. Clearly $\mathcal{W}_{\|\kappa}(g\ e, \Gamma) = (N, \{\})$.

Case $(\mathcal{W}_{\|\kappa}(\textbf{let } v = e_1 \textbf{ in } e_2, \Gamma))$: We have $\Gamma \vdash_\kappa (\textbf{let } v = e_1 \textbf{ in } e_2) :: \rho$. There are two cases to consider.

Subcase `(let-N)`: We have $\Gamma \vdash_\kappa e_1 :: N$, $\Gamma \vdash_\kappa v :: N$ and $\Gamma \vdash_\kappa e_2[v \mapsto e_1] :: \rho'$. By induction hypothesis, $\mathcal{W}_{\|\kappa}(e_1, \Gamma) = (N, \theta_1)$ and $\mathcal{W}_{\|\kappa}(e_2[v \mapsto e_1], \{v :: (N, \{\})\} \cup \Gamma) = (\rho, \theta_2; \theta_1)$ and $app(\theta_{[\,]}, \rho) <: \rho'$. Type ρ is the desire result.

Subcase `(let-R)` where $\rho == R_S$: We have $\Gamma \vdash_\kappa e_1 :: R_{S'}$, $\Gamma \cup \{v :: R_{S'}\} \vdash_\kappa e_2 :: R_{S'}$. By induction hypothesis, $\mathcal{W}_{\|\kappa}(e_1, \Gamma) = (R_S, \theta_1)$ and $\mathcal{W}_{\|\kappa}(e_2[v \mapsto e_1], \{v :: R_S\} \cup \Gamma) = (R_S, \theta_2)$ and $app(\theta_{[\,]}, R_S) <: R_{S'}$. Type R_S is the desire result.

□

4 Parallel Code Derivation

Once a function has been inferred to have `RType`, we can automatically derive its parallel counterpart. In the parallel context, the most commonly used technique is *divide-and-conquer* and the computation model for it is called *homomorphism*. In this chapter, we give a background to homomorphism and a more expressive parallel computation model *mutumorphism*, which is used in our system. After that, we describe informally an algorithm for deriving parallel code from expression of `RType` and give its correctness proof.

4.1 Homomorphisms and Mutumorphisms

In skeleton approach to data parallel list programming, homomorphisms are often used as the basic divide-and-conquer scheme. Homomorphisms are a good characterization of parallel computational models and can be effectively implemented on modern parallel architectures [**?**,15, 9].

Definition 7 (List Homomorphism). *A function hom is a homomorphism if it satisfies the equations:*

$$\begin{aligned} hom_{(f,\oplus)}\ [a] &= f\ a \\ hom_{(f,\oplus)}\ (xl ++ xr) &= (hom_{(f,\oplus)}\ xl) \oplus (hom_{(f,\oplus)}\ xr) \end{aligned}$$

It can also be expressed with two primitive skeletons *reduce* and *map* as follows.

$$hom_{(f,\oplus)}\ (xs) = reduce_{\oplus}\ (map\ f\ (xs))$$

A *near* homomorphism proposed by Cole is the composition of a projection function and a homomorphism. With Cole's idea and results in [17, 18], we choose list *mutumorphisms* [13] as our parallel computation model.

Definition 8 (List Mutumorphisms). *The functions $h_1 \ldots h_n$ are called list mutumorphisms (or mutumorphisms for short) if they are mutually defined in the following way:*

$$\begin{aligned} h_j\ [a] &= k_j\ a \\ h_j\ (x +\!\!+ y) &= ((\triangle_1^n\ h_i)\ x) \oplus_j ((\triangle_1^n\ h_i)\ y) \end{aligned}$$

Particularly, a single function, say h_i, is said to be a list mutumorphism, if there exist a set of functions $h_1, \ldots, h_{i-1}, h_{i+1}, \ldots, h_n$ which together with h_i satisfying the above equational form.

$\triangle_1^n\ f_i$ abbreviates $f_1 \triangle \ldots \triangle f_n$ where $\triangle$ is a binary operator on tuples, defined by

$$(f \triangle g)\ a = (f\ a, g\ a).$$

Mutumorphisms have more powerful descriptive power than homomorphisms. They are considered as most general recursive functions defined in an inductive manner [13], being capable of describing most interesting functions. They can be automatically transformed into efficient homomorphisms via tupling calculation as have been intensively studied in [5, 18].

Theorem 6 (Tupling [18]). *Let $h_1, \ldots, h_n$ be mutumorphism as defined in Definition 8. Then, $\triangle_1^n\ h_i = (\![\ \triangle_1^n\ k_i\ , \triangle_1^n \oplus_i\ ]\!)$ where $(\![\ k, \oplus]\!) = (reduce \oplus) \circ (map\ k)$.*

4.2 Parallel Code Derivation

Our running example is the following general definition of a sequential function f:

$$\begin{aligned} f\ [a] &= Ctx_1[a, (q\ x)] \\ f\ (a : x) &= Ctx_2[a, \langle (q\ x) \rangle, (f\ x)] \end{aligned}$$

— Ctx_1 and Ctx_2 are arbitrary contexts

If f is well-`PTyped`, f can be normalized to an s-value. In this section, we show how to parallelize each s-value automatically. If f has return type $R_{[\,]}$, parallel

$$
\begin{aligned}
&h_a\ [a]\ z\ =\ g_a\ a\ z\\
&h_a\ (xl +\!\!+\ xr)\ z\ =\ h_a\ xl\ (q\ xr\ \oplus_q\ \ z)\ \vee\ h_a\ xr\ z\\
\\
&h_b\ [a]\ z\ =\ g_b\ a\ z\\
&h_b\ (xl +\!\!+\ xr)\ z\\
&\quad = \textbf{if}\ h_a\ xl\ (q\ xr\ \oplus_q\ z)\ \textbf{then}\ h_b\ xl\ (q\ xr\ \oplus_q\ \ z)\\
&\qquad \textbf{else}\ h_1\ xl\ (q\ xr\ \oplus_q\ z) \oplus_1\ \ldots \oplus_{n-1}\\
&\qquad\qquad h_n\ xl\ (q\ xr\ \oplus_q\ z)\ \oplus_n h_b\ xr\ z
\end{aligned}
$$

— $h_i\ \forall\, 1 \leq\ i \leq\ p$ is defined as follows.

$$
\begin{aligned}
&h_i\ [a]\ =\ g_i\ a\ z\\
&h_i\ (xl +\!\!+\ xr)\ z\\
&\ =\ h_i\ xl\ (q\ xr \oplus_q\ z)\ \oplus_i\ \ldots\ \oplus_{n-1}\ h_n xl\ (q\ \ xr \oplus_q\ z)\\
&\quad \oplus_n\ h_i\ xr\ z
\end{aligned}
$$

Fig. 12. Auxillary functions

code is obvious [19]. For other cases, we consider them one by one. Auxilary functions are defined in Figure 12. Note: "$g_i\ a\ (q\ x)$" can be considered and read as "any expression involving a and/or $(q\ x)$".

