PType System: A Featherweight Parallelizability Detector

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

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

Abstract. Parallel programming is becoming an important cornerstone of general computing. In addition, type systems have significant impact on program analysis. In this paper, we demonstrate an automated type-based system that soundly detects parallelizability of sequential functional programs. Our type inference system discovers the *parallelizability* property of a sequential program in a modular fashion, by exploring a ring structure among the program's operators. It handles self-recursive functions with accumulating parameters, as well as a class of non-linear mutual-recursive functions. Programs whose types are inferred to be parallelizable can be automatically transformed to parallel code in a *mutumorphic* form – a succint model for parallel computation. Transforming into such a form is an important step towards constructing efficient data parallel programs.

1 Introduction

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

In the functional programming community, functions are usually defined recursively, and it is an open problem whether a general and formal method exists to parallelize any sequential recursive definition. One practically useful approach is the skeletal approach [20, 9], where two restrictions have been imposed on function definitions:

1. The operators used in the higher order functions should satisfy the associative property.
2. Programs should be expressed in some restrictive recursive forms captured by the higher order functions such as map, reduce, scan, *etc.*

In this paper, we propose a parallelizability detection methodology that alleviates these restrictions. Specifically, we demonstrate a system, called *Parallelizable Type System* (`PType` system in short), in which parallelizability of sequential recursive code can be detected through automatic program analysis. By parallelizability, we mean that there exists a parallel code with time complexity that is of order $O(log\ m\ /\ m)$ faster than its sequential counterpart, where m is the size of the input data.

To alleviate the first restriction, we introduce a *type inference system* that discovers the *extended-ring* property of the set of operators used in a program. We show that this property ensures parallelization of a program. Through our system, users need not know *how* associative operators are combined to enable parallelization. This separation of concern will greatly facilitate parallelization process.

To remove the second restriction, our system accepts any first-order functional programs with strict semantics. If a program passes the type checking phase, it can be automatically converted to parallel codes. Otherwise, the program will remain as it is.

For example, consider the following polynomial function definition:

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

In the skeletal approach, we have to introduce a (non-intuitive) combining operator $comb2$ (which is associative). Thus, the revised definition of $poly$ is:

$$\begin{aligned} &poly\ xs\ c\ =\ fst\ (polytup\ xs\ c) \\ &polytup\ [a]\ c\ =\ (a, c) \\ &polytup\ (a : x)\ c\ =\ (a, c)\ \text{`}comb2\text{`}\ (polytup\ x\ c) \\ &\ \text{where } comb2\ (p_1, u_1)\ (p_2, u_2)\ = (p_1 + p_2 * u_1,\ u_2 * u_1) \end{aligned}$$

As this revised definition matches the following skeleton, parallelization is thus guaranteed.

$$poly\ xs\ c\ =\ fst\ (reduce\ comb2\ (map\ (\backslash\ x\ \rightarrow\ (x, c))\ xs))$$

On the other hand, our `PType` system can detect that the sequential definition of $poly$ is parallelizable. It infers that the expression $(a\ +\ c\ \times\ (poly\ x\ c))$ has the type $R_{[+,\times]}$. This implies that $+$ and $\times$ in $R_{[+,\times]}$ exhibit an *extended-ring property*. The corresponding parallel code for $poly$ is as follows.

$$\begin{aligned} &poly\ [a]\ c\ =\ a \\ &poly\ (xl\ +\!\!+\ xr)\ c\ =\ poly\ xl\ c\ +\ (prod\ xl\ c)\ \times\ (poly\ xr\ c) \\ &prod\ [a]\ c\ =\ c \\ &prod\ (xl\ +\!\!+\ xr)\ c\ =\ (prod\ xl\ c)\ \times\ (prod\ xr\ c) \end{aligned}$$

An algorithm that automatically transforms a well-`PTyped` sequential program to an efficient homomorphism, a desired parallel computation model [21], can be found in [23].

In our implementation, the system handles first-order functional programs. It is able to parallelize a wide class of recursively-defined functions with accumulating parameters and with non-linear recursion. For clarity of the presentation, we first illustrate the system without these two features in Section 4.1 and discuss them separately in Section 4.2.

The main technical contributions of this work are as follows:

1. We propose an *extended ring property* of operators used in sequential programs, which guarantees the parallelizability of these programs. This frees programmers from the burden of finding a skeleton form.
2. We propose a novel and featherweight *type inference* system for detecting parallelizability of sequential programs in a modular fashion. We believe this is the first work on capturing parallelism in a type inference context.

