A TYPE-BASED APPROACH TO PARALLELIZATION

XU NA

(B.Comp.(Hons.), NUS)

A THESIS SUBMITTED

FOR THE DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF SINGAPORE

2003

To My Parents

Acknowledgements

During the course of this research I benefited enormously from the knowledge and expertise of my adviser, Associate Professor Khoo Siau Cheng. I especially thank him for his patient guidance on constructing the `PType` system as well as proving its correctness. I also thank him for his hospitality and kindness in general.

I would like to thank Associate Professor Chin Wei Ngan and Associate Professor Hu Zhenjiang for giving me a chance of research collaboration at Information Processing Laboratory of Department of Mathematical Informatics in the University of Tokyo for six weeks. I thank Hu for hosting my visit and for his patience in teaching me the concept of context preservation and for valuable discussions leading to the idea of `PType` system.

I am also grateful to Chin Wei Ngan, Neil Jones, Martin Rinard and Alan Edelman for sitting in my presentation on `PType` system and giving valuable feedback.

Thanks to all my colleagues sitting in PLS (Programming Language and System) laboratory II of SoC for their moral support. Especially, I thank Corneliu Popeea and Razvan Musaloiu-E for helping me resolve some system problems.

Contents

Summary

Parallel functional programming plays an important role in parallel programming [20]. Type system has significant impact on program analysis [33]. 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.

List of Tables

List of Figures

Chapter 1

Introduction

Many computational-intensive 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 far from trivial, parallel programming is much, much harder as there are many more things to consider during debugging. A more desirable way for parallel programming is to start with a sequential program, then test and debug the sequential program and systematically transform the sequential program to parallel code.

However, systematic parallelization of sequential programs still remains a major challenge in parallel computing. Particularly challenging are the restructuring of programs which make 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 program restructuring properties that allow parallelism to be provided. 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.

However, these higher-order skeletons are non-trivial to use as they require the associative property to be present in their combining (conquering) operators. Before a programmer can use them, he/she must manually fit the stated problem into the skeleton program structure. Often, this task is non-trivial as combining operators which return multiple results may be required.

For example, consider the polynomial function definition:

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

To write it using an associative combining operator, we need to introduce *comb*2 and thus *poly*'s new definition is

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

With the help of combining operator *comb*2 that is associative, we are able to match the above definition to a higher-order skeleton, as shown below.

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

A potential problem is how users are able to come out with an associative operator, such as *comb*2, from a naive sequential program. Without this extra effort, higher-order skeletons remain untenable.

In this thesis, we claim that parallelization of sequential code can be automated to a great extent through automatic program analysis and transformation.

Specifically, type inference system can be employed to detect the parallelizability of sequential code, and reduce the effort required from users. This automatic parallelization technique currently apply to a first-order typed functional language with strict semantics.

To demonstrate the feasibility of our thesis, we design a system that allows users to write recursive functions in their natural setting whenever they feel hard to express the definition using higher-order skeletons. Our approach is complementary to the traditional higher-order skeleton approach. Specifically, traditional approach encourages users to write *non*-recursive functions using higher-order skeletons while we parallelize recursive functions obtained from user-defined program.

We view parallelization as a meta-level transformation from sequential program to parallel program. As there is a big difference in the control structure of these two kinds of programs, we twist a parallelization problem to a problem of transforming sequential program to a special form (which is still a sequential program) and provide a direct mapping from this special form to parallel program.

In this thesis, we call the special form *skeleton value*. We show that skeleton values are desired special forms by

1. proving all skeleton values are parallelizable (in Section 2.3) through the concept of *context preservation* whose definitions will be explained in Section 2.1.1,
2. providing parallel code for each skeleton value (in Chapter 4) and
3. showing the correctness of the parallel code provided (in Section 4.3).

Our transformation rules are called *normalization rules*, and a type system is used for detecting parallelism of a sequential program before normalizing the program. We call this new type system the `PType` system, where `PType` denotes *Parallelizable Type*. 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. Consequently, the `PType` system is built upon meta-level program transformation, instead of usual semantic evaluation. If a sequential program is well-`PType`d, it is parallelizable and parallel code can be automatically generated by our system.

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 2.1.2). Such discovery guarantees the parallelization of the sequential program.

As another example, consider the following variant function,

$$\begin{aligned} &f_1\ [(a, y)]\ c \ = \ a + c \times y \\ &f_1\ ((a, y) : x)\ c \ = \ a + (c \times (y + (f_1\ x\ c))) \end{aligned}$$

Even though the variant appears to be quite different from *poly* definition, the `PType` system is still able to classify f_1 as having the same `PType` as that for *poly*, namely, $R_{[+,\times]}$ resulting a similar parallel code as *poly*. Parallel codes for *poly* and f_1 are shown in Figure 1.1. We can see that the definitions of *prod* and $prod_1$, and the recursive equations of *poly* and f_1 are the same. The main difference lies in the base cases of these two functions.

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

The main contributions of this thesis are as follows:

1. We propose a novel type-based approach to parallelization. We prove the

— parallel code for function *poly*

$$\mathit{poly}\ [a]\ c\ =\ a$$

$$\mathit{poly}\ (\mathit{xl} \mathbin{+\!\!+} \mathit{xr})\ c\ =\ \mathit{poly}\ \mathit{xl}\ c + (\mathit{prod}\ \mathit{xl}\ c) \times (\mathit{poly}\ \mathit{xr}\ c)$$

$$\mathit{prod}\ [a]\ c\ =\ c$$

$$\mathit{prod}\ (\mathit{xl} \mathbin{+\!\!+} \mathit{xr})\ c\ =\ (\mathit{prod}\ \mathit{xl}\ c) \times (\mathit{prod}\ \mathit{xr}\ c)$$

— parallel code for function f_1

$$f_1\ [a]\ c\ =\ a\ +\ c\ \times\ y$$

$$f_1\ (\mathit{xl} \mathbin{+\!\!+} \mathit{xr})\ c\ =\ f_1\ \mathit{xl}\ c + (\mathit{prod}_1\ \mathit{xl}\ c) \times (f_1\ \mathit{xr}\ c)$$

$$\mathit{prod}_1\ [a]\ c\ =\ c$$

$$\mathit{prod}_1\ (\mathit{xl} \mathbin{+\!\!+} \mathit{xr})\ c\ =\ (\mathit{prod}_1\ \mathit{xl}\ c) \times (\mathit{prod}_1\ \mathit{xr}\ c)$$

Figure 1.1: Parallel Codes for *poly* and f_1

soundness of our type system.

2. We construct a type (`PType`) system at the meta level (the `PType` of an expression indicates a program's parallelizability), over a set of program transformation rules. To the best of our knowledge, we have not seen type system being applied at meta-level.

3. Our system can automatically infer `PType` of each function and automatically derive its parallel counterpart as well.

4. We implement a prototype and provide a web interface for user to test out our system.

The outline for the thesis is as follows. In the next chapter, we describe the syntax of the language used in the thesis and give an overview of the `PType` system. Chapter 3 gives a set of typing rules and provides a corresponding inference algorithm. An algorithm for deriving parallel code and its correctness proof are described in Chapter 4. We illustrate how the `PType` system works using examples in Chapter 5. Chapter 6 describes two important extensions (namely, recursive functions with accumulating parameters and non-linear recursive functions) to our `PType` system. Chapter 7 shows some testing result and a web interface of the `PType` system. Related works are discussed in Chapter 8. Finally, we conclude the thesis in Chapter 9.

Chapter 2

Overview

Analogous to Damas-Milner (DM) type system [12], our `PType` system asserts some properties about the subject program. However, this property is not directly related to the program's underlying semantics. Rather, it relates to the program's parallelizability. Thus, 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 the set of normalization rules, a term in the program may be normalized to a special value, which we call *skeletal value*, or *s-value* (skeletal values are algorithmic skeletons that allow parallelism to be provided other than the higher-order skeletons mentioned in Chapter 1). The purpose of `PType` system is thus to identify as many terms as possible that can be normalized to *skeletal values*. Table 2.1 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 of a sequential function definition is $f\ (a : x)\ =\ e$ where e involves recursive call $(f\ x)$. The well-typeness

Table 2.1: Object Level vs. Meta Level

	DM Type System	`PType` System
Subject program	functional	functional
Basic Values	Semantic Value	S-Value
Reduction Rules	Evaluation rules	Normalization rules

of e asserts that e can be normalized to an s-value ; this s-value enables automatic derivation of parallel code for function f. Consequently, the subject-reduction property of `PType` system is illustrated via the set of normalization rules.

2.1 Definitions

In this section, we give defines of some important terms that will be used in later chapters.

2.1.1 Context Preservation

In [10], 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-term has 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* (we call it "*context*" for the rest of the thesis) which extracts away the recursive subterm of the RHS of this equation can be written as $\lambda(\underline{\bullet})\ .\ \alpha\ +\ \beta\ \times\ (\underline{\bullet})$. Hence, the symbol $\underline{\bullet}$ denotes an occurrence of self-recursive call, while α and β denote subterms that do not contain any recursive calls. Such a context is said to be *context preserved 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 $\beta = \beta_1 \times \beta_2$

Theorem 1 (Context Preservation [10, 23]) *Consider a recursive function f of the form $f\ (a : x) = e$ where expression e consists of recursive call(s), if e is context preserved, then function f can be succesfully parallelized.*

This transformation process is called *normalization*. Currently, the normalization process has been ad-hoc, usually guided by some heuristics.

2.1.2 Extended-Ring Property

Our `PType` system aims to detect an *extended-ring property* of the operators used in the subject program, from which context preservation is guaranteed. In this section, we define this property as follows. 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.

2.1.3 Reference to 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 defined below.

Definition 1 (Reference to Recursive Call) *A variable v is said to be a reference to recursive call(s) if an invocation of v leads to an invocation of a recursive call.*

For example, given function definition

$$f_2\,(a : x) = \textbf{let}\; v \;=\; 1 \;+\; f_2\, x \;\textbf{in}\; a \;+\; v$$

variable v is a reference to recursive call since the invocation of v in the expression $a + v$ will invoke recursive call $f_2\, x$. For another example,

$$f_3\,(a : x) \;=\; \textbf{let}\; v \;=\; 1 \;+\; f_3\, x \;\textbf{in}\;\textbf{let}\; u \;=\; 2 \;+\; v \;\textbf{in}\; a \;+\; u$$

The variables u and v are references to recursive call. Variable v denotes an expression that contains a recursive call, while variable u indirectly depends on the recursive call since it contains v.

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 2.1. We require programmers to annotate properties of binary operators used in the program. For example, annotation $\#(\mathit{Int}, [+, \times], [0, 1])$ 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.)

Function definitions in this thesis are written in Haskell syntax. For the rest of the thesis, 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 [23]. $E[\,]$ 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

τ	$\in$	`Typ`	**Type**
n	$\in$	`Cons`	**Constants**
c	$\in$	`Con`	**Data Constructors**
v	$\in$	`Var`	**Variables**
$\oplus$	$\in$	`Op`	**Binary Primitive Operators**
γ	$\in$	`Ann`	**Annotations**
	γ	$::=$	$\#(\tau, [\oplus_1, \ldots, \oplus_n], [\iota_{\oplus 1}, \ldots, \iota_{\oplus n}])$
e, t	$\in$	`Exp`	**Expressions**
	e, t	$::=$	$n \mid v \mid c\, e_1\, \ldots\, e_n \mid e_1 \oplus e_2$
			$\mid f\, e_0\, e_1\, \ldots\, e_n \mid \mathbf{let}\ v = e_1\ \mathbf{in}\ e_2$
			$\mid\ \mathbf{if}\ e_0\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2$
p	$\in$	`Pat`	**Pattern**
	p	$::=$	$v \mid c\, v_1\, \ldots\, v_n$
σ	$\in$	`Prog`	**Programs**
	σ	$::=$	$\gamma^*, (f_i\, p_0\, p_1\, \ldots\, p_n = e)^*\ \forall i.\, 1 \leq i \leq m$
			where f_1 is the main function.

Figure 2.1: Syntax of the source langauge.

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.* [23]). Lastly, $\langle\!\!\langle f\ x\rangle$ is the self-recursive call. 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; the rest of the parameters are considered accumulating parameters. (We shall discuss how to deal with multiple recursion parameters and accumulating parameters in Section 6.1 and Section 6.2 respectively).

- There are only one occurrences 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 [8, 22] as they are known to be homomorphism before normalization.)
- The recursive call $(f\ x)$ (or its references, whose definition is in Section 2.1.3) only appears in the right operand of any associative operator. (If it appears in the left operand, symmetrical typing/normalization rules can be defined.)
- 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 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 at most one reference to a self-recursive call.

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

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

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

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

2.3 Skeletal Values

S-values (defined in Figure 2.2) are possible (also desirable) final results of normalization as s-values 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 2.1.1) to denote a self-recursive call in a function definition.

$$
\begin{array}{lcl}
sv & \in & \textbf{S–Values} \\
sv & ::= & lv \mid \textbf{if}\ \zeta_1\ \textbf{then}\ \zeta_2\ \textbf{else}\ lv \\
lv & ::= & bv \mid \textbf{let}\ v = bv\ \textbf{in}\ \zeta \\
bv & ::= & \underline{\bullet} \mid (\zeta_1 \oplus_1 \ldots \oplus_{n-1} \zeta_n \oplus_n \underline{\bullet}) \\
\zeta & \in & \textbf{C–Exp} \\
\zeta & ::= & C[a,\ (q\ x)] \\
 & & \text{where } C \text{ is arbitrary expression context not} \\
 & & \text{involving references to } \underline{\bullet}
\end{array}
$$

Figure 2.2: 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 2.1.2). 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. Lastly, an s-value of the form **let** $v = bv$ **in** ζ is said to be in *local-abstraction form*. Its recursive call only occurs in the local abstraction. Note the use of ζ in

[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.

the **let**-body, which means that there is no recursive call in the body, neither is there any reference to recursive call indirectly through a variable, such as v. The reason we need to have such s-value is we follow strict semantics and assume we parallelize an unoptimized sequential program. Thus, we need to parallelize the bv in **let** $v = bv$ **in** ζ.