If the RHS of f is normalized to a bv of the form

$$f(a:x) = g_1\ a\ (q\ x)\ \oplus_1 \cdots \oplus_{n-1}\ g_n\ a\ (q\ x) \oplus_n \underline{\bullet}$$

function f's parallel version is as follows.

$$
\begin{aligned}
&f\ [a]\ =\ Ctx_1[a]\\
&f\ (xl +\!\!+\ xr)\\
&\ =\ h_1\ xl\ (q\ xr)\ \oplus_1\ \ldots\ \oplus_{n-1}\ h_n\ xl\ (q\ xr)\ \oplus_n\ (f\ xr)
\end{aligned}
$$

If the RHS of f is normalized to an s-value of the form

$$
\begin{aligned}
&f\ (a:x)\ =\\
&\quad \textbf{let}\ v\ =\ g_1\ a\ (q\ x)\ \oplus_1\ \ldots\ \oplus_{n-1}\ g_n\ a\ (q\ x)\ \oplus_n\ \underline{\bullet}\\
&\quad \textbf{in}\ g_c\ a\ (q\ x)
\end{aligned}
$$

function f's parallel version is as follows.

$$
\begin{aligned}
&f\ [a]\ =\ g_c\ a\ (q\ x)\\
&f\ (xl +\!\!+\ xr)\\
&\ =\ \textbf{let}\ v\ =\ h_1\ xl\ (q\ xr)\ \oplus_1\ \ldots\ \oplus_{n-1}\ h_n\ xl\ (q\ xr)\ \oplus_n\ (f\ xr)\\
&\qquad \textbf{in}\, f\ xl
\end{aligned}
$$

If the RHS of f is normalized to an s-value of the form

$$
\begin{aligned}
f\ (a:x)\ =\ &\textbf{if}\ g_a\ a\ (q\ x)\textbf{then}\ g_b\ a\ (q\ x)\\
&\textbf{else}\ g_1\ a\ (q\ x)\ \oplus_1\ \ldots\ \oplus_{n-1}\ g_n\ a\ (q\ x) \oplus_n\ \underline{\bullet}
\end{aligned}
$$

function f's parallel version is as follows:

$f\ [a]\ =\ Ctx_1[a]$
$f\ (xl +\!\!+\ xr)\ =$
$\quad \textbf{if}\ h_a\ xl\ (q\ x) \textbf{then}\ h_b\ xs\ (q\ x)$
$\quad \textbf{else}\ h_1\ xl\ (q\ xr)\ \oplus_1\ \ldots\ \oplus_{n-1}\ h_n\ xl\ (q\ xr)\ \oplus_n\ (f\ xr)$

If the RHS of f is normalized to an s-value of the form

$f\ (a : x)\ =$
$\quad \textbf{if}\ g_a\ a\ (q\ x) \textbf{then}\ g_b\ a\ (q\ x)$
$\quad \textbf{else let}\ v\ =\ g_1\ a\ (q\ x)\ \oplus_1\ \ldots\ \oplus_{n-1}\ g_n\ a\ (q\ x)\ \oplus_n\ \underline{\bullet}$
$\qquad \textbf{in}\ g_c\ a\ (q\ x)$

function f's parallel version is as follows.

$f\ [a]\ =\ g_c\ a\ (q\ x)$
$f\ (xl +\!\!+\ xr)$
$=\ \textbf{if}\ h_a\ xl\ (q\ x) \textbf{then}\ h_b\ xs\ (q\ x)$
$\quad \textbf{else let}\ v\ =\ h_1\ xl\ (q\ xr)\ \oplus_1\ \ldots\ \oplus_{n-1}\ h_n\ xl\ (q\ xr)\ \oplus_n\ (f\ xr)$
$\qquad \textbf{in}\ f\ xl$

4.3 Correctness of Algorithm for Parallel Code Derivation

In [19], a primitive form $f(a : x)\ =\ g\ a\ (q\ x)\ \oplus\ (f\ x)$ and a conditional form

$$f\ (a : x)\ =\ \textbf{if}\ g_1\ a\ (q_1\ x)\ \textbf{then}\ g_2\ a\ (q_2\ x)\ \oplus\ (f\ x). \\ \qquad\qquad \textit{else}\ g_3\ a\ (q_3\ x)$$

Parallel versions and the correctness proofs for each form are given in [19] as well. In this thesis, with respect to the primitive form, we introduce a more general form bv that involves arbitrary number of binary operators (which fullfill extended ring property). In this chapter, we want to show the correctness of the parallel code derived for bv. The correctness of the parallel code derived for general conditional form follows from this proof. Regarding *let*-expression, the gist of the parallel code derivation is for the bv. That is to say, it suffices to show the correctness of the parallel code derived for bv.

Firstly, given $f\ (xl +\!\!+\ xr)\ =\ (h_1\ xl)\ op_1\ (h_2\ xl)\ op_2\ (f\ xr)$ is the parallel version of $f\ (a : x)\ =\ (g1\ a)\ op_1\ (g2\ \ a) op_2\ (f\ x)$, we want to show

$$h_1\ (xl +\!\!+\ xr)\ =\ (h_1\ xl)\ op_1\ (h_2\ xl)\ op_2\ (h_1\ xr)$$
$$h_2\ (xl +\!\!+\ xr)\ = (h_2\ xl)\ op_2\ (h_2\ xr)$$

are the parallel versions of $g1$ and $g2$ respectively.

Proof. We make use of the associative property of the constructor $+\!\!+$ i.e. the fact that $f\ ((xl_1\ +\!\!+\ xl_2)\ +\!\!+\ xr)\ = f\ (xl_1\ +\!\!+\ (xl_2\ +\!\!+\ xr))$.

$f\ \ ((xl_1\ \ +\!\!+\ \ xl_2)\ \ +\!\!+\ \ xr)$

$$= (h_1\ (xl_1\ +\!\!+\ xl_2))\ op_1\ (h_2\ (xl_1\ +\!\!+\ xl_2))\ op_2\ (f\ xr) \qquad (1)$$

$f\ (xl_1\ +\!\!+\ (xl_2\ +\!\!+\ xr))$
$= (h_1\ xl_1)\ op_1\ (h_2\ xl_1)\ op_2\ (f\ (xl_2\ +\!\!+\ xr))$
$= (h_1\ xl_1)\ op_1\ (h_2\ xl_1)\ op_2\ ((h_1\ xl_2)\ op_1\ (h_2\ xl_2)\ op_2\ (f\ xr))$
— op_2 is distributive over op_1.
$= (h_1\ xl_1)\ op_1\ (((h_2\ xl_1)\ op_2\ (h_1\ xl_2))\ op_1\ ((h_2\ xl) op_2\ (h_2 xl_2)\ op_2\ (f\ xr)))$
— op_1 is associative.
$= ((h_1\ xl_1)\ op_1\ ((h_2\ xl_1)\ op_2\ (h_1\ xl_2)))\ op_1$

$$((h_2\ xl_1)\ op_2\ (h_2\ xl_2)\ op_2\ (f xr)) \quad (2)$$

Since $f\ ((xl_1\ +\!\!+\ xl_2)\ +\!\!+\ xr) = f\ (xl_1\ +\!\!+\ (xl_2 +\!\!+\ xr))$, after unifying the RHS of equation 1 and 2 we have

$h_1\ (xl_1 +\!\!+\ xl_2) = (h_1\ xl_1)\ op_1\ (h_2\ xl_1)\ op_2\ (h_1\ xl_2)$
$h_2\ (xl_1 +\!\!+\ xl_2) = (h_2\ xl_1)\ op_2\ (h_2\ xl_2)$

□

Secondly, we show that given $f\ (xl +\!\!+\ xr) = (h_1\ xl)\ op_1\ (h_2 xl)\ op_2\ (h_3\ xl)\ op_3\ (f\ xr)$ to be the parallel version of $f\ (a : x) = (g1\ a\ (q\ x))\ op_1\ (g2\ a\ (q\ x))\ op_2\ (g3\ a\ (q\ x))\ op_3\ (f\ x)$,

$h_1\ (xl +\!\!+\ xr) = (h_1\ xl)\ op_1\ (h_2\ xl)\ op_2\ (h_3\ xl)\ op_3\ (h_1\ xr)$
$h_2\ (xl +\!\!+\ xr) = (h_2\ xl)\ op_2\ (h_3\ xl)\ op_3\ (h_2\ xr)$
$h_3\ (xl +\!\!+\ xr) = (h_3\ xl)\ op_3\ (h_3\ xr)$

are the parallel versions of $g1$, $g2$ and $g3$ respectively.