The outline of the paper is as follows. In the next section, we describe the syntax of the language used, and the background of our work. Section 3 provides our account of the parallelizability property. The discovery of parallelizability using a type system is described in Section 4. We illustrate the working of the `PType` system through examples in Section 5. Section 6 describes our implementation. Finally, we discuss the related work and conclude the paper in Section 7.

2 Background

The `PType` system operates on a first-order typed functional language with strict semantics. The syntax of our source language is given in Figure 1. To aid the type inference, programmers are required to provide as annotatations properties of user-defined binary operators used in a program. Such requirements are typical for achieving reduction-style parallelism. For example, the system-defined 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.

Function definitions in this paper are written in Haskell syntax [15]. For the remainder of the paper, we shall discuss detection of parallelism for recursive functions of the form

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

where f is inductively defined on a list and $E[\,]$ denotes an *expression context* with three groups of holes, denoted by $\langle\ \rangle$. The context itself contains no occurrence of references to a, x and f. $\langle t_i \rangle_{i=1}^{m}$ is a group of m terms, each of which is allowed to contain occurrences of a, but not those of references to $(f\ x)$. The $\langle q\ x \rangle$ denotes

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

Fig. 1. Syntax of the source language

an application of a parallelizable auxiliary function.[1] Lastly, $\langle f\ x\rangle$ is the self-recursive call.

For example, given the function definition

$$f_1\ (a : x) = \textbf{if}\ a > 0\ \textbf{then}\ \mathit{length}\ x + f_1\ x\ \textbf{else}\ 1 + f_1\ x$$

we have

$$
\begin{array}{l}
f_1\ (a : x) = E[\langle a > 0,\ 1\rangle, \langle \mathit{length}\ x\rangle, \langle f\ x\rangle] \\
\ \texttt{where}\ E[\langle t_1, t_2\rangle, \langle t_3\rangle, \langle t_4\rangle] = \textbf{if}\ \ t_1\ \ \textbf{then}\ \ t_3 + t_4\ \ \textbf{else}\ \ t_2 + t_4
\end{array}
$$

As our analysis focuses on the syntactic expressions consisting of recursive calls, all variables directly or indirectly referencing an expression consisting of recursive call(s) need to be traced. We call such variables *references to a recursive call*, which is formally defined below:

Definition 1 (Reference to a Recursive Call). *A variable v is a* reference to a recursive call *if the evaluation of v leads to an invocation of that call.*

Consider the following two function definitions:

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

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

For ease of the presentation, we focus our attention on recursive function definitions that are linear self-recursive (and discuss the handling of non-linear

[1] It is possible to consider applications of multiple parallelizable auxiliary functions in an expression, as in $\langle q_j\ \ x\rangle_{j=1}^{n}$. These functions are examples of mutumorphism [14]. Their calls can be *tupled* to obtain a single $(q\ x)$ via the technique described in [4, 13].

and mutually recursive functions in Section 4.2.) Furthermore, we do not consider functions with self-recursive calls occurring in the test of a conditional. Parallelization of such functions requires these functions to be annotated with a special (constraint) form of the extended-ring property [6], which are not described in this paper.

Context Preservation. Our parallelization process is inspired from a program restructuring technique known as *context preservation* [8]. We briefly describe the technique here.

Consider the polynomial function definition again. Context preservation is performed primarily on the recursive equation of *poly*:

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

A *contextual function* (or *context*, for short) will extract away the recursive subterm of the RHS of this equation. It can be written as $\lambda\,(\underline{\bullet})\ .\ \alpha\ +\ \beta \times\ (\underline{\bullet})$. Here, the symbol $\underline{\bullet}$ denotes a recursive subterm containing an occurrence of a self-recursive call, while α and β denote subterms that do not contain any recursive call. Such a context is said to be *context preserving modulo replication* (or *context preserving*, in short) if after composing the context with itself, we can obtain (by transformation) a resulting context that has the same form as the original context. Context preservation guarantees that the underlying function can be parallelized.