From Theorem 1, 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 2 (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 1 *The proof is based on the definition of context preservation described in [10, 23]; 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:*

$$E[\langle\ A_i\rangle_{i=1}^{m}, \langle q\ \ (y \mathbin{+\!\!+} x)\rangle,\ E[\langle\ B_i\rangle_{i=1}^{m}, \langle q\ \ x\rangle, \underline{\bullet}]]\ = E[\langle\ t'_i\rangle_{i=1}^{m}, \langle\ q\ \ x\rangle, \underline{\bullet}].$$

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

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

Note: we let $C_i[a, (q\ x)] = t_i\ (q\ x)$ and $t_i = \lambda\, z.((g_i\ a)\ z)\ \forall\, 1 \leq\ i \leq\ n$ 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

$$E[\langle\, A_i\rangle_{i=1}^n, \langle\ q\ (y +\!\!+\ x)\rangle,\ E[\langle B_i\rangle_{i=1}^n, \langle\ q\ x\rangle, \underline{\bullet}]]$$
$$= E[\langle\, t'_i\rangle_{i=1}^n, \langle\ q\ x\rangle, \underline{\bullet}]$$

where

$\forall\, 1 \leq\ i \leq\ n.$

$t'_i\ =\ \lambda\, z.\, t_i\,(q\ y\ \oplus_q\ z) \oplus_i\ \ldots\ \oplus_{n-1}\ t_n\,(q\ y\ \oplus_q\ z)\ \oplus\ t_i\ z$

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

This context is preserved since

$$E[\langle\, A_i\rangle_{i=1}^2, \langle\ q\ (y +\!\!+\ x)\rangle,\ E[\langle B_i\rangle_{i=1}^2, \langle\ q\ x\rangle, \underline{\bullet}]]$$
$$= E[\langle\, t'_i\rangle_{i=1}^2, \langle q\ x\rangle, \underline{\bullet}]$$

where

$t'_1\ =\ \lambda\, z.\, A_1\,(q\ \oplus_q\ z)\ \vee\ B_1\, z$

$t'_2\ =\ \lambda\, z.$ **if** $A_1\,(q\ \oplus_q\ z)$ **then** $A_2\ (q\ \oplus_q\ z)$ **else** $B_2\, z$

Case 3b: $e =$ **if** $t_{n+1}\ (q\ x)$ **then** $t_{n+2}\,(q\ x)$

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

$E[\langle\, t_i\rangle_{i=1}^4, (q\ x), \underline{\bullet}] :: R_{[\oplus_1, \oplus_2]} =$

if $t_3\,(q\ x)$ **then** $t_4\,(q\ x)$ **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.

— checking context preservation

$$
\begin{array}{l}
\textbf{if } A_3\,(q\ (y +\!\!+ x))\ \textbf{then } A_4\,(q\ (y +\!\!+ x)) \\
\textbf{else } A_1\,(q\,(y +\!\!+ x))\ \oplus_1\ A_2\,(q\,(y +\!\!+ x)) \oplus_2 \\
\quad (\textbf{if } B_3\,(q\ x)\ \textbf{then } B_4\,(q\ x) \\
\quad\ \textbf{else } B_1\,(q\,x)\ \oplus_1\ B_2\,(q\,x)\ \oplus_2\ \underline{\bullet})
\end{array}
$$

— lifting **if** out

$$
\begin{array}{l}
= \textbf{if } A_3\,(q\ (y +\!\!+ x))\ \textbf{then } A_4\,(q\ (y +\!\!+ x)) \\
\quad \textbf{else } (\textbf{if } B_3\,(q\ x) \\
\qquad\quad \textbf{then } A_1\,(q\,(y +\!\!+ x)\ \oplus_1\ A_2\,(q\,(y +\!\!+ x)) \\
\qquad\qquad\quad \oplus_2\ B_4\,(q\,x) \\
\qquad\quad \textbf{else } A_1\,(q\,(y +\!\!+ x))\ \oplus_1\ A_2\,(q\,(y +\!\!+ x)) \\
\qquad\qquad\quad \oplus_2\ B_1\,(q\,x)\ \oplus_1\ B_2\,(q\,x)\ \oplus_2\ \underline{\bullet})
\end{array}
$$

— merging two **if**s

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

Techniques of handling $(q\ y +\!\!+ x)$ *and* $(q\ x)$ *are shown in Cases 1-3. For the ease of presentation, we use* α_i *and* $\beta_i\ \forall\, 1 \leq\ i\ \leq\ n$ *to denote terms* $t_i\,(q\,x)$*.* $V_b(\underline{\bullet})$ *denotes an s-value of the form* bv *and* $V_b(e)$ *denotes an* bv *where the the recursive call is replaced by* e*.*

Case 4:$e = \textbf{let } v = bv\ \textbf{in } \alpha$

$$(\textbf{let } v_1 =\ V_{b_1}(\underline{\bullet})\ \textbf{in } \alpha_1)\ \circ (\textbf{let } v_2 =\ V_{b_2}(\underline{\bullet})\ \textbf{in } \beta_1)$$

$= \textbf{let } v_1 = V_{b_1}(\textbf{let } v_2 = V_{b_2}(\underline{\bullet}) \textbf{ in } \beta_1) \textbf{ in } \alpha_1$

$= \textbf{let } v_2 = V_{b_2}(\underline{\bullet}) \textbf{ in } (\textbf{let } v_1 = V_{b_1}(\beta_1) \textbf{ in } \alpha_1)$

$= \textbf{let } v_2 = V_{b_2}(\underline{\bullet}) \textbf{ in } \alpha_2$

where $\alpha_2 = \textbf{let } v_1 = V_{b_1}(\beta_1) \textbf{in } \alpha_1$

Case 5: $e = \textbf{if } \alpha_1 \textbf{ then } \alpha_2 \textbf{else}(\textbf{let } v = V_b(\underline{\bullet}) \textbf{ in } \alpha_3)$

$(\textbf{if } \alpha_1 \textbf{ then } \alpha_2 \textbf{else}(\textbf{let } v_1 = V_{b_1}(\underline{\bullet}) \textbf{ in } \alpha_3))\circ$

$(\textbf{if } \beta_1 \textbf{ then } \beta_2 \textbf{else}(\textbf{let } v_2 = V_{b_2}(\underline{\bullet}) \textbf{ in } \beta_3))$

$= \textbf{if } \alpha_1 \textbf{ then } \alpha_2$

$\textbf{else}(\textbf{let} v_1 = V_{b_1}(\textbf{if } \beta_1 \textbf{ then } \beta_2 \textbf{else}(\textbf{let } v_2 = V_{b_2}(\underline{\bullet}) \textbf{ in } \beta_3))$

$\textbf{in } \alpha_3)$

$= \textbf{if } \alpha_1 \textbf{ then } \alpha_2$

$\textbf{else}(\textbf{if } \beta_1 \textbf{ then } (\textbf{let } v_1 = V_{b_1}(\beta_2) \textbf{ in } \alpha_3)$

$\textbf{else}(\textbf{let } v_1 = V_{b_1}(\textbf{let } v_2 = V_{b_2}(\underline{\bullet}) \textbf{in } \beta_3)$

$\textbf{in } \alpha_3))$

$= \textbf{if } (\alpha_1 \lor \beta_1) \textbf{ then}(\textbf{if } \alpha_1 \textbf{ then } \alpha_2 \textbf{ else } \alpha_4)$

$\textbf{else}(\textbf{let } v_1 = V_{b_1}(\textbf{let } v_2 = V_{b_2}(\underline{\bullet}) \textbf{in } \beta_3)$

$\textbf{in } \alpha_3)$

where $\alpha_4 = \textbf{let } v_1 = V_{b_1}(\beta_2) \textbf{in } \alpha_3$

$= \textbf{if } (\alpha_1 \lor \beta_1) \textbf{ then}(\textbf{if } \alpha_1 \textbf{ then } \alpha_2 \textbf{ else } \alpha_4)$

$\textbf{else}(\textbf{let } v_2 = V_{b_2}(\underline{\bullet}) \textbf{ in let } v_1 = V_{b_1}(\beta_3) \textbf{ in } \alpha_3)$

$= \textbf{if } (\alpha_1 \lor \beta_1) \textbf{ then}(\textbf{if } \alpha_1 \textbf{ then } \alpha_2 \textbf{ else } \alpha_4)$

$\textbf{else}(\textbf{let } v_2 = V_{b_2}(\underline{\bullet}) \textbf{ in } \alpha_5)$

where $\alpha_5 = \textbf{let } v_1 = V_{b_1}(\beta_3) \textbf{ in } \alpha_3$

(end of proof)

2.4 Normalization Rules and Normal Forms

Figures 2.3, 2.4, 2.5, 2.6 and 2.7 define normalization rules on expressions. They preserve the denotational semantics of those expressions of which the standard evaluation terminates. As mentioned in section 2.2, the language's semantics is strict.

$$\frac{e \rightsquigarrow e'}{\zeta \oplus_i e \rightsquigarrow \zeta \oplus_i e'} \qquad (\mathtt{N-op})$$

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

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

$$\zeta_1 \oplus \textbf{if } \zeta_0 \textbf{ then } \zeta_2 \textbf{ else } lv \rightsquigarrow \textbf{if } \zeta_0 \textbf{ then } \zeta_1 \oplus \zeta_2 \textbf{ else } \zeta_1 \oplus lv \qquad (\mathtt{N-liftIf1})$$

$$\frac{sv = \textbf{if } \zeta_1 \textbf{ then } \zeta_2 \textbf{ else } lv}{\textbf{let } v = sv \textbf{ in } \zeta_3 \rightsquigarrow \begin{array}{l}\textbf{if } \zeta_1 \textbf{ then let } v = \zeta_2 \textbf{ in } \zeta_3 \\ \quad \textbf{else let } v = lv \textbf{ in } \zeta_3\end{array}} \qquad (\mathtt{N-liftIf2})$$

Figure 2.3: Normalization Rules I

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 belonging to the syntactic categories bv and lv respectively. Sometimes (and especially in the proof), we also use the symbols $V(\underline{\bullet})$ (resp. $V_b(\underline{\bullet})$ and $V_l(\underline{\bullet})$) to represent s-values from the syntactic

$$\zeta_1 \oplus \textbf{let}\ v_1 = bv\ \textbf{in}\ \zeta_2 \rightsquigarrow \textbf{let}\ v_1 = bv\ \textbf{in}\ \zeta_1 \oplus \zeta_2 \qquad (\texttt{N}-\texttt{liftLet})$$

$$\frac{e_1 \rightsquigarrow e_1'}{\textbf{let}\ v = e_1\ \textbf{in}\ e_2 \rightsquigarrow \textbf{let}\ v = e_1'\ \textbf{in}\ e_2} \qquad (\texttt{N}-\texttt{let})$$

$$\frac{e_1 \not\rightsquigarrow \quad e_2 \rightsquigarrow e_2'}{\textbf{let}\ v = e_1\ \textbf{in}\ e_2 \rightsquigarrow \textbf{let}\ v = e_1\ \textbf{in}\ e_2'} \qquad (\texttt{N}-\texttt{in})$$

$$\frac{e_2 \not\rightsquigarrow}{\textbf{let}\ v = \zeta\ \textbf{in}\ e_2 \rightsquigarrow e_2[v \mapsto \zeta]} \qquad (\texttt{N}-\texttt{sub1})$$

$$\frac{e_2 \not\rightsquigarrow \quad v \in FV(e_2)}{\textbf{let}\ v = sv\ \textbf{in}\ e_2 \rightsquigarrow e_2[v \mapsto sv]} \qquad (\texttt{N}-\texttt{sub2})$$

$$\begin{array}{l} \textbf{let}\ v = \textbf{let}\ u = bv\ \textbf{in}\ \zeta_1\ \textbf{in}\ \zeta_2 \\ \rightsquigarrow \textbf{let}\ u = bv\ \textbf{in}\ \textbf{let}\ v = \zeta_1\ \textbf{in}\ \zeta_2 \end{array} \qquad (\texttt{N}-\texttt{elimLet})$$

Figure 2.4: Normalization Rules II

category sv (resp. bv and lv).

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 these rules, we say e is in *normal form*, and denote it by "$e \not\hookrightarrow$".

The rules in Figure 2.3 and Figure 2.4 deal with binary operations and **let** expressions. We assume that all binary operators are strict on both arguments. In `N-op`, we attempt to normalize the right operand of the binary operation first. This is due to our assumption that the self-recursive call (denoted by $\underline{\bullet}$) appears only at the right operand. `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. Note that in the rule `N-semiassoc`, the operator $\oplus'$ is an semi-associative counterpart of $\oplus$; *ie.*, $a \oplus (b \oplus c) = (a \oplus' b) \oplus c$.

`N-liftIf1/2` and `N-liftLet` lift **if** and **let** to the top of the expression respectively. As we assume that the binary operators are strict on both argument, the normalization is correct with respect to the strict semantics. `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-sub1` is applied when e_1 does not contain any recursive call. When e_1 is an s-value and v is a free variable in e_2, rule `N-sub2` is applied. Such unfolding of local definition may cause code duplication, and it should be compensated by a common-subexpression-abstraction phase after the parallelization tranformation, in order to maintain efficiency. Lastly, `N-elimLet` aims to eliminate nested **let**-expressions.