Proof. We make use of the associative property of the constructor $+\!\!+$ i.e. the fact that $f\ ((xl_1\ +\!\!+\ xl_2)\ +\!\!+\ xr) = f\ (xl_1\ +\!\!+\ (xl_2\ +\!\!+\ xr))$ again.
$f\ ((xl_1\ +\!\!+\ xl_2)\ +\!\!+\ xr)$

$$= (h_1\ (xl_1 +\!\!+ xl_2))\ op_1\ (h_2\ (xl_1 +\!\!+ xl_2))\ op_2\ (h_3\ (xl_1 +\!\!+ xl_2))\ op_3\ (f\ xr) \quad (3)$$

$f\ (xl_1\ +\!\!+\ (xl_2\ +\!\!+\ xr))$
$= (h_1\ xl_1)\ op_1\ (h_2\ xl_1)\ op_2\ (h_3\ xl_1)\ op_3\ (f\ (xl_2\ +\!\!+\ xr))$
$= (h_1\ xl_1)\ op_1\ (h_2\ xl_1)\ op_2\ (h_3\ xl_1)\ op_3$
$((h_1\ xl_2)\ op_1\ (h_2\ xl_2)\ op_2\ (h_3\ xl_2)\ op_3\ (f\ xr))$
— op_3 is distributive over op_1.
$= (h_1\ xl_1)\ op_1\ (h_2\ xl_1)\ op_2\ (((h_3\ xl_1)\ op_3\ (h_1\ xl_2))\ op_1$
$(h_3\ xl_1)\ op_3\ ((h_2\ xl_2)\ op_2\ (h_3\ xl_2)\ op_3\ (f\ xr)))$
— op_3 is distributive over op_2.
$= (h_1\ xl_1)\ op_1\ (h_2\ xl_1)\ op_2(((h_3\ xl_1)\ op_3\ (h_1\ xl_2))\ op_1$
$((h_3\ xl_1)\ op_3\ (h_2\ xl_2))\ op_2\ (h_3\ xl_1)\ op_3\ ((h_3\ xl_2)\ op_3\ (f\ xr)))$
— op_3 is associative.
$= (h_1\ xl_1)\ op_1\ (h_2\ xl_1)\ op_2(((h_3\ xl_1)\ op_3\ (h_1\ xl_2))\ op_1$
$((h_3\ xl_1)\ op_3\ (h_2\ xl_2))\ op_2\ ((h_3\ xl_1)\ op_3\ (h_3\ xl_2))\ op_3\ (f\ xr)))$
— op_2 is distributive over op_1.

$$= (h_1\ xl_1)\ op_1\ (((h_2\ xl_1)\ op_2\ ((h_3\ xl_1)\ op_3\ (h_1 xl_2)))\ op_1$$
$$(h_2\ xl_1)\ op_2((h_3\ xl_1)\ op_3\ (h_2\ xl_2))\ op_2\ (((h_3\ xl_1)\ op_3\ (h_3\ xl_2))\ op_3\ (f\ xr)))$$
— op_2 is associative.
$$= (h_1\ xl_1)\ op_1\ (((h_2\ xl_1)\ op_2\ ((h_3\ xl_1)\ op_3\ (h_1 xl_2)))\ op_1$$
$$((h_2\ xl_1)\ op_2((h_3\ xl_1)\ op_3\ (h_2\ xl_2)))\ op_2\ ((h_3\ xl_1)\ op_3\ (h_3\ xl_2))\ op_3\ (f\ xr))$$
— op_1 is associative.
$$= ((h_1\ xl_1)\ op_1\ (h_2\ xl_1)\ op_2\ (h_3\ xl_1)\ op_3\ (h_1\ xl_2))\ op_1$$
$$((h_2\ xl_1)\ op_2\ ((h_3\ xl_1)\ op_3\ (h_2\ xl_2)))\ op_2$$
$$((h_3\ xl_1)\ op_3\ (h_3\ xl_2))\ op_3\ (f\ xr) \tag{4}$$

Since $f\ ((xl_1 +\!\!+ xl_2) +\!\!+ xr) = f\ (xl_1 +\!\!+ (xl_2 +\!\!+ xr))$, after unifying equation 3 and 4 we have

$$h_1\ (xl_1 +\!\!+ xl_2) = (h_1\ xl_1)\ op_1\ (h_2\ xl_1)\ op_2\ (h_3\ xl_1)\ op_3\ (h_1\ xl_2)$$
$$h_2\ (xl_1 +\!\!+ xl_2) = (h_2\ xl_1)\ op_2\ (h_3\ xl_1)\ op_3\ (h_2\ xl_2)$$
$$h_3\ (xl_1 +\!\!+ xl_2) = (h_3\ xl_1)\ op_3\ (h_3\ xl_2)$$

□

The proofs for the above two cases give us confidence to give the following theorem.

Theorem 7. *Given function*

$$f\ [a] = Ctx_1\ [a]$$
$$f\ (a : x) = (g_1\ a) \oplus_1 \ldots \oplus_{n-1} (g_n\ a) \oplus_n (f\ x),$$

the parallel version of f *and the parallel versions of* $h_i\ \forall\, 1 \leq i \leq n$ *are*

$$f\ [a] = Ctx_1[a]$$
$$f\ (xl +\!\!+ xr) = (h_1\ xl) \oplus_1 \ldots \oplus_{n-1} (h_n\ xl) \oplus_n (f\ xr)$$

and

$$h_i\ [a] = g_i\ a\ z$$
$$h_i\ (xl +\!\!+ xr)\ z = (h_i\ xl) \oplus_i \ldots \oplus_{n-1} (h_n\ xl) \oplus_n (h_i\ xr)$$

respectively.

Proof. We prove it using mathematical induction on the number of operators used in the function body. Let $P(n)$ be the statement in the theorem 7 for n operators used in the function definition. Assume $P(k)$ is true. i.e.

$P(k)$: Given

$$f\ [a] = Ctx_1\ [a]$$
$$f\ (a : x) = (g_1\ a) \oplus_1 \ldots \oplus_{k-1} (g_k\ a) \oplus_k (f\ x),$$

the parallel version of f and the parallel versions of $h_i\ \forall\, 1 \leq i \leq k$ are

$$f\ [a]\ =\ Ctx_1[a]$$
$$f\ (xl \mathbin{+\!\!+} xr)\ =\ (h_1\ xl)\ \oplus_1\ \cdots \oplus_{k-1}\ (h_k\ xl)\ \oplus_k\ (f\ xr)$$

and

$$h_i\ [a]\ =\ g_i\ a\ z$$
$$h_i\ (xl \mathbin{+\!\!+} xr)\ z\ =\ (h_i\ xl)\ \oplus_i\ \ldots\ \oplus_{k-1}\ (h_k\ xl) \oplus_k\ (h_i\ xr)$$

respectively.

We want to show $P(k+1)$ is true. Since variable n denotes the number of operators used in the function definition, there is no difference to place the $(k+1)_{th}$ term $(g_{k+1}\ xl)$ at the 0_{th} position or $(k+1)_{th}$ position. For the easy reading of the proof, we put it at position 0 and use numbering 0 instead of $(k+1)$.

$P(k+1)$: Given

$$f\ [a]\ =\ Ctx_1\ [a]$$
$$f\ (a : x)\ =\ (g_0\ a)\ \oplus_0\ (g_1\ a)\ \oplus_1\ \ldots\ \oplus_{k-1}\ (g_k\ a)\ \oplus_k\ (f\ x)$$

Let

$$f\ (xl \mathbin{+\!\!+} xr)\ =\ (h_0\ xl) \oplus_0\ (h_1\ xl)\ \oplus_1\ \cdots \oplus_{k-1}\ (h_k\ xl)\ \oplus_k\ (f\ xr)$$