Theorem 1 (Context Preservation Theorem [8, 14]). *Given is 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 f can be parallelized.*

For function *poly*, let its context be denoted by $\lambda\,(\underline{\bullet})\ .\ \alpha_1\ +\ \beta_1\ \times (\underline{\bullet}))$. We compose this context with its renamed copy, $(\lambda(\underline{\bullet})\ .\ \alpha_2\ +\ \beta_2\ \times\ (\underline{\bullet}))$, and simplify the composition through a sequence of transformation steps:

$$\begin{array}{lll}
(\lambda\,(\underline{\bullet})\ .\ \alpha_1 + \beta_1 \times (\underline{\bullet})) \circ (\lambda(\underline{\bullet})\ .\ \alpha_2 + \beta_2 \times (\underline{\bullet})) & & \\
= \ \lambda\,(\underline{\bullet})\ .\ \alpha_1 + \beta_1 \times (\alpha_2 + \beta_2 \times (\underline{\bullet})) & & \text{— function composition} \\
= \ \lambda\,(\underline{\bullet})\ .\ \alpha_1 + (\beta_1 \times \alpha_2 + \beta_1 \times (\beta_2 \times (\underline{\bullet}))) & & \text{— } \times \text{ is distributive over } + \\
= \ \lambda\,(\underline{\bullet})\ .\ (\alpha_1 + \beta_1 \times \alpha_2) + (\beta_1 \times \beta_2) \times (\underline{\bullet}) & & \text{— } +, \times \text{ being associative} \\
= \ \lambda\,(\underline{\bullet})\ .\ \alpha + \beta \times (\underline{\bullet}) & & \\
\quad \text{where } \alpha = \alpha_1 + \beta_1 \times \alpha_2 \text{ and } \beta = \beta_1 \times \beta_2 & &
\end{array}$$

Since the simplified form matches the original context, *poly* is context preserving. However, this transformation process, which is informally described in [5], is more expensive than our type-based approach. Moreover, context preservation checking is not modular, and thus lack of reusability.

3 Parallelizability

Given that context preservation leads to parallelizability, we focus on detecting context preservation of sequential programs, but in a modular fashion. Our first technical contribution is to introduce an *extended ring property* of the operators which *guarantees automatic detection of context preservation.*

$$
\begin{array}{lll}
sv & \in & \textbf{S–Values} \\
sv & ::= & bv \mid \textbf{if}\ \zeta_a\ \textbf{then}\ \zeta_b\ \textbf{else}\ bv \\
bv & ::= & \underline{\bullet} \mid (\zeta_1 \oplus_1 \ldots \oplus_{n-1} \zeta_n \oplus_n \underline{\bullet}) \\
 & & \text{where } [\oplus_1, \ldots, \oplus_n] \text{ possesses the} \\
 & & \text{extended-ring property}
\end{array}
\qquad
\begin{array}{lll}
\zeta & \in & \textbf{C–Exp} \\
\zeta & ::= & C[a,\ (q\,x)] \\
 & & \text{where } C \text{ is an arbitrary expression} \\
 & & \text{context not involving references to } \underline{\bullet}
\end{array}
$$

Fig. 2. Skeletal Values

Definition 2. *Let $S = [\oplus_1, \ldots, \oplus_n]$ be a sequence of n binary operators. We say that S possesses the* extended-ring property *iff* [2]

1. *all operators are associative;*
2. *each operator $\oplus$ has an identity, $\iota_\oplus$, such that $\forall\ v : \iota_\oplus\ \oplus\ v = v\ \oplus\ \iota_\oplus\ = v$;*
3. *$\oplus_j$ is distributive over $\oplus_i$ $\forall\, 1 \leq\ i\ <\ j\ \leq\ n$.*

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

We now describe a set of "skeletons" (of expressions) which are constructed using a sequence of binary operators with the extended-ring property. We will show that expressions expressible in this "skeletal" form are guaranteed to be context preserving. We call them *skeletal values* (or *s-values*, in short). These are defined in Figure 2. We use $\underline{\bullet}$ to denote a self-recursive call in a function definition.

An s-value of the form $(\zeta_1\ \oplus_1\ \ldots\ \oplus_{n-1}\ \zeta_n \oplus_n\ \underline{\bullet})$[3] is said to be *composed directly by* the sequence of operators $[\oplus_1,\ \ldots,\ \oplus_n]$ with the extended-ring property. An s-value of the form **if** ζ_0 **then** ζ_1 **else** bv is said to be in *conditional form.* Its self-recursive call occurs *only* in its alternate branch.

The following lemma states that all s-values are context preserving. Consequently, any expression that can be normalized to an s-value can be parallelized.

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

The proof is done by a case analysis on the syntax of s-values. Details can be found in [23].

It is worth-mentioning that s-values cover a wide class of recursive function definitions that are parallelizable. In the remainder of the paper, we will provide many practical sequential programs that can be expressed in, or normalized to an s-value, and thus be directly parallelized.