The rules in Figure 2.5, Figure 2.6 and Figure 2.7 handle normalization of conditionals. `N-else` and `N-then` attempt to normalize the branches of the conditional. `N-sv` and `N-grpN` ensure that the self-recursive call occurs only in the alternate branch of the top-level conditional. `N-grpR1` to `N-grpR3` consider 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. Coupled with `N-rmTest`, `N-pushIf1` to `N-pushIf3` effectively eliminate multiple occurrences of self-recursive call. Rules `N-elimLet1`, `N-elimLet2` and `N-elimLet3` attempt to eliminate let-expressions in each of the branches and form nested conditionals which can be further normalized by the other rules.

The relation between normal form and s-values can be described by the following Lemma:

Lemma 3 (Normal Form) *All s-values are in normal form.*

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

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

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

$$\textbf{if}\ \zeta_0\ \textbf{then}\ sv\ \textbf{else}\ \zeta_2 \rightsquigarrow \textbf{if}\ \neg\,\zeta_0\ \textbf{then}\ \zeta_2\ \textbf{else}\ sv \qquad (\texttt{N} - \texttt{sv})$$

$$\begin{array}{ll} \textbf{if}\ \zeta_0\ \textbf{then}\ \zeta_1 & \rightsquigarrow \textbf{if}\ (\zeta_0 \vee \zeta_2) \\ \quad \textbf{else}\ (\textbf{if}\ \zeta_2\ \textbf{then}\ \zeta_3\ \textbf{else}\ lv) & \quad \textbf{then}\ (\textbf{if}\ \zeta_0\ \textbf{then}\ \zeta_1\ \textbf{else}\ \zeta_3) \\ & \quad \textbf{else}\ lv \end{array} \qquad (\texttt{N} - \texttt{grpN})$$

$$\frac{sv_2 = \textbf{if}\ \zeta_1\ \textbf{then}\ \zeta_2\ \textbf{else}\ lv_2}{\begin{array}{ll}\textbf{if}\ \zeta_0 \textbf{then}\ lv_1\ \textbf{else}\ sv_2 \rightsquigarrow & \textbf{if}\ (\neg\,\zeta_0 \wedge \zeta_1)\ \textbf{then}\ \zeta_2 \\ & \textbf{else}\ (\textbf{if}\ \zeta_0\ \textbf{then}\ lv_1\ \textbf{else}\ lv_2)\end{array}} \qquad (\texttt{N} - \texttt{grpR1})$$

$$\frac{sv_1 = \textbf{if}\ \zeta_1\ \textbf{then}\ \zeta_2\ \textbf{else}\ lv_1}{\begin{array}{ll}\textbf{if}\ \zeta_0 \textbf{then}\ sv_1\ \textbf{else}\ lv_2 \rightsquigarrow & \textbf{if}\ (\zeta_0 \wedge \zeta_1)\ \textbf{then}\ \zeta_2 \\ & \textbf{else}\ (\textbf{if}\ \zeta_0\ \textbf{then}\ lv_1\ \textbf{else}\ lv_2)\end{array}} \qquad (\texttt{N} - \texttt{grpR2})$$

$$\frac{\begin{array}{c} sv_1 = \textbf{if}\ \zeta_1\ \textbf{then}\ \zeta_2\ \textbf{else}\ lv_1 \\ sv_2 = \textbf{if}\ \zeta_3\ \textbf{then}\ \zeta_4\ \textbf{else}\ lv_2 \end{array}}{\begin{array}{ll}\textbf{if}\ \zeta_0\ \textbf{then}\ sv_1\ \textbf{else}\ sv_2 \rightsquigarrow & \textbf{if}\ (\zeta_0 \wedge \zeta_1)\ \textbf{then}\ \zeta_2 \\ & \textbf{else}\ (\textbf{if}\ \zeta_0\ \textbf{then}\ lv_1\ \textbf{else}\ sv_2)\end{array}} \qquad (\texttt{N} - \texttt{grpR3})$$

Figure 2.5: Normalization Rules III

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

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

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

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

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

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

Figure 2.6: Normalization Rules IV

$$\frac{lv_1 \;= \textbf{let}\, v_1 = bv_1 \;\textbf{in}\, \zeta_1 \quad lv_2 \;= \textbf{let}\, v_2 = bv_2 \;\textbf{in}\, \zeta_2}{\begin{array}{l}\textbf{if}\ \zeta_0\ \textbf{then}\ lv_1\ \textbf{else}\ lv_2 \leadsto \textbf{if}\ \zeta_0\ \textbf{then}\ \zeta_1 \\ \qquad\qquad\qquad\qquad\quad \textbf{else}\ (\textbf{if}\ \zeta_0\ \textbf{then}\ bv_1\ \textbf{else}\ lv_2)\end{array}} \qquad (\mathtt{N-elimLet1})$$

$$\frac{lv_1 \;=\; bv_1 \quad lv_2 \;= \textbf{let}\, v_2 = bv_2 \;\textbf{in}\, \zeta_2}{\begin{array}{l}\textbf{if}\ \zeta_0\ \textbf{then}\ lv_1\ \textbf{else}\ lv_2 \leadsto \textbf{if}\ \neg\, \zeta_0\ \textbf{then}\ \zeta_2 \\ \qquad\qquad\qquad\qquad\quad \textbf{else}\ (\textbf{if}\ \neg\, \zeta_0\ \textbf{then}\ bv_2\ \textbf{else}\ bv_1)\end{array}} \qquad (\mathtt{N-elimLet2})$$

$$\frac{lv_1 \;= \textbf{let}\, v_1 = bv_1 \;\textbf{in}\, \zeta_1 \quad lv_2 \;= bv_2}{\begin{array}{l}\textbf{if}\ \zeta_0\ \textbf{then}\ lv_1\ \textbf{else}\ lv_2 \leadsto \textbf{if}\ \neg\, \zeta_0\ \textbf{then}\ lv_2 \\ \qquad\qquad\qquad\qquad\quad \textbf{else}\ lv_1\end{array}} \qquad (\mathtt{N-elimLet3})$$

Figure 2.7: Normalization Rules V

Chapter 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` consists of `NType`s and `RType`s. `PType` expressions are defined in Figure 3.1. (We assume that the input program to our system is well-typed under the Milner-Damas type system. This assumption is reasonable because any programmer should know sequential programming before parallel programming.). The S in Figure 3.1 is a sequence of n binary operators satisfying the extended-ring property. For example, for the self-recursive equation of a function

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

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

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

$$
\begin{array}{lll}
\rho & \in & \texttt{PType} \\
\rho & ::= & \psi \mid \phi \\
\psi & \in & \texttt{NType} \\
\psi & ::= & N \\
\phi & \in & \texttt{RType} \\
\phi & ::= & R_S \\
 & & \text{where } S \text{ is a sequence of operators}
\end{array}
$$

Figure 3.1: Type Expressions

transformed to s-values via the normalization rules defined in Section 2.4. Consequently, 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,

$$[\![N]\!] \ = \ \mathbf{C{-}Exp},$$

where **C-Exp** is defined in Figure 2.2.

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

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

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 only a subset of operators in S, as 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 4 *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 subsumption relation:

Definition 2 (Subsumption of `RType`) *Given two sequences of operators S_1 and S_2, both with extended-ring property. We say R_{S_1}* is subsumed by *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 `PType`s. 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* (defined in Definition 1) 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 `PType` N to the recursive parameters of f.

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

Definition 3 (Well-`PType`dness) *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.*

3.1 Type-Checking Rules

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

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

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

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

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

Figure 3.2: Type-Checking Rules I

The `PType` of a function f is defined as the `PType` of the RHS of its recursive equation. Figure 3.2 and Figure 3.3 list the type-checking rules which are explained below.

As explained earlier, any expression not enclosing any references of the recursive call $(f\ x)$ will be given `NType`. Thus, both constants and variables not referencing

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

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

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

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

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

Figure 3.3: Type-Checking Rules II

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`. Implicitly, 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` yields an expression of `NType`. As we restrict our language to have references to the self-recursive call occurring only at the right operand of binary operation, we only consider the case where the self-recursive call occur on the right operand. In this case, the binary operation yields an `RType` if the right operand is already so, and the binary operator under investigation is part of the sequence S.

We have already assumed that references to the self-recursive call cannot appear 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 $\triangledown_{\texttt{if}}$, which is declared as follows:

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

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. If $e_2 :: N$, we know that e_2 does not contain the variable v (which is a reference to the recursive call). Since the source language has a strict semantics, we still need to parallelize e_1. Thus, the type of e_1 is returned. Otherwise, if e_2 containing v can be found to have the same type as that of e_1, the resulting type follows the type of e_2. These inferences are expressed by the operator $\triangledown_{\texttt{let}}$, defined as follows:

$$\frac{}{\triangledown_{\texttt{let}}(R_S, S, N)} \qquad \frac{}{\triangledown_{\texttt{let}}(R_S, S, R_S)}$$

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, 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 6, Theorem 7 and Theorem 8, expressions that do not contain recursive call or references to recursive calls are not in the scope of these proofs.

3.2.1 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 5 (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* $(\zeta_1 \oplus_1 \ldots \oplus_n \underline{\bullet})$, $(\textbf{let} v = (\zeta_1 \oplus_1 \ldots$

$\oplus_n\ \underline{\bullet})$ **in** $\zeta)$, (**if** ζ_a **then** ζ_b **else** $(\zeta_1 \oplus_1\ \ldots\ \oplus_n\ \underline{\bullet}))$ *or* (**if** ζ_a **then** ζ_b **else** (**let** $v\ =$ $(\zeta_1 \oplus_1\ \ldots\ \oplus_n\ \underline{\bullet})$ *in* $\zeta))$.

Proof 2 *The grammar in Figure 2.2 gives the desired result immediately.*

Theorem 6 (Progress) *If $\Gamma\ \vdash_\kappa\ e :: R_S$, then either e is an s-value or $e \rightsquigarrow\ e'$.*

Proof 3 *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`*,* `N-liftIf1` *or* `N-liftLet` *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 e is an s-value or either rule* `N-grpR1`*,* `N-grpR2`*,* `N-rmTest`*,* `N-pushIf1`*,* `N-pushIf2` *or* `N-pushIf3`*,* `N-elimLet1`*,* `N-elimLet2` *or* `N-elimLet3` *applies. If e_1 is not an* *s-value but e_2 is an s-value, rule* `N-then` *applies. If e_1 is an s-value but e_2 is not, rule* `Nrule-else` *applies. If neither e_1 nor e_2 are s-values,* `N-else` *applies.*

Case `(let-N)` *where $e_2 :: R_S$: If e_2 is an s-value,* `N-sub1` *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, either e is an s-value or either* `N-in`*,* `N-sub2` *or* `N-elimLet` *applies.* ***(end of proof)***

Theorem 7 (Preservation) *If $e :: R_S$ and $e \rightsquigarrow e'$, then $e' :: R_S$*

Proof 4 *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 $\zeta \oplus e$, for some ζ and e. We must also have subderivations with conclusions $\zeta :: N$, $e :: R_S$ and $\oplus \in S$. Now, looking at the normalization rules with such form on the left-hand side in Figure 2.3, we find that there are 5 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 $e' :: R_S$. We can apply rule* `(op)` *to get $(\zeta \oplus_i e') :: R_S$.*

Subcase `N-distr`*: If $e \leadsto e'$ 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. Applying rule* `(op)`*, we 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 $e \leadsto e'$ is derived using* `N-semiassoc`*, 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. Applying rule* `(op)`*, we obtain $(\zeta_1 \oplus' \zeta_2) :: N$, $bv_1 :: R_S$ and $((\zeta_1 \oplus' \zeta_2) \oplus bv_1) :: R_S$.*

Subcase `N-liftIf1`*: Applying induction hypothesis to subderivations, we get $(\textbf{if}\ \zeta_0 :: N\ \textbf{then}\ \zeta_2 :: N\ \textbf{else}\ lv :: R_S) :: R_S$. 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 $(\textbf{if}\ \zeta_0\ \textbf{then}\ \zeta_1 \oplus_i \zeta_2\ \textbf{else}\ \zeta_1 \oplus_i lv) :: R_S$.*

Subcase `N-liftLet`*: Applying induction hypothesis to subderivation, we get $(\textbf{let}\ v_1 = bv :: R_S\ \textbf{in}\ \zeta_2 :: N) :: R_S$. Applying rule* `(op)`*, we get $(\zeta_1 \oplus \zeta_2) :: N$. Applying rule* `(let-R)`*, we get $(\textbf{let}\ v_1 = bv\ \textbf{in}\ \zeta_1 \oplus \zeta_2) :: R_S$*

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 $(\textbf{if}\ e_0 :: N\ \textbf{then}\ e_1 :: N\ \textbf{else}\ e_2 :: R_S) :: R_S$. We find that* `N-else` *and* `N-grpN` *can be derived. We consider each case separately.*

Subcase `N-else`*: Applying induction to subderivation $e_2 \leadsto e_2'$, we get $e_2' :: R_S$. Applying typing rule* `(if)`*, we get $(\textbf{if}\ \zeta\ \textbf{then}\ e_1\ \textbf{else}\ e_2') :: R_S$*

Subcase `N-grpN`*: Applying rule* `(if)`*, we have* $(\zeta_0 \vee \zeta_2) :: N$*. Similarly, we have* $(\textbf{if } \zeta_0 \textbf{then } \zeta_1 \textbf{ else } \zeta_3) :: N$ *as well. Applying induction to subderivation* $(\textbf{if } \zeta_2 \textbf{ then } \zeta_3 \textbf{ else } lv$*, we get* $lv :: R_S$*. Applying rule* `(if)`*, we obtain* $(\textbf{if } (\zeta_0 \vee \zeta_2) \textbf{ then } (\textbf{if } \zeta_0 \textbf{then } \zeta_1 \textbf{ else } \zeta_3) \textbf{ else } lv) :: R_S$*.*

Case `(if)` *where* $\bigtriangledown_{\texttt{if}}(R_S, R_S, N)$*: We find that* `N-sv` *can be applied. Applying induction on subderivations, we get* $\zeta_0 :: N$*,* $sv :: R_S$ *and* $\zeta_2 :: N$*. 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 2.4, we find that there are 8 rules by which* $e \rightsquigarrow e'$ *can be derived. We consider each case separately.*