We know

$$f\ (xl_1 \mathbin{+\!\!+} xl_2)\ \mathbin{+\!\!+}\ xr)\ =\ h_0\ (xl_1 \mathbin{+\!\!+} xl_2)\ \oplus_0$$
$$(h_1\ (xl_1 \mathbin{+\!\!+} xl_2)\ \oplus_1\ \ \ldots\ \ \oplus_{k-1}\ h_k\ (xl_1 \mathbin{+\!\!+} xl_2)\ \oplus_k\ (f\ x)) \qquad (5)$$

$$f\ (xl_1 \mathbin{+\!\!+} (xl_2 \mathbin{+\!\!+} xr))$$
$$=\ h_0\ xl_1\ \oplus_0\ (h_1\ xl_1\ \oplus_1\ \ldots\ h_k\ xl_1\ \oplus_k\ (f\ (xl_2 \mathbin{+\!\!+} xr))$$
$$= h_0\ xl_1\ \oplus_0\ h_1\ xl_1\ \oplus_1\ \ldots\ h_k\ xl_1\ \oplus_k\ (h_0\ xl_2\ \oplus_0\ h_1\ xl_1\ \oplus_1\ \ldots\ h_n\ xl_1\ \oplus_k\ (f\ xr))$$
$$\vdots$$
$$\vdots$$
$$=\ (h_0\ xl_1\ \oplus_0\ h_1\ xl_1\ \oplus_1\ \ldots\ h_k\ xl_1 \oplus_k\ h_0\ xl_2) \oplus_0$$
$$((h_1\ xl_1\ \oplus_1\ \ \ldots\ h_k\ xl_1\ \oplus_k\ h_1\ xl_2)\ \oplus_1\ \ \ldots\ \ \oplus_k\ (f\ xr)) \qquad (6)$$

Since $f\ ((xl_1\ \mathbin{+\!\!+}\ xl_2)\ \mathbin{+\!\!+}\ xr)\ = f\ (xl_1\ \mathbin{+\!\!+}\ (xl_2 \mathbin{+\!\!+} xr))$, after unifying terms at RHS of both definition 5 and 6, we have

$$h_0\ [a]\ =\ g_0\ a\ z$$
$$h_0\ (xl_1 \mathbin{+\!\!+} xl_2)\ = h_0\ xl_1\ \oplus_0\ h_1\ xl_1\ \oplus_1\ \ldots\ h_k\ xl_1\ \oplus_k\ h_0\ xl_2$$
$$h_i\ [a]\ =\ g_i\ a\ z$$
$$h_i\ (xl \mathbin{+\!\!+} xr)\ z\ =\ (h_i\ xl)\ \oplus_i\ \ldots\ \oplus_{k-1}\ (h_k\ xl) \oplus_k\ (h_i\ xr)\ \forall\ i.\ 1 \leq\ i\ \leq\ k$$

By induction hypothesis, we know

$$h_i\ [a]\ =\ g_i\ a\ z$$

$$h_i\ (xl +\!\!+\ xr)\ z\ =\ (h_i\ xl)\ \oplus_i\ \ldots\ \oplus_{k-1}\ (h_k\ xl) \oplus_k\ (h_i\ xr)\ \forall\, i.\, 1 \le\ i\ \le\ k$$

gives $P(k)$. Thus,

$$\begin{aligned} h_i\ [a]\ &=\ g_i\ a\ z \\ h_i\ (xl +\!\!+\ xr)\ z\ &=\ (h_i\ xl)\ \oplus_i\ \ldots\ \oplus_{k-1}\ (h_k\ xl) \oplus_k\ (h_i\ xr)\ \forall\, i.\, 0 \le\ i\ \le\ k \end{aligned}$$

proves the $P(k+1)$ case and therefore proves the theorem 7. □

5 Examples

In this section, we show some interesting well-`PTyped` sequential programs by giving their `PType`.

5.1 Conditional Form

The following function, f_7, traverses a list and returns an integer.

$$\begin{aligned} &\#(Int, [+, \times], [0, 1]) \\ &f_7\ [a]\ =\ a \\ &f_7\ (a : x)\ =\ \textbf{if}\ (a > 5)\ \textbf{then}\ a\ +\ f_7\ x \\ &\qquad\qquad\qquad\quad \textbf{else}\ a\ \times\ f_7\ x \end{aligned}$$

Let us initialize $\Gamma = \{\ a\ \mapsto\ N,\ x\ \mapsto\ N\}$. The main steps of `PType` inference of the RHS of f_7 is illustrated below:

$$\begin{aligned} &\mathcal{W}_{\|\kappa}(\textbf{if}\ (a > 5)\ \textbf{then}\ a\ +\ f_7\ x\ \textbf{else}\ a\ \times\ f_7\ x, \Gamma) \\ &\quad \mathcal{W}_{\|\kappa}((a > 5), \Gamma) \\ &\quad \Rightarrow\ (N, \{\}) \\ &\quad \mathcal{W}_{\|\kappa}(a\ +\ f_7\ x, app(\{\}, \Gamma)) \\ &\quad \Rightarrow\ (R_{\Pi_1}, \theta_1)\ \text{where}\ \ \Pi_1\ =\ [+]\ \bowtie\ \beta_2;\ \theta_1\ =\ \{\beta_1\ \mapsto\ \Pi_1\} \\ &\quad \mathcal{W}_{\|\kappa}(a\ \times\ f_7\ x, app((\theta_1\ ; \{\}), \Gamma)) \\ &\quad \Rightarrow\ (R_{\Pi_2}, \theta_2)\ \text{where}\ \ \Pi_2\ =\ [\times]\ \bowtie\ \beta_4\ ;\ \theta_2\ =\ \{\beta_3\ \mapsto\ \Pi_2\} \\ &\quad \mathcal{U}(R_{\Pi_1}, R_{\Pi_2}) \\ &\quad \Rightarrow\ \theta_3\ \ \text{where}\ \ \theta_3\ =\ \{\beta_2\ \mapsto\ [\times]\ \bowtie\ \beta_5,\ \beta_4\ \mapsto\ [+]\ \bowtie\ \beta_5\} \\ &\Rightarrow\ (R_{[+,\times]\,\bowtie\,\beta_5},\ \theta_3\ ;\theta_2\ ;\theta_1) \end{aligned}$$

The expression $\textbf{if}\ (a > 5)\ \textbf{then}\ a\ + f_7\ x\ \textbf{else}\ a\ \times\ f_7\ x$ is normalized to an s-value of the following form.

$$\begin{aligned} &(\textbf{if}\ (a > 5)\ \textbf{then}\ a\ \textbf{else}\ 0) + \\ &\qquad\qquad (\textbf{if}\ (a > 5)\ \textbf{then}\ 1\ \textbf{else}\ a)\ \times\ (f_7 x) \end{aligned}$$

This example shows the usefulness of the identities provided for each operator used in the program.

5.2 The mss Problem

Consider a sequential program to find the maximum segment sum (mss) of a list.

$$
\begin{aligned}
&\#(Int,\ [max, +], [0, 0]) \\
&mis\ [a]\ =\ a \\
&mis\ (a : x)\ =\ a\ `max`\ (a\ +\ mis\ x) \\
\\
&mss\ [a]\ =\ a \\
&mss\ (a : x)\ =\ (a\ `max`\ (a\ +\ mis\ x))\ `max`\ mss\ x
\end{aligned}
$$

In the definition of function *mss*, it calls function *mis* with argument x, the recursion parameter. This implies the parallelization of *mss* requires the parallelization of *mis* to be present. Thus, we need to type check function definition of *mis* before that of *mss*. The `PType` inferred for both definition are: $mis\ ::\ R_{[max,+]}$ and $mss :: R_{[max]}$ respectively.

5.3 List and Higher-Order Skeletons

For a function that returns a list, we may use the annotation $\#(List, [+\!\!+, map2], [Nil, Nil])$, where *map*2 is defined as follows:

$$y\ `map2`\ z\ =\ map\ (y\ +\!\!+)\ z$$

Function *map*2 has the following properties:

$$
\begin{aligned}
&\quad\text{— distributive over } +\!\!+ \\
&y\ `map2`\ (zl +\!\!+ zr)\ =\ y\ `map2`\ zl\ +\!\!+\ y\ `map2`\ zr \\
&\quad\text{— semi-associative} \\
&x\ `map2`\ (y\ `map2`\ z)\ =\ (x\ +\!\!+\ y)\ `map2`\ z
\end{aligned}
$$

When *map*2 is used as a binary operator for *scan*, as shown below:

$$
\begin{aligned}
&\#(List, [+\!\!+, map2], [Nil, Nil]) \\
&scan\ [a]\ =\ [\,[a]\,] \\
&scan\ (a : x)\ =\ [\,[a]\,]\ +\!\!+\ ([a]\ `map2`\ (scan\ x))
\end{aligned}
$$

we can infer that *scan* has type $R_{[+\!\!+,\ map2]}$[2].