[2] We can also extend this property to include semi-associative operators and their corresponding left or right identities. Such extension enables more sequential programs to be parallelized.

[3] By default, it is equivalent to $(\zeta_1 \oplus_1 (\cdots \oplus_{n-1} (\zeta_n \oplus_n \underline{\bullet}) \ldots))$.

4 PType System

The main focus of the `PType` system is a type-inference system that enables discovery of parallelizability of sequential programs. Operationally, the type system aims to deduce the extended-ring property of a sequential program in a modular fashion. To this end, it associates each sub-expression in a recursive function definition with a type term from the type language `PType`.

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

Fig. 3. `PType` Expressions

The set of `PType` terms are defined in Figure 3. It comprises two categories: `NType` and `RType`. We write $[\![\rho]\!]$ to denote the semantics of `PType` ρ. Thus,

$$[\![N]\!] \; = \; \mathbf{C\text{–}Exp},$$

where **C–Exp** is defined in Figure 2.

Given that $S = [op_1, \ldots, op_n]$ with the extended-ring property, we have:

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

where $\rightsquigarrow^*$ represents a normalization process that we have defined to obtain s-values. The core set of rules for the normalization process is in [23].

Since expressions of type R_S (for some S) can be normalized to an s-value, any expression containing a self-recursive call but could *not* be translated to an s-value is considered ill-typed in our `PType` system.

As an illustration, the RHS of the self-recursive equation of the following function definition has ptype $R_{[max,+,\times]}$.

$$f_6 \; (a : x) = 5 \; \text{‘}max\text{‘} \; (a + 2 \times (f_6 \; x)),$$

Note that in the definition of $[\![\, R_S \,]\!]$, the expression e' is said to be composable, rather than to be 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 conditionals and local abstractions, but its set of operators must be limited to S.
2. As operators in S have identities, we allow e' to contain just a subset of operators in S. We can always extend e' to contain all operators in S using their respective identities.

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

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

The above lemma leads to the following subtyping relation:

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

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

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

This states that the expression e has `PType` ρ assuming that any free variable in it has `PType` given by Γ and κ is an expression that may occur in e. κ is either a self-recursive call or a *reference* to such a 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 initialize κ to be the term $(f\ x)$. In Γ, we also assign `PType` N to the recursive parameters of f.

Finally, 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 both a and x to N. Otherwise, it is said to be *ill-PTyped.*

4.1 PType Checking

The `PType` of a function f is defined as the `PType` of the RHS of its recursive equation. Figure 4 lists the core set of type-checking rules.

Both constants and variables not referencing any recursive call are given `NType`, as shown in the rules (`var-N`) and (`con`). Use of a variable has type `RType` if it is the currently active reference, namely κ. The self-recursive call $(f\ \ x)$ will also be given an `RType`. We note that any use of inactive references are ill-`PType`, as there is no corresponding rule for it.

Rule `(op)` handles a binary operation in which a recursive function call occurs in its right operand. The operation yields a `RType` if the right operand has `RType`, and the operator under investigation is part of the sequence S. The case in which the recursive call occurs in its left operand is symmetrical, and thus omitted.

In the 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 coerced to the same type R_S. These constraints are expressed by the relation $\bigtriangledown_{\texttt{if}}$, defined by `(if-merge)`, while the coercion is defined via a type subsumption `(sub)`. For example, consider the following function definition:

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

Under the type assumption $\Gamma = \{a :: N, x :: N\}$, the types for each of the branches are $R_{[+]}$ and $R_{[\times]}$. By the rules `(if-merge)` and `(sub)`, the type of the conditional becomes $R_{[+,\times]}$.

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

$$\frac{}{\Gamma \vdash_\kappa n :: N}\ (\texttt{con}) \qquad \frac{}{\Gamma \vdash_{(f\ x)} (f\ x) :: R_S}\ (\texttt{rec})$$

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

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

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

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

$$\frac{\Gamma \vdash_\kappa e :: N \quad g \notin FV(\kappa)}{\Gamma \vdash_\kappa (g\,e) :: N}\ (\texttt{g}) \qquad \frac{\Gamma \vdash_\kappa e : \rho \quad \rho <: \rho'}{\Gamma \vdash_\kappa e :: \rho'}\ (\texttt{sub})$$

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

Fig. 4. Type-Checking Rules

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

Note that in the rule `(let-R)`, the deductive operator has changed from $\vdash_\kappa$ to $\vdash_v$. This means that in e_2, v is the sole active reference to the recursive function. Thus, the following two expressions will fail the `PType` check: In the first expression, the recursive call is non-linear; in the second expression, the use of v is non-linear.[4]

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

In rule `(g)`, the application of an auxiliary function g is of `NType` if its argument e is of `NType` too. Otherwise, such an application may not be effectively parallelized [10], and the application will be deemed ill-`PTyped`.