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

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`*: 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* $(\mathbf{if}\,\zeta_0\,\mathbf{then}\,\zeta_1\,\mathbf{else}\,\zeta_2) :: N$ *and* $(\mathbf{if}\ \zeta_0\ \mathbf{then}\ bv_3\ \mathbf{else}\ bv_4) :: R_S$*. Applying rule* `(op)`*,* $((\mathbf{if}\ \zeta_0\mathbf{then}\,\zeta_1\,\mathbf{else}\,\zeta_2) \oplus_i (\mathbf{if}\ \zeta_0\ \mathbf{then}\ bv_3\ \mathbf{else}\ bv_4)) :: R_S$

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

Subcase `N-pushIf3`*: Similar to that of Subcase* `N-pushIf1`*.*

Subcase `N-elimLet1`*: Apply induction to subderivations, we get* $lv_1 :: R_S$*,* $bv_1 :: R_S$ *and* $lv_2 :: R_S$*. Applying rule* `(if)`*, we have* $(\mathbf{if}\ \zeta_0\mathbf{then}\ bv_1\ \mathbf{else}\ lv_2) :: R_S$*. Applying rule* `(if)` *again, we obtain* $(\mathbf{if}\ \zeta_0\ \mathbf{then}\ \zeta_1\mathbf{else}\ (\mathbf{if}\ \zeta_0\ \mathbf{then}\ bv_1\ \mathbf{else} lv_2)) :: R_S$*.*

Subcase `N-elimLet2`*: Similar to that of Subcase* `N-elimLet1`*.*

Subcase `N-elimLet3`*: Similar to that of Subcase* `N-elimLet1`*.*

Case `(let-N)`*: If the last rule in the derivation is* `(let-N)`*, we know that* e *must have the form* $(\mathbf{let}\ \ v = e_1 :: N\ \mathbf{in} e_2 :: \rho) :: \rho$*. It matches either* `N-in` *or* `N-sub1`*.*

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

Subcase `N-sub1`*: The result is immediate.*

Case `(let-R)` *where* $\bigtriangledown_{\texttt{let}}(R_S, R_S, N)$*: If this is the last rule in the derivation, then we know from the form of this rule that* e *must have the form* $(\textbf{let}\ v = e_1 :: R_S$ $\textbf{in}\ e_2 :: N) :: R_S$*. Now, looking at the normalization rules with such form on the left-hand side in Figure 2.3, we find that there are 3 rules by which* $e \rightsquigarrow e'$ *can be derived. We consider each case separately.*

Subcase `N-liftIf2`*: Apply induction to subderivations, we get* $\zeta_1, \zeta_2, \zeta_3 :: N$ *and* $lv :: R_S$*. Applying rule* `(let-N)`*, we have* $(\textbf{let} v = \zeta_2\ \textbf{in}\ \zeta_3) :: N$ *and* $(\textbf{let}\ v = lv$ $\textbf{in}\ \zeta_3) :: R_S$*. Applying rule* `(if)`*, we obtain* $(\textbf{if}\ \zeta_1\ \textbf{then}\ (\textbf{let}\ v = \zeta_2\ \textbf{in}\ \zeta_3)\ \textbf{else}$ $(\textbf{let}\ v = lv\ \textbf{in}\ \zeta_3)) :: R_S$*.*

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

Subcase `N-elimLet`*: Apply induction to subderivations, we get* $bv :: R_S$ *and* $\zeta_1, \zeta_2 :: N$*. Applying rule* `(let-N)`*, we have* $(\textbf{let}\ v = \zeta_1\ \textbf{in}\ \zeta_2) :: N$*. Applying rule* `(let-R)`*, we have* $(\textbf{let}\ u = bv\ \textbf{in}\ (\textbf{let} v = \zeta_1\ \textbf{in}\ \zeta_2)) :: R_S$*.*

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* $(\textbf{let}\ v = e_1 :: R_S$ $\textbf{in}\ e_2 :: R_S) :: R_S$*. Now, looking at the normalization rules with such form on the left-hand side in Figure 2.3, we find that there are 2 rules by which* $e \rightsquigarrow e'$ *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)`*,* $\textbf{let}\ v = e_1'\ \textbf{in}\ e_2$ *has the same type as* $\textbf{let}\ v = e_1\ \textbf{in}\ e_2$*.*

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-sub2`*: Resulting type follows the type of e_2.* ***(end of proof)***

3.2.2 Strong Normalization

In this section, we prove that a well-`PType`d expression is strong normalizing with respect to the normalization rules in Figure 2.3, 2.4, 2.5, 2.6 and 2.7.

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

The definition of atomic expression is valid because an expression of `NType` does not play a role in normalization and a function application with a recursive call or a reference to recursive call is treated as $\underline{\bullet}$ in the normalization.

Definition 5 (Effective Syntax Tree) *An effective syntax tree is a syntax tree built from atomic expressions.*

Note: the size of an atomic expression in an effective syntax tree is 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 normalization system.

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. 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-sub1`, 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. This fact is obtained from the canonical forms we defined. Before applying normalization rule `N-sub2`, both e_1 and e_2 are in normal form. Since the type of e_2 is R_S, all s-values. According to the canonical form lemma, only one v can appear in e_2. So the rule `N-sub2` does not increase the size of the effective syntax tree.

Theorem 8 (Strong Normalization) *If $e :: \rho$, then $\exists\, e'$ s.t. $e \rightsquigarrow^* e' \not\rightsquigarrow$.*

Proof 5 *We prove it by induction on the size of the effective syntax tree (EST).*
Case `N-op`*: By induction hypothesis, $e_2 \rightsquigarrow e_2'$ decreases the size of the EST, it is obvious that $\zeta \oplus_i e_2 \rightsquigarrow \zeta \oplus_i e_2'$ 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-liftIf1`*: The part that will participate in further normalization is$\zeta_1 \oplus_j lv_1$. Thus, the size of EST is decreased by 4.*

Case `N-liftIf2`*: The RHS of the rule is an s-value.*

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

Case `N-let`*: By induction hypothesis, $e_1 \rightsquigarrow e_1'$ decreases the size of the EST, it is obvious that* **let** e_1 **in** $e_2 \rightsquigarrow$ **let** e_1' **in** e_2 *decreases the size of the EST.*

Case `N-in`*: Similar reasoning as Case* `N-let`*.*

Case `N-sub1` *& Case* `N-sub2`*: It is obvious that the size of the EST is reduced based on the definition of atomic expression.*

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

Case `N-then`*: By induction hypothesis, subexpression* $e_1 \leadsto e_1'$ *decreases the size of the EST. Thus the resulting normalization decreases the size of the EST.*

Case `N-else`*: Similar reasoning as Case* `N-then`*.*

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** lv_1 **else** lv_2*. Thus, the size of the EST is reduced.*

Case `N-grpR2` *& Case* `N-grpR3`*: Similar reasoning as Case* `N-grpR1`*.*

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

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

Case `N-pushIf2` *to Case* `N-pushIf5`*: Similar reasoning as Case* `N-pushIf1`*.*

Case `N-elimLet1`*: The part that will participate in further normalization is* **if** ζ_0 **then**

bv_1 **else** lv_2. *Thus, the size of the EST is reduced.*

Case `N-elimLet2`*: Similar reasoning as Case* `N-elimLet1`*.*

Case `N-elimLet3`*: The RHS of the rule maps rule* `N-elimLet2` *and this rule will not be re-visited at the same program point.*

From the above case analysis, we can see that each normalization rule either help reducing the size of the effective syntax tree or the right hand side of the rule is an s-value. This proves the lemma. ***(end of proof)***

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:

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

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 based on the notion of type subsumption, as defined in Definition 2, with the extension to include 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 3.4 and Figure 3.5. It is expressed as:

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

where $\mathit{\Gamma} \in$ **Env** is type assumption containing mapping between program variables and EPTypes, and $\theta \in$ **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 $\mathit{\Gamma}_{init}$ 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 3.6 and Figure 3.7 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 3.7. Lastly, we define a *ground substitution*, $\theta_{[\,]}$ such that, $\forall\,\beta,\ app(\theta_{[\,]},\beta) = [\,]$, the

```
$\mathcal{W}_{\|\kappa}(n, \mathit{\Gamma}) = (N, \{\})$
$\mathcal{W}_{\|\kappa}(v, \mathit{\Gamma}) =$ $\mathit{if}\ v \neq \kappa\ \mathit{then\ let}\ \rho = \mathit{\Gamma}(v)$
                                  $\mathit{in\ if}\ (\rho == N)\ \mathit{then}\ (N, \{\})\ \mathit{else\ Error}$
                      $\mathit{else}\ (\mathit{\Gamma}(v), \{\})$
$\mathcal{W}_{\|\kappa}(f\ x, \mathit{\Gamma}) =$ $\mathit{if}\ (f\ x) == \kappa\ \mathit{then}\ \mathit{let}\ \beta$ be fresh
                                      $\mathit{in}\ (R_\beta, \{\})$
                              $\mathit{else\ Error}$
$\mathcal{W}_{\|\kappa}(e_1 \oplus e_2, \mathit{\Gamma}) =$
  $\mathit{let}$ $(\rho_1, \theta_1) = \mathcal{W}_{\|\kappa}(e_1, \Gamma)$
       $(\rho_2, \theta_2) = \mathcal{W}_{\|\kappa}(e_2, \mathit{app}(\theta_1, \Gamma))$
  $\mathit{in}$ $\mathit{case}\ (\mathit{app}(\theta_2, \rho_1), \rho_2)\ \mathit{of}$
       $(N, N) \rightarrow (N, \theta_2\,;\theta_1)$
       $(N, R_S) \rightarrow \mathit{let}\ \beta'$ be fresh
                     $S' = [\oplus] \bowtie \beta'$
                     $\theta_3 = \mathcal{U}(R_S, R_{S'})$
                 $\mathit{in}\ (\mathit{app}(\theta_3, R_S), \theta_3\,;\theta_2\,;\theta_1)$
       $(\_, \_) \rightarrow \mathit{Error}$
```

Figure 3.4: Type Inference Algorithm - I

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 $\mathit{\Gamma}$ and ρ if and only if it is a ground substitution that covers $\mathit{\Gamma}$ and ρ. We can safely say $\theta_{[\,]}$ is a ground validation of $\mathit{\Gamma}$ and ρ in all the cases.

The algorithm resembles a typical type inference algorithm. The detail can be found in standard program analysis textbook [28].

```
$\mathcal{W}_{\|\kappa}(\textbf{if}\ e_0\ \textbf{then}\ e_1\ \textbf{else}\ e_2, \mathit{\Gamma}) \ =$
  $let\ (\rho_0, \theta_0) \ = \ \mathcal{W}_{\|\kappa}(e_0, \Gamma)$
      $(\rho_1, \theta_1) \ = \ \mathcal{W}_{\|\kappa}(e_1, app(\theta_0, \Gamma))$
      $(\rho_2, \theta_2) \ = \ \mathcal{W}_{\|\kappa}(e_2, app(\theta_1\ ; \theta_0, \Gamma))$
      $\theta \ = \ \theta_2\ ; \theta_1\ ; \theta_0$
  $in\ if\ (app(\theta_2\ ; \theta_1, \rho_0)) \ = \ N$
    $then\ \ case\ (app(\theta_2, \rho_1), \rho_2)\ of$
          $(N, N) \ \rightarrow (N, \theta)$
          $(N, R_S) \ \rightarrow (R_S, \theta)$
          $(R_S,\ N) \ \rightarrow (R_S, \theta)$
          $(R_{S_1}, R_{S_2}) \ \rightarrow let\ \theta' \ = \mathcal{U}(R_{S_1}, R_{S_2})$
                              $in\ \ (app(\theta', R_{S_1}),\ \theta'\ ; \theta)$
    $else\ Error$
$\mathcal{W}_{\|\kappa}(\textbf{let}\ v \ = \ e_1\ \textbf{in}\ e_2, \mathit{\Gamma}) \ =$
   $let\ (\rho_1, \theta_1) \ = \ \mathcal{W}_{\|\kappa}(e_1, \mathit{\Gamma})$
   $in\ if\ (\rho_1 == N)\ \ then$
         $\mathcal{W}_{\|\kappa}(e_2, \{v :: N\} \cup \mathit{\Gamma})$
       $else\ \ let\ (\rho_2, \theta_2) \ = \ W_{\|\ v}\ (e_2, \{v :: \rho_1\} \cup \mathit{\Gamma})$
           $in\ \ if\ (\rho_2 == N)\ then\ \ (app(\theta_2, \rho_1), \theta_2; \theta_1)$
               $else\ \ let\ \theta \ = \ \mathcal{U}(app(\theta_2, \rho_1), \rho_2)$
                   $in\ \ (app(\theta, \rho 2),\ \theta\ ; \theta_2\ ; \theta_1)$
$\mathcal{W}_{\|\kappa}(g\ e, \mathit{\Gamma}) \ =$
   $let\ (\rho, \theta) \ = \ \mathcal{W}_{\|\kappa}(e, \mathit{\Gamma})$
   $in\ if\ (\rho == N\ \&\ g\ \notin\ FV(\kappa))\ then\ (N, \theta)\ else\ Error$
```