In this section, we want to show how some higher-order skeletons lead to parallelization with the reasoning of our `PType` system. Besides function *scan* which is one of the higher-order skeleton candidates, *map* and *reduce* are also `PType`able.

[2] Usually programmers may write recursive part of *scan* as: $scan(a : x) = a : ([a]`map2`(scanx))$. Before type-checking, we can transform this to $[a] +\!\!+ ([a]`map2`(scanx))$. Such transformation is trivial and will be done in a pre-processing phase.

$map\ [a]\ f\ =\ [(f\ a)]$
$map\ (a : x)\ f\ =\ [(f\ a)]\ +\!\!+\ map\ x\ f$

$reduce\ [a]\ op\ =\ a$
$reduce\ (a : x)\ op\ =\ a\ `op`\ reduce\ x\ op$

For the simplicity of explanation, readers can assume both parameters f and op in function definition of *map* and *reduce* have `NType`. It is obvious that the `PType`s of function *map* and *reduce* are $R_{[+\!\!+]}$ and $R_{[op]}$ respectively.

Programs proposed via these higher-order skeletons can be transformed to recursive functions through deforestation [30]. This implies our `PType` system can cover at least as many applications as that with these higher-order skeletons.

5.4 Fractal Image Decompression

A fractal image may be encoded by a series of mappings, called affine transformations, which are combinations of scalings, rotations and translations of the coordinate axes. This problem was considered in [15]. For clarity, we only present two those important functions used in the process of fractal image decomposition. The function *tr* takes a list of transformations and applies each of them to a pixel and the function k applies these transformations to a set of pixels with the help of *tr*. The auxilliary function *nodup* removes repeated occurrences of a value in a list effectively generating a set. We assume efficient implementation of *nodup* is provided.

$\#(List, [+\!\!+], [Nil])$
$\#(Set, [union], [Nil])$
$tr\ ::\ [a \rightarrow\ a]\ \rightarrow\ a\ \rightarrow\ [a]$
$tr\ [f]\ p\ =\ [f\ p]$
$tr\ (f : fs)\ p\ =\ [f\ p]\ +\!\!+\ tr\ fs\ p$

$k\ ::\ [[a]]\ \rightarrow\ [a]$
$k\ [a]\ fs\ =\ nodup\ (tr\ fs\ a)$
$k\ (a : x)\ fs\ =\ nodup\ (tr\ fs\ a)\ `union`\ (k\ x)$

Types of *tr* and k are $R_{[+\!\!+]}$ and $R_{[union]}$ respectively.

Parallel code derived:

$tr\ [f]\ p\ =\ [f\ p]$
$tr\ (xl +\!\!+ xr)\ p\ =\ h_1\ xl\ p\ +\!\!+\ tr\ xr\ p$
$h_1\ [f]\ p\ =\ [f\ p]$
$h_1\ (xl +\!\!+ xr)\ p\ =\ h_1\ xl\ p\ +\!\!+\ h_1\ xr\ p$
$k\ [a]\ fs\ =\ nodup\ (tr\ fs\ a)$
$k\ (xl +\!\!+ xr)\ fs\ =\ h_2\ xl\ fs\ +\!\!+\ k\ xr\ fs$
$h_2\ [a]\ fs\ =\ nodup\ (tr\ fs\ a)$
$h_2\ (xl +\!\!+ xr)\ fs\ =\ h_2\ xl\ fs\ +\!\!+\ h_2\ xr\ fs$

6 Extensions

In this section, we show the `PType` system can be extended to cover broader classes of parallelizable code.

6.1 Multiple Recursion Parameters

When a function has multiple recursion parameters, we require the function recurses over all its recursion parameters at the same frequency whose formal definition is as follows.

Definition 9. *A recursive function f is said to recurse over all its recursion parameters at the same frequency if f is in the form*

$$
\begin{array}{l}
f\ [a_1]\ \ldots\ [a_n]\ =\ Ctx[a_1, \ldots, a_n] \\
f\ (a_1 : x_1)\ \ldots\ (a_n : x_n)\ =\ \ldots\ (f\ x_1\ \ldots\ x_n)\ \ldots
\end{array}
$$

where $Ctx[]$ is an arbitrary context and expression $\ldots\ (f\ x_1\ \ldots\ x_n) \ldots$ says any recursive call in the definition should be in the form of $(f\ x_1\ \ldots\ x_n)$.

For example, function definition of *zip* satisfies this requirement as *zip* is defined as follows.

$$
\begin{array}{l}
zip\ [a]\ [b]\ =\ [(a, b)] \\
zip\ (a : x)\ (b : y)\ =\ [(a, b)]\ +\!\!+\ zip\ x\ y
\end{array}
$$

For simplicity of presentation, we use $\overrightarrow{x}$ to express $x_1\ \ldots\ x_n$. In order to handle multiple recursion parameters, we can simply replace all $(f\ x)$ with $(f\ \overrightarrow{x})$ in the type checking rules and the inference algorithm and adding $\{a_1 :: N, \ldots,\ a_n :: N,\ x_1 :: N,\ \ldots,\ x_n :: N\}$ to Γ before type checking the RHS of the function definition. In the case of *zip*, its `PType` is $R_{[+\!\!+]}$.

The correctness of the above type checking strategy can be reasoned as following:
Since the function recurses all its recursion parameters at the same frequency, we can zip all recursion parameters to form one recursion parameter using function *mzip* (multiple zip). Definition of *mzip* is

$$
\begin{array}{l}
mzip\ [a_1]\ \ldots\ [a_n]\ =\ [(a_1, \ldots, a_n)] \\
mzip\ (a_1 : x_1)\ \ldots\ (a_n : x_n)\ =\ (a_1,\ \ldots,\ a_n)\ :\ mzip\ x_1 \ldots x_n
\end{array}
$$

Thus, the parallelizability of the function with multiple recursion parameters (of same recursive frequency) is as same as the function with one recursion parameters.

6.2 Accumulating Parameters

When a function f has accumulating parameters, we shall type check each of them individually to see if they are well-`PType`d before we type check the body

of f. If one of the accumulating parameters is ill-typed, function f is considered ill-typed regardless of the well-PTypedness of the function body. Thus, given a function definition

$$f\ (a_1 : x_1)\ \ldots\ (a_n : x_n)\ p_1\ \ldots\ p_i\ \ldots\ p_n\ =\ e$$

where e is in the form $\ldots\ (f\ \overrightarrow{x}\ e_1\ \ldots\ e_i\ \ldots\ e_n)\ \ldots$ and $p_1\ \ldots\ p_n$ are accumulating parameters, we need to do the following:

1. Extract the context for each accumulating parameter p_i using the function $\mathcal{C}$ defined in Figure 13.
2. Verify that $\forall\, i.\ i\ \in\ \{1, \ldots, n\}$. let e'_i be the result of $\mathcal{C}[\![\ e\ ]\!]_{p_i}$. $\Gamma\ \cup\ \{a_i :: N,\ x_i :: N,\ p_i :: N\ \forall\, i.\ i\ \in \{1, \ldots, n\}\}\ \vdash_{p_i}\ e'_i\ ::\ \rho_i$
3. Verify that $\Gamma\ \cup\ \{a_i :: N,\ x_i :: N,\ p_i :: N\ \forall\, i.\ i\ \in\ \{1, \ldots, n\}\ \vdash_{(f\ \overrightarrow{x})}\ e :: R_S$
4. Apply Normalization rules defined earlier to e with each recursive call as $\underline{\bullet}$ (though they have different accumulating argument) and e'_i with p_i as $\underline{\bullet}$, $\forall\, i.\ i\ \in\ \{1,\ \ldots, n\}$
5. Derive parallel code with the same strategy for each accumulating parameter with p_i as $\underline{\bullet}$.