The soundness of our type-checking rules is shown by relating the rules to a set of normalization rules defining $\leadsto$, as shown in [23]. The main results are listed below; we refer the reader to [23] for detail.

Theorem 2 (Progress). *If $\Gamma \vdash_\kappa e :: R_S$, then either e is an s-value or $e \leadsto e'$.*

[4] Conversion of these simple non-linear expressions to their linear counterparts can be trivially done via pre-processing. We omit the detail here.

Theorem 3 (Preservation). *If* $e :: R_S$ *and* $e \leadsto e'$*, then* $e' :: R_S$*.*

Furthermore, in [23], we define a `PType` inference algorithm which is both sound and complete with respect to our `PType` checking rules. Our algorithm adopts the idea of the *type reconstruction algorithm* $\mathcal{W}_{\mathrm{UL}}$ as described in [17].

4.2 Enhancement of the PType System

In this section, we show that the `PType` system can be enhanced to cover broader classes of parallelizable codes. These enhancements have been included in our implementation [23].

Multiple Recursion Parameters. When a function f has multiple recursion parameters, we require f to recurse over all its recursion parameters *at the same frequency*. That is, f is of the following form:

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

where $Ctx[\,]$ is an arbitrary context, and the expression $\ldots(f\ x_1 \ldots x_n)\ldots$ states that any self-recursive call in the equation should be of the form $(f\ x_1 \ldots x_n)$.

Example: Polynomial Addition. The following definition of *polyadd* satisfies this requirement, and its `PType` is $R_{[+\!+]}$.

$$\begin{array}{l} \#(List\ Float, [+\!+], [Nil]) \\ polyadd\ [\,]\ [\,]\ =\ [\,] \\ polyadd\ [\,]\ ys\ =\ ys \\ polyadd\ xs\ [\,]\ =\ xs \\ polyadd\ (a : x)\ (b : y)\ =\ [(a + b)]\ +\!+\ polyadd\ x\ y \end{array}$$

For clarity, we use $\overrightarrow{x}$ to denote $x_1 \ldots x_n$. To handle multiple recursion parameters, we replace all occurrences of $(f\ x)$ with $(f\ \overrightarrow{x})$ in the type checking rules, and include $\{a_1 :: N, \ldots, a_n :: N, x_1 :: N, \ldots, x_n :: N\}$ to Γ before type checking the RHS of the equation.

Accumulating Parameters. When a function f has accumulating parameters, we shall verify the well-`PType`dness of each of these parameters *individually* before type-checking f's body. If any one of the accumulating parameters is found to be ill-typed, we conclude that the function f is ill-typed too. Thus, given a function definition of the form

$$f\ (a_1 : x_1)\ \ldots\ (a_n : x_n)\ p_1\ \ldots\ p_n\ =\ \ldots(f\ \overrightarrow{x}\ e_1 \ldots e_i \ldots e_n)\ldots,$$

where $p_1\ \ldots\ p_n$ are accumulating parameters, the type checking proceeds as follows:

$$\frac{\forall\, i\, \in\, \{1,\ldots,n\} : \Gamma\, \cup\, \{a_i :: N,\, x_i :: N,\, p_i :: N\}_{i\,\in\,\{1,\ldots,n\}}\ \vdash_{p_i} \mathcal{C}[\![\, e\,]\!]_{p_i}\ ::\ \rho_i}{\Gamma\, \cup\, \{a_i :: N,\, x_i :: N,\, p_i :: N\}_{i\,\in\,\{1,\ldots,n\}}\ \vdash_{(f\ \overrightarrow{x})}\ e :: R_S}$$

The context derivation function $\mathcal{C}$ takes an expression e and an accumulating parameter p_i as inputs and returns an expression which is the context of the accumulating parameter p_i. Its definition is available in Figure 5.

$$
\begin{array}{l}
\mathcal{C} :: \texttt{Exp} \rightarrow \texttt{Var} \rightarrow (\texttt{Exp},\ \texttt{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\quad \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 \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}
$$

Fig. 5. Definition of Context-Derivation Function $\mathcal{C}$

Example: Bracket Matching Problem. This is a language recognition problem which determines whether the brackets $'('$ and $')'$ occurring in a given string can be matched correctly. 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 is increased/decreased whenever an opening/closing bracket is encountered. The following definition is taken from [14].