Figure 3.5: Type Inference Algorithm - II

```
$\mathcal{U}$ :: ( $\mathbf{EPType}$, $\mathbf{EPType}$) $\rightarrow$ $\mathbf{Sub}$
$\mathcal{U}(N, N) = \{\}$
$\mathcal{U}(R_{\Pi_1}, R_{\Pi_2}) = \mathcal{U}_S(\Pi_1, \Pi_2)$
$\mathcal{U}(\rho_1, \rho_2) = Error$

$\mathcal{U}_S$ :: ( $\mathbf{ESeq}$, $\mathbf{ESeq}$) $\rightarrow$ $\mathbf{Sub}$
$\mathcal{U}_S(\beta, \Pi) = \{\beta \mapsto \Pi\}$
$\mathcal{U}_S(\Pi, \beta) = \{\beta \mapsto \Pi\}$
$\mathcal{U}_S(S_1, S_2 \bowtie \beta)$ = let $S = S_1 \uplus S_2$
                        in if $(S == S_2$ & $S \neq S_1)$
                           then Error
                           else let $T_2 = diff(S, S_2)$
                                in $\{\beta \mapsto T2\}$
$\mathcal{U}_S(S_1 \bowtie \beta, S2) = \mathcal{U}_S(S_2, S_1 \bowtie \beta)$
$\mathcal{U}_S(S_1 \bowtie \beta_1, S_2 \bowtie \beta_2)$ = let $S = S_1 \uplus S_2$
                                    $T_1 = diff(S, S_1)$
                                    $T_2 = diff(S, S_2)$
                                    $\beta$ be fresh
                                 in $\{\beta_1 \mapsto (T_1 \bowtie \beta), \beta_2 \mapsto (T_2 \bowtie \beta)\}$
```

Figure 3.6: Unify Function

3.4 Soundness and Completeness of Inference Algorithm

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

```
$app$ :: ( $\mathbf{Sub}$, $\mathbf{ESeq}$) $\rightarrow$ $\mathbf{ESeq}$
$app(\theta_1 ; \theta_2, \Pi) = app(\theta_1, (app(\theta_2, \Pi)))$
$app(\theta, S) = S$
$app(\theta, \beta) =$ $if$ $((\beta, \Pi) \in \theta)$ $then$ $\Pi$ $else$ $\beta$
$app(\theta, S \bowtie \beta) =$ $let$ $\Pi = \theta\,\beta$
                            $in$ $case$ $\Pi$ $of$
                                $S \rightarrow S$
                                $\beta' \rightarrow S \bowtie \beta'$
                                $S' \bowtie \beta' \rightarrow (S \uplus S') \bowtie \beta'$
  — $app$ is extended naturally to operate on $\mathbf{EPType}$ and $\mathbf{Env}$.

$\uplus$ :: $\mathbf{Seq} \rightarrow \mathbf{Seq} \rightarrow \mathbf{Seq}$
$s_1 \uplus s_2 =$ $case$ $s_1$ $of$ $[\,] \rightarrow s_2$
                           $(x : s_1') \rightarrow s_1' \uplus (insert\ x\ s_2)$

$insert$ :: $\mathbf{Op} \rightarrow \mathbf{Seq} \rightarrow \mathbf{Seq}$
$insert\ \oplus\ [\,] = [\oplus]$
$insert\ \oplus\ [\oplus_1, \ldots, \oplus_n] =$
   $if$ $(\oplus \in [\oplus_1, \ldots, \oplus_n])$ $then$ $[\oplus_1, \ldots, \oplus_n]$
   $else$ $if$ $(\exists\, k \in 0..n : \forall\, i : 1 \leq i \leq k.$
                              $\oplus$ $is\ distributive\ over$ $\oplus_i$ $and$
                          $\forall\, j : (k+1) \leq j \leq n.$
                              $\oplus_j$ $is\ distributive\ over$ $\oplus)$
       $then$ $[\oplus_1, \ldots, \oplus_k, \oplus, \oplus_{k+1}, \ldots, \oplus_n]$
       $else$ $Error$
```

Figure 3.7: Associate Functions to Type Inference Algorithm

Theorem 9 (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 6 *By the definition of $\theta_{[\,]}$, $\theta_{[\,]}$ is a ground validation for all θ. If θ is a ground validation of $app(\theta1; \theta2, \Gamma)$, then $(\theta; \theta1)$ is a ground validation of $app(\theta2, \Gamma)$.*

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 3.2, 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 3.2, 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 3.2, it is immediate that $app(\theta_{[\,]}, \Gamma \cup \{v :: R_S\}) \vdash_\kappa v :: app(\theta_{[\,]}, \Gamma(v))$.*

Case $(f\ x)$: We have $\mathcal{W}_{\|\kappa}(f\ x, \Gamma) = (R_\beta, \{\})$. From rule `(rec)` *in Figure 3.2, 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(θ, Γ), 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(θ_1, Γ) 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$(\theta_{[\,]}; \theta; \theta_2; \theta_1, \rho_1)\}$ $\vdash_v$* (**let** $v\ = e_1$ **in** e_2) :: *app $(\theta_{[\,]};\ \theta; \theta_2; \theta_1, \rho_2)$.* ***(end of proof)***

Theorem 10 (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 7 *The proof is by induction on the shape of the inference tree since the type-checking rules in Figure 3.2 and Figure 3.3 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 ($\textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2$*): 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* $\mathit{\Gamma} \vdash_{\kappa} (\textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2) :: N$*. 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* $\mathcal{W}_{\|\kappa}(e_0, \mathit{\Gamma}) = (N, \{\})$*,* $\mathcal{W}_{\|\kappa}(e_1, \mathit{\Gamma}) = (N, \{\})$ *and* $\mathcal{W}_{\|\kappa}(e_2, \mathit{\Gamma}) = (N, \{\})$*. Clearly* $\mathcal{W}_{\|\kappa}(\textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2, \mathit{\Gamma}) = (N, \{\})$*.*

Subcase `(if)` *where* $\bigtriangledown_{\texttt{if}}(R_S, N, R_S)$*: We have* $\mathit{\Gamma} \vdash_{\kappa} (\textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2) :: R_S$*. If rule* `(if)` *is the last rule applied, we have* $\mathit{\Gamma} \vdash_{\kappa} e_0 :: N$*,* $\mathit{\Gamma} \vdash_{\kappa} e_1 :: N$ *and* $\mathit{\Gamma} \vdash_{\kappa} e_2 :: R_S$*. By induction hypothesis, we have* $\mathcal{W}_{\|\kappa}(e_0, \mathit{\Gamma}) = (N, \theta_0)$*,* $\mathcal{W}_{\|\kappa}(e_1, \mathit{\Gamma}) = (N, \theta_1)$ *and* $\mathcal{W}_{\|\kappa}(e_2, \mathit{\Gamma}) = (R_S, \theta_2)$ *where* $R_S <: R_S$*. Clearly*
$\mathcal{W}_{\|\kappa}(\textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2, \mathit{\Gamma}) = (R_S, \theta_2; \theta_1; \theta_0)$*.*

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

Case ($g\ e$*): We have* $\mathit{\Gamma} \vdash_{\kappa} e :: \rho$*. From rule* `(g)` *in Figure 3.3, we know that* $e :: N$ *and* $g \notin FV(\kappa)$*. By induction hypothesis, we have* $\mathcal{W}_{\|\kappa}(e, \mathit{\Gamma}) = (N, \{\})$*. Clearly* $\mathcal{W}_{\|\kappa}(g\ e, \mathit{\Gamma}) = (N, \{\})$*.*

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

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

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

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

Chapter 4

Algorithm for deriving parallel code

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 [38, 18, 11].

Definition 6 (List Homomorphism) *A function hom is a homomorphism if it satisfies the equations:*

$$hom_{(f,\oplus)}\ [a] \ = \ f\ a$$

$$hom_{(f,\oplus)}\,(xl ++ xr) \;=\; (hom_{(f,\oplus)}\; xl) \;\oplus\; (hom_{(f,\oplus)}\; xr)$$

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 [21, 22], we choose list *mutumorphisms* [16] as our parallel computation model.

Definition 7 (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 [16], being capable of describing most interesting functions. They can be automatically transformed into efficient homomorphisms via tupling calculation as have been intensively studied in [8, 22].

Theorem 11 (Tupling [22]) *Let $h_1, \ldots, h_n$ be mutumorphism as defined in Definition 7. 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:

$$\begin{aligned}
&f\ [a]\ =\ Ctx_1[a, (q\ x)]\\
&f\ (a : x)\ =\ Ctx_2[a, \langle (q\ x)\rangle, (f\ x)]\\
&\quad \text{— } Ctx_1 \text{ and } Ctx_2 \text{ are arbitrary contexts}
\end{aligned}$$

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 code is obvious [23]. For other cases, we consider them one by one. Auxilary functions are defined in Figure 4.1. 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.

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

$$h_a\ [a]\ z\ =\ g_a\ a\ z$$

$$h_a\ (xl +\!\!+\ xr)\ z\ =\ h_a\ xl\ (q\ xr\ \oplus_q\ z)\ \vee\ h_a\ xr\ z$$

$$h_b\ [a]\ z\ =\ g_b\ a\ z$$

$$h_b\ (xl +\!\!+\ xr)\ z$$

$$=\ \mathbf{if}\ h_a\ xl\ (q\ xr\ \oplus_q\ z)\ \mathbf{then}\ h_b\ xl\ (q\ xr\ \oplus_q\ z)$$

$$\mathbf{else}\ h_1\ xl\ (q\ xr\ \oplus_q\ z) \oplus_1\ \ldots\ \oplus_{n-1}$$

$$h_n\ xl\ (q\ xr\ \oplus_q\ z)\ \oplus_n\ h_b\ xr\ z$$

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

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

$$h_i\ (xl +\!\!+\ xr)\ z$$

$$=\ h_i\ xl\ (q\ xr \oplus_q\ z)\ \oplus_i\ \ldots\ \oplus_{n-1}\ h_n xl\ (q\ \ xr \oplus_q\ z)$$

$$\oplus_n\ \ h_i\ xr\ z$$

Figure 4.1: Auxillary functions

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

$$f\ (a : x)\ =\ \mathbf{if}\ g_a\ a\ (q\ x) \mathbf{then}\ g_b\ a\ (q\ x)$$

$$\mathbf{else}\ g_1\ a\ (q\ x)\ \oplus_1\ \ldots\ \oplus_{n-1}\ g_n\ a\ (q\ x) \oplus_n\ \underline{\bullet}$$

function f's parallel version is as follows:

$$f\ [a]\ =\ Ctx_1[a]$$

$$f\ (xl +\!\!+\ xr)\ =$$

$$\mathbf{if}\ h_a\ xl\ (q\ x) \mathbf{then}\ h_b\ xs\ (q\ x)$$

$$\mathbf{else}\ h_1\ xl\ (q\ xr)\ \oplus_1\ \ldots\ \oplus_{n-1}\ h_n\ xl\ (q\ xr)\ \oplus_n\ (f\ xr)$$

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

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

function f's parallel version is as follows.

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

4.3 Correctness of Algorithm for Parallel Code Derivation

In [23], 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 [23] 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 8 *We make use of the associative property of the constructor* $+\!\!+$ *i.e. the fact that* $f\ ((xl_1\ +\!\!+\ xl_2)\ +\!\!+\ xr)\ = f\ (xl_1\ +\!\!+\ (xl_2\ +\!\!+\ xr))$.

$$f\ \ ((xl_1\ \ +\!\!+\ \ xl_2)\ \ +\!\!+\ \ xr)$$
$$= (h_1\ (xl_1\ +\!\!+\ xl_2))\ op_1\ (h_2\ (xl_1\ +\!\!+\ xl_2))\ op_2\ (f\ xr) \tag{4.1}$$

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

— 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.

$$=\ ((h_1\ \ xl_1)\ \ op_1\ \ ((h_2\ \ xl_1)\ \ op_2\ \ (h_1\ \ xl_2)))\ \ op_1$$
$$((h_2\ xl_1)\ op_2\ (h_2\ xl_2)\ op_2\ (f xr)) \tag{4.2}$$

Since $f\ ((xl_1\ +\!\!+\ xl_2)\ +\!\!+\ xr)\ = f\ (xl_1\ +\!\!+\ (xl_2 +\!\!+\ xr))$, *after unifying the RHS of equation 4.1 and 4.2 we have*

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

(end of proof)

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 $f\ (a : x)\ =\ (g1\ a\ (q\ x))\ op_1\ (g2\ a\ (q\ x))\ op_2\ (g3\ a\ (q\ x))\ op_3\ (f\ x)$,

$$
\begin{aligned}
h_1\ (xl \mathbin{+\!\!+} xr) &= (h_1\ xl)\ op_1\ (h_2\ xl)\ op_2\ (h_3\ xl)\ op_3\ (h_1\ xr)\\
h_2\ (xl \mathbin{+\!\!+} xr) &= (h_2\ xl)\ op_2\ (h_3\ xl)\ op_3\ (h_2\ xr)\\
h_3\ (xl \mathbin{+\!\!+} xr) &= (h_3\ xl)\ op_3\ (h_3\ xr)
\end{aligned}
$$

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

Proof 9 *We make use of the associative property of the constructor* $\mathbin{+\!\!+}$ *i.e. the fact that* $f\ ((xl_1 \mathbin{+\!\!+} xl_2) \mathbin{+\!\!+} xr) = f\ (xl_1 \mathbin{+\!\!+} (xl_2 \mathbin{+\!\!+} xr))$ *again.*

$f\ ((xl_1 \mathbin{+\!\!+} xl_2) \mathbin{+\!\!+} xr)$

$$= (h_1\ (xl_1 \mathbin{+\!\!+} xl_2))\ op_1\ (h_2\ (xl_1 \mathbin{+\!\!+} xl_2))\ op_2\ (h_3\ (xl_1 \mathbin{+\!\!+} xl_2))\ op_3\ (f\ xr) \qquad (4.3)$$

$f\ (xl_1 \mathbin{+\!\!+} (xl_2 \mathbin{+\!\!+} xr))$

$= (h_1\ xl_1)\ op_1\ (h_2\ xl_1)\ op_2\ (h_3\ xl_1)\ op_3\ (f\ (xl_2 \mathbin{+\!\!+} xr))$

$= (h_1\ xl_1)\ op_1\ (h_2\ xl_1)\ op_2\ (h_3\ xl_1)\ op_3$
$((h_1\ xl_2)\ op_1\ (h_2\ xl_2)\ op_2\ (h_3\ xl_2)\ op_3\ (f\ xr))$
— 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.

$$\begin{aligned}
&= ((h_1\ xl_1)\ op_1\ (h_2\ xl_1)\ op_2\ (h_3\ xl_1)\ op_3\ (h_1\ xl_2))\ op_1 \\
&\quad ((h_2\ \ xl_1)\ \ op_2\ \ ((h_3\ \ xl_1)\ \ op_3\ \ (h_2\ \ xl_2)))\ op_2 \\
&\qquad\qquad ((h_3\ xl_1)\ op_3\ (h_3\ xl_2))\ op_3\ (f\ \ xr)
\end{aligned} \tag{4.4}$$

Since $f\ ((xl_1\ ++\ xl_2)\ ++\ xr) = f\ (xl_1\ ++\ (xl_2\ ++\ xr))$*, after unifying equation 4.3 and 4.4 we have*

$$\begin{aligned}
h_1\ (xl_1 ++ xl_2) &= (h_1\ xl_1)\ op_1\ (h_2\ xl_1)\ op_2\ (h_3\ xl_1)\ op_3\ (h_1\ xl_2) \\
h_2\ (xl_1 ++ xl_2) &= (h_2\ xl_1)\ op_2\ (h_3\ xl_1)\ op_3\ (h_2\ xl_2) \\
h_3\ (xl_1 ++ xl_2) &= (h_3\ xl_1)\ op_3\ (h_3\ xl_2)
\end{aligned}$$

(end of proof)

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

Theorem 12 *Given function*

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

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

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

and

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

respectively.