Function $\mathcal{C}$ takes an expression e and an accumulating parameter p_i as inputs and returns an expression which is the context of the accumulating parameter p_i. The second field of the returned results from $\mathcal{C}$ is to identify redundant context. If the second field is `True`, it means that the expression in the first field *may* be redundant; if the second field is `False`, it means that the expression in the first field *must* be part of the context of the accumulating parameter.

When an expression *may* be redundant, we still need to propagate it. The reason is that if the accumulating parameter depends on the value of such expression, the expression will no longer be redundant. This usually happens in a let-expression. For example, in the following function definition

$$\begin{array}{l} f_{10}\ [a]\ =\ 0 \\ f_{10}\ (a : x)\ c\ =\ \mathbf{let}\ d\ =\ a + 1\ \mathbf{in}\ f_{10}\ x\ (d + c) \end{array}$$

Although d is an expression involving neither a recursive call nor the accumulating parameter c, it is used in the accumulating argument $(d + c)$. Thus,

$$\begin{array}{l} \mathcal{C}[\![\ \mathbf{let}\ d\ =\ a + 1\ \mathbf{in}\ f_{10}\ x\ d + c\ ]\!]_c\ = \\ \qquad \mathbf{let}\ d = a + 1\ \mathbf{in}\ (d + c). \end{array}$$

We use *bracketing problem* to illustrate type-checking of a recursive function definition with accumulating parameters. Bracketing problem is a language recognition problem for determining whether the brackets '(' and ')' in a given string are correctly matched. This problem has a straightforward linear sequential algorithm, in which the string is examined from left to right. A counter is initialized to 0, and increased or decreased as opening and closing brackets are encountered. This definition is taken from [19].

$$
\begin{array}{l}
\#(Bool, [\wedge], [True]) \\
\#(Int, [+, *], [0, 1]) \\
sbp\ x\ =\ sbp'\ x\ 0 \\
sbp'\ [\,]\ c\ =\ c == 0 \\
sbp'\ (a : x)\ c\ = \\
\quad \textbf{if}\ (a == `(`)\ \textbf{then}\ sbp'\ x\ (1 + c) \\
\quad \textbf{else if}\ (a == `)`)\ \textbf{then}\ c > 0\ \wedge\ sbp'\ x\ ((-1) + c) \\
\qquad\qquad\qquad \textbf{else}\ sbp'\ x\ c
\end{array}
$$

Two annotations are needed in order to type-check this program. The new annotation found here defines the operators for type $Bool$. Operator $\wedge$ can be used and its identity (i.e. unit) is $True$. Context for accumulating parameter c is computed as follows.

$$
\begin{array}{rl}
\mathcal{C}[\![\ RHS\ of\ sbp']\!]_c\ = & \textbf{if}\ (a == `(`)\textbf{then}\ 1 + c \\
& \textbf{else if}\ (a == `)`)\textbf{then}\ (-1)\ +\ c \\
& \qquad\qquad \textbf{else}\ c
\end{array}
$$

The `PType` inferred are : $sbp\ ::\ N,\ c\ ::\ R_{[+]}$ and $sbp'\ ::\ R_{[\wedge]}$. Note that, when we type check function body of sbp', the `PType` of c is initialized to N.

6.3 Non-linear Recursion

We extend the `PType` system to cover a subset of non-linear recursion with an additional requirement that $\oplus$ must be commutative. This requirement is often found in parallel works on non-linear recursion.

To parallelize mutually defined non-linear recursive functions, we need to group these functions together forming a tuple and type-check them together. Thus, extending κ to a set which captures the names of different recursive calls is crucial. For self-recursive non-linear recursive functions, the group will become a singleton set.

Given the following mutually defined recursive functions as follows

$$
\begin{array}{l}
f_1\ (a : x)\ =\ e_1 \\
f_2\ (a : x)\ =\ e_2 \\
\ \vdots \\
\ \vdots \\
f_m\ (a : x)\ =\ e_m \\
\text{where}\ e_1\ =\ p_{11}\ \oplus\ (p_{12} \otimes\ f_1\ x)\ \oplus\ \ldots\ \oplus\ (p_{1m}\ \otimes\ f_m\ x) \\
\qquad\quad e_2\ =\ p_{21}\ \oplus\ (p_{12} \otimes\ f_1\ x)\ \oplus\ \ldots\ \oplus\ (p_{2m} \otimes\ f_m\ x) \\
\qquad\quad \vdots \\
\qquad\quad \vdots \\
\qquad\quad e_m\ =\ p_{m1}\ \oplus\ (p_{m2} \otimes\ f_1\ x)\ \oplus\ \ldots\ \oplus\ (p_{mm} \otimes\ f_m\ x) \\
\qquad\quad p_{ij}\ =\ g_{ij}\ a\ (q_j\ x)
\end{array}
$$

Function g_{ij} is an arbitrary function (i.e. an arbitrary context) involving a and $q_j\ x$), $\forall\ i,\ j\ \in\ \{1, \ldots, m\}$. We need to do the following in order to obtain its parallel counterpart.

$$
\begin{array}{l}
\mathcal{C} \;::\; \text{Exp} \;\rightarrow\; \text{Var} \;\rightarrow (\text{Exp},\; \mathit{Bool}) \\
\mathcal{C}[\![\; n \;]\!]_{p_i} \;=\; (n,\; \mathit{True}) \\
\mathcal{C}[\![\; v \;]\!]_{p_i} \;=\; (v,\; \mathit{True}) \\
\mathcal{C}[\![\; f \;\overrightarrow{x}\; e_0 \;\ldots\; e_i \;\ldots\; e_n \;]\!]_{p_i} \;=\; (e_i,\; \mathit{False}) \\
\mathcal{C}[\![\; g \; e_0 \;\ldots\; e_n \;]\!]_{p_i} \;=\; (g \; e_0 \;\ldots\; e_n,\; \mathit{True}) \\
\mathcal{C}[\![\; e_1 \;\oplus\; e_2 \;]\!]_{p_i} \;= \\
\quad \mathit{let}\; (e'_1,\; b_1) \;=\; \mathcal{C}[\![\; e_1 \;]\!]_{p_i} \\
\quad\quad (e'_2,\; b_2) \;=\; \mathcal{C}[\![\; e_2 \;]\!]_{p_i} \\
\quad \mathit{in\; case}\; (b_1,\; b_2)\; \mathit{of} \\
\quad\quad (\mathit{True},\; \mathit{True}) \;\rightarrow\; (e_1 \;\oplus\; e_2,\; \mathit{True}) \\
\quad\quad (\mathit{True},\; \mathit{False}) \;\rightarrow\; (e'_2,\; \mathit{False}) \\
\quad\quad (\mathit{False},\; \mathit{True}) \;\rightarrow\; (e'_1,\; \mathit{False}) \\
\quad\quad (\mathit{False},\; \mathit{False}) \;\rightarrow\; \mathit{error} \\
\mathcal{C}[\![\; \mathbf{if}\; e_0 \;\mathbf{then}\; e_1 \;\mathbf{else}\; e_2]\!]_{p_i} \;= \\
\quad \mathit{let}\; (e'_1,\; b_1) \;=\; \mathcal{C}[\![\; e_1 \;]\!]_{p_i} \\
\quad\quad (e'_2,\; b_2) \;=\; \mathcal{C}[\![\; e_2 \;]\!]_{p_i} \\
\quad \mathit{in\; case}\; (b_1,\; b_2)\; \mathit{of} \\
\quad\quad (\mathit{True},\; \mathit{True}) \;\rightarrow\; (\mathbf{if}\; e_0 \;\mathbf{then}\; e_1 \;\mathbf{else}\; e_2,\; \mathit{True}) \\
\quad\quad (\mathit{True},\; \mathit{False}) \;\rightarrow\; (e'_2,\; \mathit{False}) \\
\quad\quad (\mathit{False},\; \mathit{True}) \;\rightarrow\; (e'_1,\; \mathit{False}) \\
\quad\quad (\mathit{False},\; \mathit{False}) \;\rightarrow\; (\mathbf{if}\; e_0 \;\mathbf{then}\; e'_1 \;\mathbf{else}\; e'_2,\; \mathit{False}) \\
\mathcal{C}[\![\; \mathbf{let}\; v \;=\; e_1 \;\mathbf{in}\; e_2 \;]\!]_{p_i} \;= \\
\quad \mathit{let}\; (e'_1,\; b_1) \;=\; \mathcal{C}[\![\; e_1 \;]\!]_{p_i} \\
\quad \mathit{in\; if}\; b_1 \;\mathit{then\; let}\; (e'_2,\; b_2) = \mathcal{C}[\![\; e_2 \;]\!]_{p_i} \\
\quad\quad\quad\quad \mathit{in\; if}\; b_2 \;\mathit{then}\; (\mathbf{let}\; v \;=\; e_1 \;\mathbf{in}\; e_2,\; \mathit{True}) \\
\quad\quad\quad\quad\quad \mathit{else}\; (\mathbf{let}\; v \;=\; e'_1 \;\mathbf{in}\; e'_2,\; \mathit{False}) \\
\quad\quad \mathit{else}\; (e'_1,\; \mathit{False})
\end{array}
$$