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

Two annotations are needed to type-check this program. The annotation for operators of *Bool* is meant for type checking the function sbp', and that for operators of *Int* is for type checking the context of the accumulating parameter c. The context is computed as follows:

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

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

Non-linear Mutual Recursion. We extend the `PType` system to cover a subset of non-linear recursive functions with an additional requirement that the binary operators must be commutative. This additional requirement is typical for research in the parallelization of non-linear recursive functions.

To parallelize a set of non-linear mutual recursive functions, we group these functions into a tuple and type-check them together. Thus, we extend κ in $\vdash_\kappa$ to become a set of mutual-recursive calls.

Consider the following mutually defined recursive functions:

$$\begin{array}{l} f_i\,(a:x) \;=\; e_i \quad \forall\, i \,\in\, \{1,\ldots,m\} \\ \text{where}\,\forall\, i \,\in\, \{1,\ldots,m\} : e_i \;=\; p_{i1} \,\oplus\, (p_{i2} \otimes f_1\,x) \,\oplus\, \ldots \,\oplus\, (p_{im} \,\otimes\, f_m\,x) \\ \qquad\qquad\qquad\qquad\quad\; \forall\, j \,\in\, \{1,\ldots,m\} :\; p_{ij} \;=\; g_{ij}\; a\,(q_j\; x) \end{array}$$

Here, functions g_{ij} are arbitrary functions (*i.e.*, arbitrary contexts) involving a and $(q_j\ x)$, $\forall i,j \in \{1,\ldots,m\}$. Before type checking, we group the function definitions into a tuple: $(f_1,\ldots,f_m) = (e_1,\ldots,e_m)$. For all $j \in \{1,\ldots,m\}$, type check e_j with rules defined in Figure 4, together with the `(op-RR)` rule and type check the tuple $(e_1,\ldots,\ e_m)$ using the `(nonlinear)` rule.

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

$$\frac{\Gamma\; \vdash_{\{(f_1\ x),\ldots,(f_m\ x)\}}\; e_j :: R_S \quad \forall\, j\,\in\,\{1,\ldots,m\}}{\Gamma\; \vdash_{\{(f_1\ x),\ldots,(f_m\ x)\}}\; (e_1,\ldots,\ e_m) :: R_S}\;(\mathtt{nonlinear})$$

Example: Fibonacci. For the following non-linear recursive definition of the Fibonacci function,

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

we sketch below the type checking process:

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

Hence, both *lfib* and *lfib'* have type $R_{[+]}$.

For functions which are defined with both non-linear recursion and accumulating parameters, we first transform them into their linear recursive counterpart such that the accumulating parameters and the recursion arguments can be processed in a synchronized manner [7]. If this succeeds, the transformed functions will be amenable to parallelization.

5 Examples

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

The mss Problem. Consider a sequential program that finds the *maximum segment sum* (mss) of a list.

$\#(Int,\ [max, +], [0, 0])$

$mis\,[a] \ = \ a$ $\qquad mss\,[a] \ = \ a$

$mis\,(a : x) \ = \ a\,`max`\,(a \ + \ mis\,x)$ $\quad mss\,(a : x) \ = \ (a\,`max`\,(a \ + \ mis\,x))\,`max`\,mss\,x$

In the definition of function *mss*, function *mis* is called with the recursion argument x. This implies that an effective parallelization of *mss* requires *mis* to be parallelizable as well. Thus, we type check the definition of *mis* before that of *mss*. The PType inferred for both definitions are: $mis :: R_{[max,+]}$ and $mss :: R_{[max]}$ respectively.

Lists and Skeletons. We show that components of the traditional skeletons such as *scan*, *map*, and *reduce*, can be viewed as parallelizable components in our PType system. Consequently, programs constructed via these skeletons can be parallelized by our system.