Proof 10 *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 12 for n operators used in the function definition. Assume $P(k)$ is true. i.e.*

$P(k)$: Given

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

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

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

and

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

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

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

Let

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

We know

$$f\ (xl_1 +\!\!+ xl_2)\ +\!\!+ xr)\ =\ h_0\ (xl_1 +\!\!+ xl_2) \oplus_0$$

$$(h_1\ (xl_1 +\!\!+ xl_2)\ \oplus_1\ \ldots\ \oplus_{k-1}\ h_k\ (xl_1 +\!\!+ xl_2)\ \oplus_k\ (f\ x)) \tag{4.5}$$

$$\begin{aligned}
& f\ (xl_1 +\!\!+ (xl_2 +\!\!+ xr)) \\
& =\ h_0\ xl_1\ \oplus_0\ (h_1\ xl_1\ \oplus_1\ \ldots\ h_k\ xl_1\ \oplus_k\ (f\ (xl_2 +\!\!+ xr)) \\
& = h_0\ xl_1\ \oplus_0\ h_1\ xl_1\ \oplus_1\ \ldots\ h_k\ xl_1\ \oplus_k\ (h_0\ xl_2\ \oplus_0\ h_1\ xl_1\ \oplus_1\ \ldots\ h_n\ xl_1\ \oplus_k\ (f\ xr)) \\
& : \\
& : \\
& =\ (h_0\ xl_1\ \oplus_0\ h_1\ xl_1\ \oplus_1\ \ldots\ h_k\ xl_1 \oplus_k\ h_0\ xl_2) \oplus_0
\end{aligned}$$

$$((h_1\ xl_1\ \oplus_1\ \ldots\ h_k\ xl_1\ \oplus_k\ h_1\ xl_2)\ \oplus_1\ \ldots\ \oplus_k\ (f\ xr)) \tag{4.6}$$

Since $f\ ((xl_1\ +\!\!+ xl_2)\ +\!\!+ xr)\ = f\ (xl_1\ +\!\!+ (xl_2 +\!\!+ xr))$, after unifying terms at RHS of both definition 4.5 and 4.6, we have

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

By induction hypothesis, we know

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

gives $P(k)$. Thus,

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

proves the $P(k+1)$ case and therefore proves the theorem 12. ***(end of proof)***

Chapter 5

Examples

In this section, we show some interesting well-`PTyped` sequential programs by giving their `PType` and their corresponding parallel code in mutumorphism form. Note that all parallel code derived by our system can be transformed into more efficient code as mentioned in Chapter 4.

5.1 Conditional Form

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

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

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:

$\mathcal{W}_{\|\kappa}(\textbf{if } (a > 5) \textbf{ then } a \ + \ f_7\, x \textbf{ else } a \ \times \ f_7\, x, \Gamma)$

$\quad \mathcal{W}_{\|\kappa}((a > 5), \Gamma)$

$\quad \Rightarrow \ (N, \{\})$

$\quad \mathcal{W}_{\|\kappa}(a \ + \ f_7\, x, app(\{\}, \Gamma))$

$\quad \Rightarrow \ (R_{\Pi_1}, \theta_1)$ where $\Pi_1 \ = \ [+] \ \bowtie \ \beta_2; \ \theta_1 \ = \ \{\beta_1 \ \mapsto \ \Pi_1\}$

$\quad \mathcal{W}_{\|\kappa}(a \ \times \ f_7\, x, app((\theta_1 \, ; \{\}), \Gamma))$

$\quad \Rightarrow \ (R_{\Pi_2}, \theta_2)$ where $\Pi_2 \ = \ [\times] \ \bowtie \ \beta_4 \, ; \ \theta_2 \ = \ \{\beta_3 \ \mapsto \ \Pi_2\}$

$\quad \mathcal{U}(R_{\Pi_1}, R_{\Pi_2})$

$\quad \Rightarrow \ \theta_3$ where $\theta_3 \ = \ \{\beta_2 \ \mapsto \ [\times] \ \bowtie \ \beta_5, \ \beta_4 \ \mapsto \ [+] \ \bowtie \ \beta_5\}$

$\Rightarrow \ (R_{[+,\times] \bowtie \beta_5}, \ \theta_3 \, ;\theta_2 \, ; \theta_1)$

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.

$$(\textbf{if } (a > 5) \textbf{ then } a \textbf{ else } 0) \ + (\textbf{if } (a > 5) \textbf{ then } 1 \textbf{ else } a) \ \times \ (f_7 x)$$

Parallel code derived is given below:

$$
\begin{aligned}
& f_7\, [a] \ = \ \textbf{if } (a > 5) \textbf{ then } a \textbf{ else } 0 \\
& f_7\, (xl +\!\!+\, xr) \ = \ h_1\, xl \ + \ h_2\, xl \ \times \ f_7\, xr \\
& h_1\, [a] \ = \ \textbf{if } (a > 5) \textbf{ then } a \textbf{ else } 0 \\
& h_1\, (xl +\!\!+\, xr) \ = \ h_1\, xl \ + \ h_2\, xl \ \times \ h_1\, xr \\
& h_2\, [a] \ = \ \textbf{if } (a > 5) \textbf{ then } 1 \textbf{ else } a \\
& h_2\, (xl +\!\!+\, xr) \ = \ h_2\, xl \ \times \ h_2\, xr
\end{aligned}
$$

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

5.2 mss Problem

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

$$\#(Int,\ [max, +], [0, 0])$$
$$mis\,[a] \;=\; a$$
$$mis\,(a : x) \;=\; a\,`max`\,(a \;+\; mis\,x)$$

$$mss\,[a] \;=\; a$$
$$mss\,(a : x) \;=\; (a\,`max`\,(a \;+\; mis\,x))\,`max`\,mss\,x$$

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.

Parallel codes derived are given below:

$$mis\,[a] \;=\; a$$
$$mis\,(xl +\!\!+\, xr) \;=\; h_1\,xl\,`max`\,(h_2\,xl \;+\; mis\,xr)$$
$$h_1\,[a] \;=\; a$$
$$h_1\,(xl +\!\!+\, xr) \;=\; h_1\,xl\,`max`\,(h_2\,xl \;+\; h_1\,xr)$$
$$h_2\,[a] \;=\; a$$
$$h_2\,(xl +\!\!+\, xr) \;=\; h_2\,xl \;+\; h_2\,xr$$
$$mss\,[a] \;=\; a$$
$$mss\,(xl +\!\!+\, xr) \;=\; h_3\,xl\,(mis\,xr)`max`\,mss\,xr$$
$$h_3\,[a]\,z \;=\; a\,`max`\,z$$
$$h_3\,(xl +\!\!+\, xr)\,z \;=\; h_3\,xl\,(mis\,xr\,`max`\,z)\,`max`\,h_3\,xr\,z$$

5.3 List and Higher-Order Skeletons

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

$$y \text{ `}map2\text{` } z \;=\; map\ (y +\!\!+)\ z$$

Function $map2$ has the following properties:

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

When $map2$ is used as a binary operator for $scan$, as shown below:

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

we can infer that $scan$ has type $R_{[+\!\!+,\, map2]}$[1].

Parallel codes derived are given below:

$$\begin{aligned}
&scan\ [a] \;=\; [\,[a]\,]\\
&scan\ (xl +\!\!+ xr) \;=\; h_1\ xl +\!\!+ h_2\ xl \text{ `}map2\text{` } scan\ xr\\
&h_1\ [a] \;=\; [\,[a]\,]\\
&h_1\ (xl +\!\!+ xr) \;=\; h_1\ xl +\!\!+ h_2\ xl \text{ `}map2\text{` } h_1\ xr\\
&h_2\ [a] \;=\; [\,[a]\,]
\end{aligned}$$

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

$$h_2\,(xl \mathbin{+\!\!+} xr) \;=\; h_2\,xl\,`map2`\,h_2\,xr$$

In this section, we want to show 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.

$$\begin{aligned} map\,[\,]\,f \;&=\; [\,] \\ map\,(a : x)\,f \;&=\; [(f\ a)] \mathbin{+\!\!+} map\,x\,f \end{aligned}$$

$$\begin{aligned} reduce\,[a]\;op \;&=\; a \\ reduce\,(a : x)\;op \;&=\; a\,`op`\,reduce\;x\;op \end{aligned}$$

Details of dealing with functions with parameters other than the recursion parameter is discussed in Chapter 6. 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.

Above shows that our `PType` system can cover at least as many as applications that using higher-order skeletons. Functions that need accumulating parameters and non-linear recursion are not supported by higher-order skeletons, however, they are supported by the `PType` system. Detailed on this is discussed in Chapter 6.

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 [18]. For clarity, we only present the type-inference and parallelization of those important functions used in the process of fractal image decomposition. 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.

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

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

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

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

Chapter 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 8 *A recursive function f is said to recurse over all its recursion parameters at the same frequency if f is in the form*

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

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.

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

For the simplicity of presentation, we use $\overrightarrow{x}$ (for the rest of the thesis) 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 type 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

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

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 whether the function body is well-typed or not. 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. We extract context for each accumulating parameter p_i using the function $\mathcal{C}$ which is defined in Figure 6.1.

2. $\forall\, i.\, i \in \{1, \ldots, n\}$. 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. $\Gamma \cup \{a_i :: N,\ x_i :: N,\ p_i :: N \;\forall\, i.\, i \in \{1, \ldots, n\} \vdash_{(f\ \vec{x})} e :: R_S$

4. Normalization rules in Figures 2.3, 2.4, 2.5, 2.6 and 2.7 can be applied to e_i' with p_i as $\underline{\bullet}$ $\forall\, i.\, i \in \{1, \ldots, n\}$

5. Same parallel code derivation strategy can be used for each accumulating parameter with p_i as $\underline{\bullet}$.

Function $\mathcal{C}$ (defined in Figure 6.1) takes the 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 for eliminating 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.

For example,

$$\begin{aligned} f_8\,[a]\ c &= a \\ f_8\,(a : x)\ c &= \textbf{if}\ (a > 0)\ \textbf{then}\ a + (f_8\ x\ (1 + c)) \\ &\ \textbf{else}\ a \times (f_8\ x\ (a + c)) \end{aligned}$$

$$\begin{aligned} &\mathcal{C}[\![\ \textbf{if}\ (a > 0)\ \textbf{then}\ a + (f_8\ x\ (1 + c))\ \textbf{else}\ a \times (f_8\ x\ (a + c))]\!]_c = \\ &\quad \textbf{if}\ (a > 0)\ \textbf{then}\ 1 + c\ \textbf{else}\ a + c \end{aligned}$$

Generally speaking, expressions not involving recursive calls tend to be redundant. Particularly, a in expression $(a + (f_8\ x\ (1 + c)))$ is redundant for capturing the context of the accumulating parameter c (similarly for the a at the left operand of $\times$ in expression $(a \times (f_8\, x\, (a + c))))$. The expression involving the accumulating parameter in the recursive call must be part of the context. Specifically, expressions $(1 + c)$ and $(a + c)$ are part of the context of the accumulating parameter c.

For if-expression, if both branches have recursive call (like the above example), the conditional test must be captured. If one of the branches has recursive call, the conditional test is redundant and only the context from the branch with recursive call needs to be captured.

For example, we have function definition

$$f_9\,[a]\; c \;=\; a$$
$$f_9\,(a : x)\; c \;=\; \textbf{if}\;(a > 0)\;\textbf{then}\; a \;+\; f_9\; x\; (1 + c)\;\textbf{else}\; a + c$$

$$C[\![\;\textbf{if}\;(a > 0)\;\textbf{then}\; a \;+\; f_9\; x\; (1 + c)\;\textbf{else}\; a + c\;]\!]_c \;=\; 1 + c$$

Note that although accumulating parameter c appears at else-branch, it is not in a recursive call. Thus, it does not play a role in parallelizing the accumulating parameter and it is a redundant context.