Fig. 13. Definition of Context-Derivation Function $\mathcal{C}$.

1. Group function definitions to form

$$(f_1, \ldots, f_m) = (e_1, \ldots, e_m).$$

2. Type check $e_j \;\forall\, j.\, 1 \leq j \leq\; m$ with rules discussed in Figure 8 together with the rule `op-RR`.

$$
\frac{\begin{array}{c} S \;=\; \oplus \;:\; S' \qquad (\mathit{length}\, S) \;\leq 2 \\ \oplus \;\text{is commutative} \\ \mathit{\Gamma} \;\vdash_{\{(f_1\; x), \ldots, (f_m\; x)\}}\; e_1 :: R_S \\ \mathit{\Gamma} \;\vdash_{\{(f_1\; x), \ldots, (f_m\; x)\}}\; e_2 :: R_S \end{array}}{\mathit{\Gamma} \;\vdash_{\{(f_1\; x), \ldots, (f_m\; x)\}}\; (e_1 \;\oplus e_2) :: R_S} \;(\mathtt{op-RR})
$$

3. Type check $(e_1, \ldots,\; e_m)$ with rule `nl` (non-linear).

$$
\frac{\mathit{\Gamma} \;\vdash_{\{(f_1\; x), \ldots, (f_m\; x)\}}\; e_j :: R_S \;\forall\, j.1 \leq\; j \;\leq\; m}{\mathit{\Gamma} \;\vdash_{\{(f_1\; x), \ldots, (f_m\; x)\}}\; (e_1,\; \ldots,\; e_j, \ldots,\; e_m) :: R_S} \;(\mathtt{nl})
$$

4. If $(f_1, \ldots, f_m)$ is well-`PTyped`, normalize each e_j, $\forall\, j.\ 1 \leq j\ \leq\ m$.
5. Generalize the parallel code in [19] to obtain the parallel counter-part for functions $f_1\ \ldots f_m$. This task is trivial.

To illustrate type-checking of non-linear recursive function definitions, we examine the following function definitions that compute Fibonacci Number . This example is taken from [19] (Section 4.2.4) where the corresponding parallel code is also provided.

$$
\begin{array}{l}
\mathit{lfib}\ [\,]\ =\ 1 \\
\mathit{lfib}\ (a : x)\ =\ \mathit{lfib}\ x\ +\ \mathit{lfib}'\ x \\
\mathit{lfib}'\ [\,]\ =\ 0 \\
\mathit{lfib}'\ (a : x)\ =\ \mathit{lfib}\ x
\end{array}
$$

Sketch of type checking is shown below.

$$
\begin{array}{l}
\Gamma\ \cup\ \{a :: N, x :: N\}\ \vdash_{\{(\mathit{lfib}\ x),(\mathit{lfib}'\ x)\}}\ (\mathit{lfib}\ x + \mathit{lfib}'\ x)\ ::\ R_{[+]} \\
\Gamma\ \cup\ \{a :: N, x :: N\}\ \vdash_{\{(\mathit{lfib}\ x),(\mathit{lfib}'\ x)\}}\ (\mathit{lfib}\ x)\ ::\ R_{[\,]} \\
\quad \text{---}\ R_{[\,]}\ <:\ R_{[+]} \\
\Gamma\ \cup\ \{a :: N, x :: N\}\ \vdash_{\{(\mathit{lfib}\ x),(\mathit{lfib}'\ x)\}}\ (\mathit{lfib}\ x)\ ::\ R_{[+]} \\
\Gamma\ \cup\ \{a :: N, x :: N\}\ \vdash_{\{(\mathit{lfib}\ x),(\mathit{lfib}'\ x)\}} \\
\qquad\qquad\qquad\qquad ((\mathit{lfib}\ x\ + \mathit{lfib}'\ x),\ (\mathit{lfib}\ x))\ ::\ R_{[+]}
\end{array}
$$

Thus, *lfib* and *lfib'* can be parallelized.

The reason we need the condition $(\mathit{length}\ S)\ \leq\ 2$ in rule `op-RR` is explained in Theorem 8. The constraint $(\mathit{length}\ S)\ \leq\ 2$ indicates at most two operators in S are allowed to relate recursive calls.

Theorem 8 (Non-linear Recursion). *For any function definition consisting of multiple recursive calls and involving use of more than two operators related to recursive calls (even if they satisfy extended ring property), it is not parallelizable with respect to the theorem of context preservation.*

An informal argument supports Theorem 8 is as follows. After unfolding each recursive calls, the resulting form cannot match back to the original form of the function definition. By the theorem of context preservation, the parallelizability of the function is unknown. Thus, such function is not parallelizable by our system [3].

We have yet to know any work that demonstrate the parallelization of a non-linear recursive function with accumulating parameters.

7 Implementation

We have implemented a prototype of `PType` system in Haskell 98 [20], a widely used purely functional language. The implementation corresponds closely to the

[3] Since there is no completeness proof for the theorem of context preservation, there may possibly exist other methods to parallelize such function.

theory developed in the previous chapters. All the examples presented in Chapter 5 have been verified with this implementation. The expression syntax recognized by the prototype is a subset of the language Haskell.

We have tested our system with sample files of different sizes. The purpose of doing such experiment is to show the scalability of our analysis. The experiments are done on a PC with pentium-4 CPU of 2.00GHz, 512 MB of RAM. The time taken to do (`PType` inference + normalization + parallel code generation) for each application is shown in Table 7. The results in the table verify the time complexity of parallelization process (`PType` inference + normalization + parallel code generation) is $O(n^2)$. (The `PType` inference and parallel code generation both have time complexity of $O(n)$ and normalization has time complexity of $O(n^2)$ which contributes to the overall time complexity.) Further more, we have

Table 2. Statistics

Option	Lines of Code	total computation time (sec)
Sample1	100	0.92
Sample2	200	3.96
Sample3	400	18.18
Sample4	600	39.26
Sample5	800	71.50
Sample6	1000	107.68

tried one benchmark - matrix multiplication whose functional definition is taken from [1]. Its definition in Haskell syntax is in [31]. Total time taken to do type checking, normalization and parallel code generation for matrix multiplication function definition is 0.10sec.

We have provided a web interface to the `PType` system. The URL is *http://loris-1.ddns.comp.nus.edu.sg/˜xun*

8 Related Works

Generic program schemes have been advocated for use in structured parallel programming, both for imperative programs expressed as first-order recurrences through a classic result of [28] and for functional programs via Bird's homomorphism [27]. However, most sequential specifications fail to match up *directly* with these schemes. To overcome this shortcoming, there have been calls to constructively transform programs to match these schemes, but these proposals [25, 15] often require deep intuition and the support of ad-hoc lemmas – making automation difficult. Another approach is to provide more specialized schemes, either statically [24] or via a procedure [19], that can be directly matched to sequential specification.