An extended-ring property for lists is: $\#(List, [+\!\!+, map2], [Nil, Nil])$, where $map2$ is in turn defined as: $y\ `map2`\ z \ = \ map\ (y +\!\!+)\ z$. Function $map2$ has the following properties:

1. distributive over $+\!\!+$: $\quad y\,`map2`\,(zl +\!\!+ zr) \ = \ y\,`map2`\,zl \ +\!\!+ y\,`map2`\,zr$
2. semi-associative : $\quad x\,`map2`\,(y\,`map2`\,z) \ = \ (x \ +\!\!+ y)\,`map2`\,z$

From the following recursive definition of the function *scan*, we can infer that *scan* has type $R_{[+\!\!+,map2]}$:

$\#(List, [+\!\!+, map2], [Nil, Nil])$

$scan\,[a] \ = \ [\,[a]\,]$

$scan\,(a : x) \ = \ [\,[a]\,] \ +\!\!+ ([a]\,`map2`\,(scan\,x))$

Similarly, we can apply this methodology to obtain the ptypes of *map* and *reduce*.

Technical Indicators in Financial Analysis. Many technical indicators used in technical analysis of financial market can be parallelized with our system. Following is a program for computing *exponential moving average* [1]:

$\#(Indicator\ Price,\ [+, \times], [0, 1])$

$ema\,(a : x) \ = \ (close\ a) \ : \ ema'\,(a : x)\,(close\ a)$

$ema'\,[\,]\,p \ = \ [\,]$

$ema'\,(a : x)\,p \ = \ \mathbf{let}\,r \ = \ (0.2 \ \times \ (close\ a) \ + \ 0.8 \ \times \ p)\ \mathbf{in}\ [r] \ +\!\!+ ema'\,x\,r$

The *ema* for the first day in a price history is just its closing price. At any other day, the *ema* is computed by summing the weighted closing price of that day and the weighted moving average of the previous day. The PType of the accumulating parameter p is $R_{[+,\times]}$ and that of the function ema' is $R_{[+\!\!+]}$. Finally, the PType of the function *ema* is N.

Fractal Image Decompression. A fractal image may be encoded by a series of affine transformations (which are combinations of scalings, rotations and translations) to the coordinate axes. The decompression problem has beenconsidered in [12]. Here, we look into the parallelization of two important functions used in the decomposition process.

$\#(List, [+\!\!+], [Nil])$
$\#(Set, [union], [Nil])$

$tr :: [a \rightarrow a] \rightarrow a \rightarrow [a]$	$k :: [[a]] \rightarrow [a]$
$tr\ [f]\ p = [f\ p]$	$k\ [a]\ fs = nodup\ (tr\ fs\ a)$
$tr\ (f : fs)\ p = [f\ p] +\!\!+ tr\ fs\ p$	$k\ (a : x)\ fs = nodup\ (tr\ fs\ a)$ ‘$union$‘ $(k\ x)$

Here, function *tr* applies a list of transformations to a pixel, and function *k* applies these transformations to a set of pixels with the help of *tr*. Function *nodup* generates a set by removing repeated occurrences of a value from a list. Types of *tr* and *k* can be inferred to be $R_{[+\!\!+]}$ and $R_{[union]}$ respectively.

6 Implementation

We have implemented a prototype of the `PType` system in Haskell 98 [15]. We have also provided a web interface to the `PType` system. The URL is

`http://loris-4.ddns.comp.nus.edu.sg/~xun`.

We have tested our system with a set of non-trivial sequential programs including applications such as matrix multiplication, inversion, and polynomial multiplication, *etc.* Details of these programs can be found in the above URL as well. The experiment was performed on a 2 GHz Pentium-4 CPU with 512 MB of RAM. The total times taken to do `PType` inference for some of the applications are shown in Table 1. In general, the time complexity of `PType` inference is $O(n)$ where n is the size of the sequential program. The parallel code generation has time complexity of $O(n^2)$. The time complexity for executing the resulting parallel code is typically $O(log\ m)$ where m is the size of the input data.

Table 1. Parallelization Times for Some Sequential Programs

	matrix multiplication	matrix inverse	polynomial multiplication	mss	fractal image decompression
Lines of Code	50	65	16	10	22
Time (Sec)	0.026	0.04	0.007	0.007	0.04

7 Related Works and Conclusion

Generic program schemes, such as algorithmic skeletons, have been advocated for use in structured parallel programming, both for imperative programs expressed as first-order recurrences through a classic result of [22] and for functional programs via Bird's homomorphism [20, 9]. However, most sequential specifications fail to match up *directly* with these schemes. To overcome this shortcoming, there have been calls to constructively transform programs to match these schemes. But these proposals [19, 12] often require deep intuition and the support of ad-hoc lemmas – making automation difficult. Another approach is to provide more specialized schemes, either statically [18] or via a procedure [14], that can be directly matched to a sequential specification.