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,

$$f_1 0\,[a] \;=\; 0$$
$$f_1 0\,(a : x)\; c \;=\; \textbf{let}\; d \;=\; a + 1\;\textbf{in}\, f_1 0\; x\; (d + c)$$

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,

$$\mathcal{C}[\![\;\textbf{let}\; d \;=\; a + 1\;\textbf{in}\, f\; x\; d + c\;]\!]_c \;=\; \textbf{let}\; d = a + 1\;\textbf{in}\;(d + c)$$

.

$$
\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} \;=\; \mathit{let}\ (e_1',\ b_1) \;=\; \mathcal{C}[\![\ e_1\ ]\!]_{p_i} \\
\qquad\qquad\qquad\qquad\quad (e_2',\ b_2) \;=\; \mathcal{C}[\![\ e_2\ ]\!]_{p_i} \\
\qquad\qquad\qquad\quad \mathit{in\ case}\ (b_1,\ b_2)\ \mathit{of} \\
\qquad\qquad\qquad\qquad (\mathit{True},\ \mathit{True}) \;\rightarrow\; (e_1\ \oplus\ e_2,\ \mathit{True}) \\
\qquad\qquad\qquad\qquad (\mathit{True},\ \mathit{False}) \;\rightarrow\; (e_2',\ \mathit{False}) \\
\qquad\qquad\qquad\qquad (\mathit{False},\ \mathit{True}) \;\rightarrow\; (e_1',\ \mathit{False}) \\
\qquad\qquad\qquad\qquad (\mathit{False},\ \mathit{False}) \;\rightarrow\; \mathit{error} \\
\mathcal{C}[\![\ \textbf{if}\ e_0\ \textbf{then}\ e_1\ \textbf{else}\ e_2]\!]_{p_i} \;=\; \mathit{let}\ (e_1',\ b_1) \;=\; \mathcal{C}[\![\ e_1\ ]\!]_{p_i} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad (e_2',\ b_2) \;=\; \mathcal{C}[\![\ e_2\ ]\!]_{p_i} \\
\qquad\qquad\qquad\qquad\qquad\quad \mathit{in\ case}\ (b_1,\ b_2)\ \mathit{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{True},\ \mathit{True}) \;\rightarrow\; (\textbf{if}\ e_0\ \textbf{then}\ e_1\ \textbf{else}\ e_2,\ \mathit{True}) \\
\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{True},\ \mathit{False}) \;\rightarrow\; (e_2',\ \mathit{False}) \\
\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{False},\ \mathit{True}) \;\rightarrow\; (e_1',\ \mathit{False}) \\
\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{False},\ \mathit{False}) \;\rightarrow\; (\textbf{if}\ e_0\ \textbf{then}\ e_1'\ \textbf{else}\ e_2',\ \mathit{False}) \\
\mathcal{C}[\![\ \textbf{let}\ v\ =\ e_1\ \textbf{in}\ e_2\ ]\!]_{p_i} \;=\mathit{let}\ (e_1',\ b_1) \;=\; \mathcal{C}[\![\ e_1\ ]\!]_{p_i} \\
\qquad\qquad\qquad\qquad\quad \mathit{in\ if}\ b_1\ \mathit{then\ let}\ (e_2',\ b_2) = \; \mathcal{C}[\![\ e_2\ ]\!]_{p_i} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{in\ if}\ b_2\ \mathit{then}\ (\textbf{let}\ v\ =\ e_1\ \textbf{in}\ e_2,\ \mathit{True}) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{else}\ (\textbf{let}\ v\ =\ e_1'\ \textbf{in}\ e_2',\ \mathit{False}) \\
\qquad\qquad\qquad\qquad\quad \mathit{else}\ (e_1',\ \mathit{False})
\end{array}
$$

Figure 6.1: Definition of Context-Derivation Function $\mathcal{C}$.

6.2.1 Example 1 - *pack*

Consider function *pack* which takes a list of integers and converts them to a single integer. For example, *pack* $[1, 2, 3]\ 0$ will return 123.

$$\begin{aligned} \mathit{pack}\ [\,]\ c &= c \\ \mathit{pack}\ (a : x)\ c &= \mathit{pack}\ x\ (a + 10 \times\ c) \end{aligned}$$

For the accumulating parameter c, we have

$$\Gamma \cup \{a :: N, x :: N\} \vdash_c (a + 10 \times c) :: R_{[+,\times]}.$$

Since the accumulating parameter c is well-`PType`d, we continue type-checking the body of the function definition *pack*. It is obvious that

$$\Gamma \cup \{a :: N, x :: N\} \vdash_{(\mathit{pack}\ x)} \mathit{pack}\ x\ (a + 10 \times c) :: R_{[\,]}.$$

Therefore, function *pack* is well-`PType`d.

6.2.2 Example 2 - Bracketing Problem

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 [23].

$$\begin{aligned} &\#(\mathit{Bool}, [\wedge], [\mathit{True}]) \\ &\#(\mathit{Int}, [+, *], [0, 1]) \\ &\mathit{sbp}\ x\ =\ \mathit{sbp}'\ x\ 0 \\ &\mathit{sbp}'\ [\,]\ c\ =\ c == 0 \\ &\mathit{sbp}'\ (a : x)\ c\ =\ \mathbf{if}\ (a == \text{`(`})\ \mathbf{then}\ \mathit{sbp}'\ x\ (c + 1) \\ &\qquad\qquad \mathbf{else\ if}\ (a == \text{`)`})\ \mathbf{then}\ c > 0\ \wedge\ \mathit{sbp}'\ x\ (c - 1)\ \mathbf{else}\ \mathit{sbp}'\ x\ c \end{aligned}$$

Two annotations are needed in order to type-check this program. The first annotation says that for expression of type *Bool*, operator $\wedge$ can be used and its identity (i.e. unit) is *True*. The second annotation has been explained in earlier chapters.

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

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

In previous chapters, we only consider linear recursion. However, we can 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.

6.3.1 Strategy

Given mutually defined recursive functions as follows

$$\begin{aligned} f_1\ (a : x) \ &= \ e_1 \\ f_2\ (a : x) \ &= \ e_2 \\ &\vdots \\ &\vdots \end{aligned}$$

$$\begin{aligned}
f_m\,(a : x) \;&=\; e_m \\
\text{where } e_1 \;&=\; g_{11}\; a\,(q_1\; x) \;\oplus\; \cdots \oplus\; (g_{1m}\; a\,(q_m\; x) \;\otimes\; f_m\; x) \\
e_2 \;&=\; g_{21}\; a\,(q_4\; x) \;\oplus\; \ldots \;\oplus\; (g_{2m}\; a\,(q_m\; x) \otimes\; f_m\; x) \\
&: \\
&: \\
e_m \;&=\; g_{m1}\; a\,(q_4\; x) \;\oplus\; \ldots \;\oplus\; (g_{mm}\; a\,(q_m\; x) \otimes\; f_m\; x),
\end{aligned}$$

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

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 in Figure 3.2, Figure 3.3 together with the rule `op-RR`.

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

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

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

4. If $(f_1, \ldots, f_m)$ is well-`PType`d, normalize each $e_j \forall\, j.\, 1 \leq\; j\; \leq\; m$.
5. It is trivial to generalize the parallel code in [23] to obtain the parallel counter-part for functions $f_1\; \ldots f_m$.

Theorem 13 (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.*

We need to show that context preservation does not hold for such function definition. 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.

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

Theorem 13 explains why we need the condition $(\mathit{length}\ S) \leq 2$ in rule `op-RR`. The constraint $(\mathit{length}\ S) \leq 2$ indicates at most two operators in S are allowed to relate recursive calls.

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

6.3.2 Example - Fibonacci Number

Function *lfib* computes Fibonacci Number. We take this example from section 4.2.4 of [23] in which corresponding parallel code is provided.

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

Sketch of type checking is shown below.

$$\begin{aligned}
&\Gamma\ \cup\ \{a :: N, x :: N\}\ \vdash_{\{(\mathit{lfib}\ x),(\mathit{lfib}'\ x)\}}\ (\mathit{lfib}\ x + \mathit{lfib}'\ x)\ ::\ R_{[+]}\\
&\Gamma\ \cup\ \{a :: N, x :: N\}\ \vdash_{\{(\mathit{lfib}\ x),(\mathit{lfib}'\ x)\}}\ (\mathit{lfib}\ x) ::\ R_{[\,]}\\
&\quad —\ R_{[\,]} <: R_{[+]}\\
&\Gamma\ \cup\ \{a :: N, x :: N\}\ \vdash_{\{(\mathit{lfib}\ x),(\mathit{lfib}'\ x)\}}\ (\mathit{lfib}\ x) ::\ R_{[+]}\\
&\Gamma\ \cup\ \{a :: N, x :: N\}\ \vdash_{\{(\mathit{lfib}\ x),(\mathit{lfib}'\ x)\}}\ ((\mathit{lfib}\ x + \mathit{lfib}'\ x),\ (\mathit{lfib}\ x))\ ::\ R_{[+]}
\end{aligned}$$

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

Chapter 7

Implementation

We have implemented a prototype of `PType` system in Haskell 98 [25], a widely used purely functional language. The implementation corresponds closely to the 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 provide executable code for two platforms

1. Microsoft WindowsXp (version 2002)

2. Linux Red Hat 7.2

Softwares/Tools used to compile source code include

1. Happy 1.13 (Parser Generator for Haskell)

2. Glasgow Haskell Compiler GHC-5.04.2

7.1 Experiments

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 time taken to do

(PType inference + normalization + parallel code generation) for each application is shown in Table 7.1. 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.)

Table 7.1: 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

Further more, we have tried one benchmark - matrix multiplication whose functional definition is taken from [1] where *ip* is inner product, *distl* is distribute left, *distr* is distribute right, and *trans* is transpose. Its definition in Haskell syntax is in Figure 7.1. Total time taken to do type checking, normalization and parallel code generation for matrix multiplication function definition is 0.10sec.

7.2 PType Online

We have provided a web interface to the PType system. The URL is

http://loris-1.ddns.comp.nus.edu.sg/˜xun

In Figure 7.2, a snapshot of the web interface is shown. Users have two ways to input the program:

```
combine xs ys = case (xs, ys) of
                  ([a], [b]) → [(a ++ b)]
                  ((a : x), (b : y)) → [(a + +b)] ++ (combine x y)
trans1 xs = case xs of
              [a] → [[a]]
              (a : x) → [[a]] ++ trans1 x
trans xs = case xs of
             [a] → trans1 a
             (a : x) → (trans1 a) ‘combine‘ (trans x)
ip xs ys = case (xs, ys) of
             ([a], [b]) → a ∗ b
             ((a : x), (b : y)) → a ∗ b + ip x y
distl xs y = case xs of
               [a] → [(y, a)]
               (a : x) → [(y, a)] ++ distl x y
distr xs y = case xs of
               [a] → [(a, y)]
               (a : x) → [(a, y)] ++ distr x y
mapL f xs = case xs of
              [a] → [f a]
              (a : x) → [f a] ++ (mapL f x)
mapLL f xs = case xs of
               [a] → [f a]
               (a : x) → [f a] ++ (mapLL f x)
matrixmul a b = let b′ = trans b
                    ab = (mapLL (\ (x, y) → distl y x) (distr a b′))
                    f = \ w → (mapL (\ (u, v) → (ip u v)) w)
                in mapLL f ab
```

Figure 7.1: Matrix Multiplication Definition in Haskell

1. upload a file directly from his/her machine by either giving the path of the file or selecting a file using the 'Browse' button;

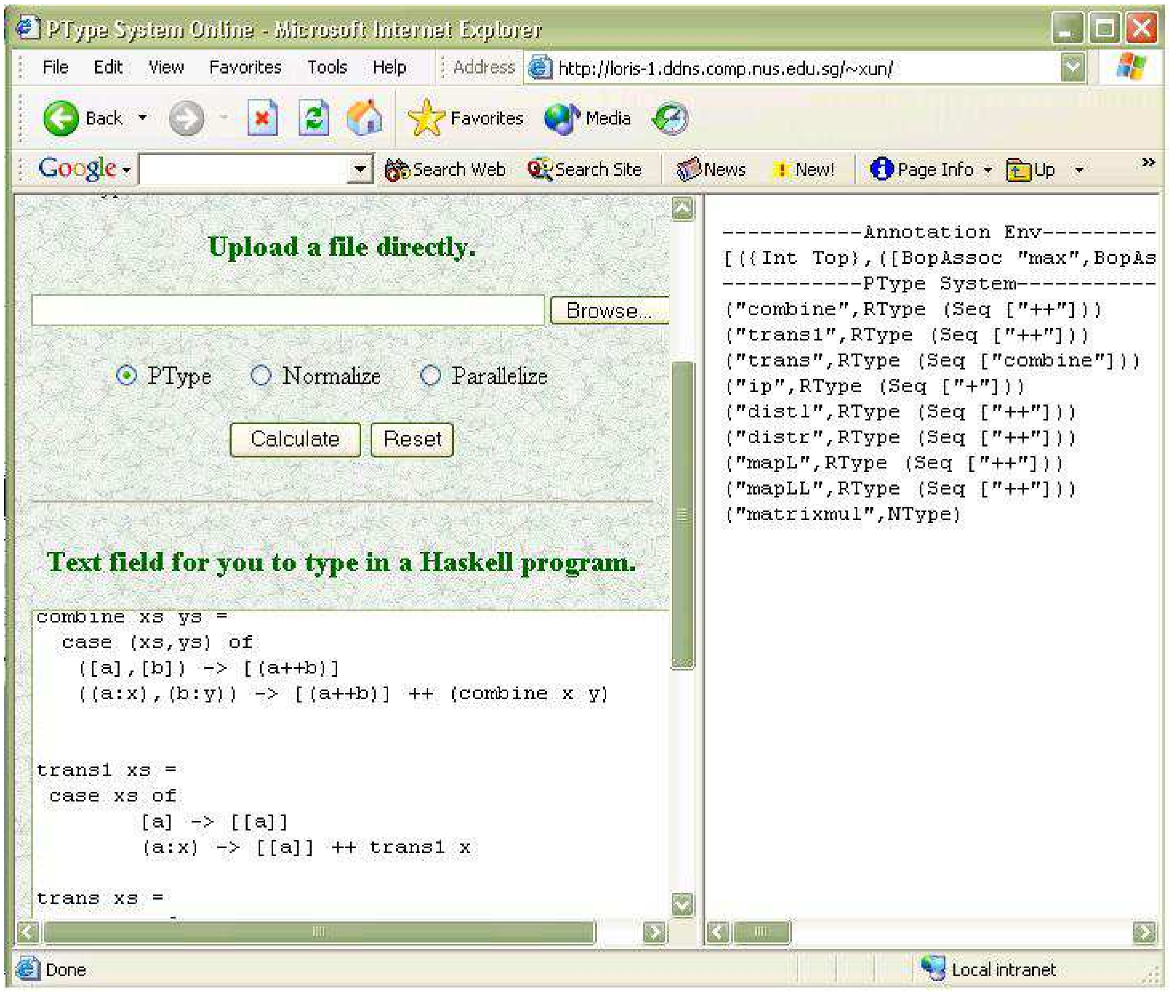
2. type a program in the text field provided.