On the imperative language (e.g. Fortran) front, there have been interests in parallelization of reduction-style loops [12, 14]. By modeling loops via functions,

they noted that function-type values could be reduced (in parallel) via associative function composition. These propagated function-type values could only be efficiently combined if they have a template closed under composition. This requirement is similar to the need to find a common context under recursive call unfolding, *aka.*, context preservation, as described in [4]. Imperative loop corresponds to tail recursion, and can be considered as a special case of linear recursive form that we are dealing with.

Type-based analysis has traditionally been used to support both program safety and optimization. More recently, it has also been used to support program transformations, such as useless variable elimination [21, 2]. However, these two type systems are still based on the evaluation rules of the underlying language.

In contrast to conventional approach, our `PType` system is constructed and proven correct from a set of meta-rules that are used for transforming programs into skeletal forms. We believe such a bond between type meta-system and program transformation is novel, and can help open up more sophisticated type-based analysis for computation at the meta-level.

9 Conclusion

Murray Cole's characterization of algorithmic skeletons [9] through the use of higher-order functions has inspired several prominent research effort into parallel functional programming [26]. These research projects have investigated the importance of *algorithmic skeletons*, as well as their specification and application.

In this paper, we have introduced a novel view to parallelization. To the best of our knowledge, this is the first piece of work that brings together type system and parallelization. Prior to our work, researchers working on type systems do not look into parallelization, and those who work on parallelization do not bother to use type system in their work. By bringing the two fields together, we hope to apply the formalism of type theory to yet another important application domain.

In our design, we have managed to hide the detail mechanisms of type inference/checking under the carpet, and provide a clean and simple interface to the user. The system frees the user from the hassle of performing normalization (which is required in [8]) and parallelizability checking (which is required in [19]). Users only need to provide our system with the extended ring property of the binary operators used in the program. This provision is, in general, the minimum that is required for users working in parallelization.

Besides the type system, our system can automatically generate parallel code from a well-`PType`d sequential program. Due to the space limitation, the derivation detail and its correctness proof are available in [31]. A prototype has been implemented and a web interface has been provided.

The present work has several avenues for further enhancement. So far, we assume a non-recursive function is not considered for parallelization. However, as mentioned in Section 1, functions defined using higher-order skeletons are all non-recursive functions and are parallelizable. Thus, having an enhanced type system that can capture parallelism of both non-recursive functions and recursive

functions is desirable. Furthermore, the idea of using invariants to assist the context preservation of expression, as described in [7], enables the detection of an even broader class of parallelizable functions. To bring this idea into the framework of type inference, it would require a new approach to discover such invariants in an inductive manner.

References

1. J. Backus. 1977 turing award lecture: Can programmign be liberated from the von neumann style? a functional style and its algebra of programs. *CACM*, 21(8):613–641, August 1978.
2. S. Berardi. Pruning simply-typed lambda-terms. *Journal of Logic and Computation*, 6(5):663–681, 1996.
3. G.E. Blelloch, S Chatterjee, J.C. Hardwick, J. Sipelstein, and M. Zagha. Implementation of a portable nested data-parallel language. In *4th Principles and Practice of Parallel Programming*, pages 102–111, San Diego, California (ACM Press), May 1993.
4. W. N. Chin. Synthesizing parallel lemma. In *Proc of a JSPS Seminar on Parallel Programming Systems, World Scientific Publishing*, pages 201–217, Tokyo, Japan, May 1992.
5. W. N. Chin. Towards an automated tupling strategy. In *Proc. Conference on Partial Evaluation and Program Manipulation*, pages 119–132, Copenhagen, Denmark, June 1993.
6. W.N. Chin, J. Darlington, and Y. Guo. Parallelizing conditional recurrences. In *2nd Annual EuroPar Conference*, Lyon, France, (LNCS 1123) Berlin Heidelberg New York: Springer, August 1996.
7. W.N. Chin, S.C Khoo, Z. Hu, and M. Takeichi. Deriving parallel codes via invariants. In *International Static Analysis Symposium (SAS2000)*, Santa Barbara, California, June 2000. LNCS 1824, Springer Verlag.
8. W.N. Chin, A. Takano, and Z. Hu. Parallelization via context preservation. In *IEEE Intl Conference on Computer Languages*, Chicago, U.S.A., May 1998. IEEE CS Press. `http://www.comp.nus.edu.sg/~chinwn/iccl98.ps`.
9. M. Cole. Parallel programming with list homomorphisms. *Parallel Processing Letters*, 5(2), 1995.
10. P. Cousot and R. Cousot. Systematic design of program transformation frameworks by abstract interpretation. In *29th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Programming Languages*, Portland, OR, USA, January 2002. ACM Press, New York, USA.
11. L. Damas and R. Milner. Principal type-schemes for functional programs. In *9th Annual ACM Symposium on Principles of Programming Languages*, pages 207–212. ACM Press, 1982.
12. A.L. Fischer and A.M. Ghuloum. Parallelizing complex scans and reductions. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 135–136, Orlando, Florida, ACM Press, 1994.
13. M. Fokkinga. Law and order in algorithmics. *PhD thesis, National University of Twente*, 1992.
14. A.M. Ghuloum and A.L. Fischer. Flattening and parallelizing irregular applications, recurrent loop nests. In *3rd ACM Principles and Practice of Parallel Programming*, pages 58–67, Santa Barbara, California, ACM Press, 1995.

15. Z.N. Grant-Duff and P. Harrison. Parallelism via homomorphism. *Parallel Processing Letters*, 6(2):279–295, 1996.
16. K. Hammond and G. Michaelson. *Research Directions in Parallel Functional Programming*. Springer Verlag, 1999.
17. Z. Hu, H. Iwasaki, and M Takeichi. Formal derivation of efficient parallel programs by construction of list homomorphisms. *ACM Transactions on Programming Languages and Systems*, 19(3):444–461, 1997.
18. Z. Hu, H. Iwasaki, and M. Takeichi. Tupling calculation eliminates multiple data traversals. In *ACM SIGPLAN International Conference on Functional Programming*, pages 164–175, Amsterdam, The Netherlands, June 1997. ACM Press.
19. Z. Hu, M. Takeichi, and W.N. Chin. Parallelization in calculational forms. In *25th Annual ACM Symposium on Principles of Programming Languages*, pages 316–328, San Diego, California, January 1998. ACM Press.
20. S. P. Jones, J. Hughes, and et al. Report on the programming language haskell 98, a non-strict, purely functional language. February 1985.
21. N. Kobayashi. Type-based useless variable elimination. In *ACM Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 84–93, Boston, Massachusett, January 2000.
22. F. Nielson, H.R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer Verlag, 1999.
23. B. C. Pierce. *Types and Programming Languages*. The MIT Press Cambridge, Massachusetts, 2002.
24. SS. Pinter and RY. Pinter. Program optimization and parallelization using idioms. In *ACM Principles of Programming Languages*, pages 79–92, Orlando, Florida, ACM Press, 1991.
25. Paul Roe. *Parallel Programming using Functional Languages (Report CSC 91/R3)*. PhD thesis, University of Glasgow, 1991.
26. SkeletonHomepage. http://www.dcs.ed.ac.uk/home/mic/skeletons.html.
27. D. Skillicorn. Architecture-independent parallel computation. *IEEE Computer*, 23(12):38–50, December 1990.
28. Harold S. Stone. Parallel tridiagonal equation solvers. *ACM Transactions on Mathematical Software*, 1(4):287–307, 1975.
29. D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. Til:a type-directed optimizing compiler for ml. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1996.
30. P. Wadler. Deforestation: Transforming programs to eliminate trees. In *ESOP '88, 2nd European Symposium on Programming*. Springer, March 1988.
31. N. Xu, S.-C. Khoo, W.-N. Chin, and Z. Hu. A type-based approach to parallelization. *Technical Report, National University of Singapore http://www.comp.nus.edu.sg/˜xun/research/parallel.ps*, July 2003.