On the imperative language (e.g. Fortran) front, there have been interests in parallelization of reduction-style loops [10, 11]. By modeling loops via functions, 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 [3]. Imperative loop corresponds to tail recursion, and can be considered as a special case of the linear recursive form that we have described here.

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 [16, 2]. However, these two type systems are still based on the evaluation rules of the underlying language.

We have introduced a novel view to parallelization. To the best of our knowledge, this is the first piece of work that brings together type systems and parallelization. By bringing the two fields together, we hope to apply the formalism of type theory to yet another important application domain. The marriage of type systems and parallelization hinges on the idea of the extended-ring property. Through the `PType` system, we have relaxed the restrictions which have usually been imposed on parallelization (*eg.*, restriction on specific recursive form). Furthermore, the system is able to handle recursively defined functions with accumulating parameters.

With the help of the `PType` inferenced, we develop an algorithm that can automatically generates parallel code from a well-`PType`d sequential program. Due to space limitation, the derivation detail and its correctness proof are omitted.

All the above benefits are obtained without the need for users to know the detail mechanisms behind the parallelization process. Through a clean and simple interface, the system frees the user from the burden of performing normalization (which is required in [8]) and parallelizability checking (which is required in [14]). Users only need to provide the extended-ring property of the binary operators used in the programs. Indeed, we have provided a web interface to the system, through which users can parallelize their programs, or test run many of the non-trivial programs which are available at our website.

References

1. S. Anand, W.N. Chin, and S.C. Khoo. Charting patterns on price history. In *ACM SIGPLAN International Conference on Functional Programming*, pages 134–145. ACM Press, June 2001.
2. S. Berardi. Pruning simply-typed lambda-terms. *Journal of Logic and Computation*, 6(5):663–681, 1996.
3. W. N. Chin. Synthesizing parallel lemma. In *Proc of a JSPS Seminar on Parallel Programming Systems, World Scientific Publishing*, pages 201–217, Tokyo, Japan, May 1992.

4. W. N. Chin. Towards an automated tupling strategy. In *Proc. Conference on Partial Evaluation and Program Manipulation*, pages 119–132, Copenhagen, Denmark, June 1993.
5. W.N. Chin, J. Darlington, and Y. Guo. Parallelizing conditional recurrences. In *2nd Annual EuroPar Conference*, Lyon, France, (LNCS 1123) Berlin Heidelberg New York: Springer, August 1996.
6. 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.
7. W.N. Chin, S.C. Khoo, and T.W. Lee. Synchronisation analyses to stop tupling. In *European Symposium on Programming (LNCS 1381)*, pages 75–89, March 1998.
8. W.N. Chin, A. Takano, and Z. Hu. Parallelization via context preservation. In *IEEE Intl Conference on Computer Languages*, Chicago, U.S.A., May 1998. IEEE CS Press. `http://www.comp.nus.edu.sg/~chinwn/iccl98.ps`.
9. M. Cole. Parallel programming with list homomorphisms. *Parallel Processing Letters*, 5(2), 1995.
10. 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.
11. 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.
12. Z.N. Grant-Duff and P. Harrison. Parallelism via homomorphism. *Parallel Processing Letters*, 6(2):279–295, 1996.
13. 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.
14. 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.
15. S. P. Jones, J. Hughes, and et al. Report on the programming language Haskell 98, a non-strict, purely functional language.
16. N. Kobayashi. Type-based useless variable elimination. In *ACM Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 84–93, Boston, Massachusett, January 2000.
17. F. Nielson, H.R. Nielson, and C. Hankin. *Principles of Program Analysis.* Springer Verlag, 1999.
18. S.S. Pinter and R.Y. Pinter. Program optimization and parallelization using idioms. In *ACM Principles of Programming Languages*, pages 79–92, Orlando, Florida, ACM Press, 1991.
19. P. Roe. *Parallel Programming using Functional Languages (Report CSC 91/R3).* PhD thesis, University of Glasgow, 1991.
20. D. Skillicorn. Architecture-independent parallel computation. *IEEE Computer*, 23(12):38–50, December 1990.
21. D. Skillicorn. Foundations of parallel programming. In *Cambridge International Series on Parallel Computation:6*, 1994.
22. H.S. Stone. Parallel tridiagonal equation solvers. *ACM Transactions on Mathematical Software*, 1(4):287–307, 1975.
23. N. Xu. A type-based approach to parallelization. *MSc thesis, School of Computing, National University of Singapore http://www-appn.comp.nus.edu.sg/~esubmit/search/index.html*, July 2003.