There are three options a user can choose:

1. **PType** which will infer `PType` for each top-level function;

2. **Normalize** which will give the normal form of each top-level function;

3. **Parallelize** which will show the parallel counter-part of each top-level function.


PType System Online - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://loris-1.ddns.comp.nus.edu.sg/~xun/
Back Favorites Media
Search Web Search Site News New! Page Info Up
Browse...
Calculate Reset
Done
Local intranet

Upload a file directly.

[✓ PType] Normalize Parallelize

Text field for you to type in a Haskell program.

```
combine xs ys =
  case (xs,ys) of
   ([a],[b]) -> [(a++b)]
   ((a:x),(b:y)) -> [(a++b)] ++ (combine x y)


trans1 xs =
 case xs of
        [a] -> [[a]]
        (a:x) -> [[a]] ++ trans1 x

trans xs =
```

```
-----------Annotation Env---------
[((Int Top),([BopAssoc "max",BopAs
-----------PType System-----------
("combine",RType (Seq ["++"]))
("trans1",RType (Seq ["++"]))
("trans",RType (Seq ["combine"]))
("ip",RType (Seq ["+"]))
("distl",RType (Seq ["++"]))
("distr",RType (Seq ["++"]))
("mapL",RType (Seq ["++"]))
("mapLL",RType (Seq ["++"]))
("matrixmul",NType)
```

Figure 7.2: PType System Online

Chapter 8

Related Works

To the best of our knowledge, this is the first piece of work that brings together type system and parallelization together. 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 approach, 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. 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.

In this chapter, we shall describe related work under two broad categories and to compare our work with existing works under each of the two categories.

8.1 Comparison with other Works on Parallelization

In our thesis, we make use of a pure functional language with strict semantics. The parallelization technique we use also make use of implicit models. In this section, we give a brief survey on each of these areas.

8.1.1 Functional versus Imperative

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 [39] and for functional programs via Bird's homomorphism [37]. Unfortunately, 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 [35, 18] often require deep intuition and the support of ad-hoc lemmas – making automation difficult. Another approach is to provide more specialized schemes, either statically [34] or via a procedure [23], 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 [15, 17]. By modelling 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 [7]. Imperative loops correspond to tail recursion, and this can be considered as a special case of linear recursive forms that we are dealing with.

8.1.2 Strict versus Non-strict

There are two broad classes of functional language: strict languages, where all the arguments to a function are evaluated before the function itself is evaluated (e.g. Hope [6], OPAL [13]), and non-strict languages, where arguments are evaluated only if they are needed (e.g. pH [30], Id [29]). Some languages also use hybrid mechanisms, where simple arguments such as integers are evaluated before the function itself, but where complex arguments such as lists and other recursive data structures are not (e.g. Hope [32]).

Strict languages have more predictable execution order, and are therefore more amenable to explicitly controlled parallelism.

8.1.3 Pure versus Impure

The term *purity* has a well-known intuitive definition - excluding implicit side-effects such as assignment, I/O or exceptions. In a parallel setting, side-effects are certainly bad for automatic or semi-automatic parallelization since they inhibit easy program decomposition into parallel tasks and introduce new dependencies between tasks which can be difficult or impossible to disentangle without using explicit parallel control.

Because a purely functional program has no side-effects, it is relatively easy to partition programs so that sub-program can be executed in parallel. Any computation which is needed to produce the result of the program may be run as a separate task. The control dependencies which are implicit in the language serve to enforce any sequential behaviour, and may also be used to limit the creation of excess parallelism.

8.1.4 Implicit versus Explicit

In the context of parallelism, parallel control can be distinguished into several levels, depending on the involvement of programmers.

1. pure implicit approaches
2. restricted implicit approaches
3. controlled approaches
4. explicit approaches

Fully implicit approaches rely on the compiler and run-time system while fully explicit approaches would specify all behavioural details, with overall control in the hands of the almighty programmer. In implicit approaches, the system tries to exploit parallelism that is inherent in the reduction semantics. For example, in the case of data parallelism, we heavily exploit the parallel semantics of a set of special operations. In languages with explicit parallelism, there is often a notion of a standalone process and there are also language constructs for the definition of process systems.

The most implicit approaches, which require least programmer effort for parallelization; are exemplified by pH [30] or evaluation transformers [4, 5]. The pH language is a parallel, eagerly-evaluated variant of Haskell with syntactic provisions for loops, barriers, and I and M storage structures. The arguments to a function are evaluated in parallel and each iteration of the parallel loop-construct is similarly executed as a separate task. Evaluation transformers exploit the properties of non-strict languages, but rely on the system being able to generate good strictness information. Each function is provided with a set of evaluation transformers, one for each formal argument. Each transformer is a forcing operation that will evaluate an actual argument to the extent required by the function, perhaps evaluating it to weak head normal form, or forcing evaluation of the spine of a list. In any context where the function is used, these transformers can be applied in parallel to the corresponding actual arguments.

Restricted implicit approaches match certain language characteristics with desirable program properties. For example, in the skeleton approach, certain patterns of computation are recognized and matched with suitable templates of parallel behaviour. Other examples of restricted implicit approaches include the data parallel language NESL [3], SAC [19] and SISAL [14]. All three languages are strict, purely functional languages, and they obtain parallelism from bulk types such as lists or arrays. NESL provides nestable data parallel operations over lists; SISAL uses parallel loop-constructs over arrays; while SAC uses control parallel with-loops.

Annotation-based approaches may be classified on either side of the implicit/explicit divide. If an annotation is a directive to the compiler, then this is clearly an example of explicit parallelism. If the annotation is a suggestion, however, that may perhaps be checked by the compiler or even ignored entirely, then the construct lies more in the realm of implicit parallelism. Besides annotation-based approaches, controlled approaches include evaluation strategies [40] and first-class schedules [27]. In these systems, functions are used to control parallel behaviour. These functions are higher-order functions that manipulate sequential or parallel program components to yield a more complex parallel program behaviour, but whose definition is entirely within the normal semantics of the sequential programming language.

In explicit approaches, not only is every detail of the parallel execution under the programmer's control, but it must be specified in the parallel program. In principle, this allows a skilled programmer to produce a highly optimized parallel program for target system architecture. This is usually achieved by providing new parallel control constructs to deal with data partitioning, communication, and task placement etc. Concurrent ML [31] and Concurrent Haskell [24] fall into this category.

8.2 Comparison with other Works on Type System

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 [26, 2].

[26] formalizes Useless Variable Elimination (UVE) as a type-preserving, source-to-source transformation that replace some subterms with an void value (). In order to support this particular transformation, Kobayashi formulated a type-system which can safely determine when a particular sub-expression is useless. Coupled with an inference algorithm for this type-system, it is possible to now formulate a useless variable elimination transformation.

The reader may notice similar utility between our `PType` system and UVE type system. However, the UVE type system is still based on the evaluation rules of the underlying language. In contrast to the usual type system based on the underlying language, 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.

Chapter 9

Conclusion

Murray Cole's characterization of algorithmic skeletons [11] through the use of higher-order functions has inspired several prominent research effort into parallel functional programming [36]. These research projects have investigated how important the *algorithmic skeletons* are, as well as, how they could be specified and applied.

In this thesis, we have introduced a fresh alternative view to parallelization - a transition system where initial state is the user-defined sequential program, final state is the algorithmic skeleton and each transition is a transformation rule (specifiable as a normalization rule). We have also introduced a `PType` system, which is a novel type system for detecting parallelism for recursive functions. A well-`PType`d program is guaranteed to be parallelizable (i.e. from initial sequential state, it is guaranteed to reach its final parallel state.)

Besides the type system, we have also given an informal description of the algorithm (i.e. strategy) to automatically obtain parallel code from a normalized well-`PType`d sequential program and proved the correctness of the derivation. A prototype has been implemented and a web interface has been provided for users to test out the system. The system frees the user from the hassle of performing normalization (which is required in [10]) and parallelizability checking (which is required in [23]).

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 Chapter 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 [9], enables the detection of an evenly broader class of parallelizable functions. To bring this idea into the framework of type inference, it requires a new approach to discover such invariants in a inductive manner.

Bibliography

[1] John 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] Stefano 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] G. Burn. Evaluation transformers - a model for the parallel evaluation of functional languages. In *Functional Programming Languages and Computer Architecture*, pages 446–470, Berlin, DE, 1987. Springer-Verlag.

[5] G. Burn. Evaluation transformer model of reduction and its correctness. In *TAPSOFT'91*, pages 458–482, New York, NY, 1991. Springer-Verlag.

[6] R. M. Burstall, D. B. MacQueen, and D. T. Sannella. Hope. *Technical Report CSR-62-80, Edinburgh University*, 1980.

[7] Wei-Ngan Chin. Synthesizing parallel lemma. In *Proc of a JSPS Seminar on Parallel Programming Systems, World Scientific Publishing*, pages 201–217, Tokyo, Japan, May 1992.

[8] Wei-Ngan Chin. Towards an automated tupling strategy. In *Proc. Conference on Partial Evaluation and Program Manipulation*, pages 119–132, Copenhagen, Denmark, June 1993.

[9] 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.

[10] 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`.

[11] M Cole. Parallel programming with list homomorphisms. *Parallel Processing Letters*, 5(2), 1995.

[12] 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.

[13] K. Didrich, A. Fett, C. Gerke, W. Grieskamp, and P. Pepper. Opal: Design and implementation of an algebraic programming language. In *J. Gutknecht, editor, Programming Languages and System Architectures*, Zurich, Switzerland, March 1994. Springer-Verlag LNCS 782.

[14] McGraw J. et al. Sisal: Streams and iteration in a single assignment language: Reference manual version 1.2. *Technical Report Memo M-146, Rev. 1, Lawrence Livermore National Laboratory*, 1985.

[15] 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.

[16] M. Fokkinga. Law and order in algorithmics. *PhD thesis, National University of Twente*, 1992.

[17] 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.

[18] Z.N. Grant-Duff and P. Harrison. Parallelism via homomorphism. *Parallel Processing Letters*, 6(2):279–295, 1996.

[19] C. Grelck. Shared-memory multiprocessor support for sac. In *Proceedings of 10th International Workshop on the Implementation of Functional Languages*, pages 38–54. Springer-Verlag, 1998.

[20] Kevin Hammond and Greg Michaelson. *Research Directions in Parallel Functional Programming*. Springer Verlag, 1999.

[21] 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.

[22] 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.

[23] 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.

[24] Simon Peyton Jones, Andrew , Gordon, and Sigbjorn Finne. Concurrent haskell. In *ACM Symposium on Principles of Programming Languages.* ACM Press, 1996.

[25] Simon Peyton Jones, John Hughes, and et al. Report on the programming language haskell 98, a non-strict, purely functional language. February 1985.

[26] Naoki Kobayashi. Type-based useless variable elimination. In *ACM Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 84–93, Boston, Massachusett, January 2000.

[27] R. Mirani and P. Hudak. First-class schedules and virtual maps. In *Proceedings of FPCA*, pages 78–85, La Jolla, CA, 1995.

[28] F. Nielson, H.R. Nielson, and Chris Hankin. *Principles of Program Analysis.* Springer Verlag, 1999.

[29] R. S. Nikhil. Id (version 90.1) reference manual. *Technical Report CSR Memo 284-2, Lab for Computer Science, MIT*, July 1991.

[30] R. S. Nikhil, Arvind, and J. Hicks. ph language proposal (preliminary), 1st. In *Electronic communication on comp.lang.functional*, 1993.

[31] P. Panangaden and J. H. Reppy. The essence of concurrent ml. In *Nielson, editor, ML with Concurrency: Design, Analysis, Implementation and Application, Monographs in Computer Science*, pages 5–29. Springer-Verlag, 1997.

[32] N. Perry. Hope. *Technical Report IC/FPR/LANG/2.5.1/7 Issue 5, Imperial College, London*, February 1988.

[33] Benjamin C. Pierce. *Types and Programming Languages.* The MIT Press Cambridge, Massachusetts, 2002.

[34] 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.

[35] Paul Roe. *Parallel Programming using Functional Languages (Report CSC 91/R3).* PhD thesis, University of Glasgow, 1991.

[36] SkeletonHomepage. http://www.dcs.ed.ac.uk/home/mic/skeletons.html.

[37] D. Skillicorn. Architecture-independent parallel computation. *IEEE Computer*, 23(12):38–50, December 1990.

[38] D. Skillicorn. The bird-meertens formalism as a parallel model. *NATO ARW Software for Parallel Computation*, June 1992.

[39] Harold S. Stone. Parallel tridiagonal equation solvers. *ACM Transactions on Mathematical Software*, 1(4):287–307, 1975.

[40] P. W. Trinder, K. Hammond, H.-W. Loidl, and S. L. Peyton Jones. Algorithm + strategy = parallelism. *Journal of Functional Programming*, 8(1), January 1998.