Deriving Parallel Codes via Invariants

Wei-Ngan Chin[1], Siau-Cheng Khoo[1], Zhenjiang Hu[2], and Masato Takeichi[2]

[1] School of Computing, National University of Singapore, Singapore
{chinwn, khoosc}@comp.nus.edu.sg
[2] Department of Information Engineering, University of Tokyo, Japan
hu@ipl.t.u-tokyo.ac.jp, takeichi@u-tokyo.ac.jp

Abstract. Systematic parallelization of sequential programs remains a major challenge in parallel computing. Traditional approaches using program schemes tend to be narrower in scope, as the properties which enable parallelism are difficult to capture via ad-hoc schemes. In [CTH98], a systematic approach to parallelization based on the notion of preserving the *context* of recursive sub-terms has been proposed. This approach can be used to derive a class of divide-and-conquer algorithms. In this paper, we enhance the methodology by using *invariants* to guide the parallelization process. The enhancement enables the parallelization of a class of recursive functions with conditional and tupled constructs, which were not possible previously. We further show how such invariants can be discovered and verified systematically, and demonstrate the power of our methodology by deriving a parallel code for maximum segment product. To the best of our knowledge, this is the first systematic parallelization for the maximum segment product problem.
Keywords: *Parallelization, Context Preservation, Invariants, Conditional Recurrences, Constraints.*

1 Introduction

It is well-recognised that a key problem of parallel computing remains the development of efficient and correct parallel software. Many advanced language features and constructs have been proposed to alleviate the complexities of parallel programming, but perhaps the simplest approach is to stick with sequential programs and leave it to parallelization techniques to do a more decent transformation job. This approach could also simplify the program design and debugging processes, and allows better portability to be achieved.

A traditional approach to this problem is to identify a set of useful higher-order functions for which parallelization can be guaranteed. As an example, Blelloch's NESL language [BCH$^+$93] supports two extremely important parallel primitives, scan and segmented scan, that together covers a wide range of parallel programs. Specifically, segmented scan can be used to express non-trivial problems (such as sparse matrix calculations and graph operations [Ble90]) that are difficult to parallelize due to their use of irregular data structures.

However, before a programmer can use these higher-order parallel primitives, she must *manually match* her problem to the sequential form of scan and segmented scan (based on flattened list), shown below.[1]

$$
\begin{array}{ll}
scan\ (\oplus)\ ([], w) & = [] \\
scan\ (\oplus)\ (x : xs, w) & = [(w \oplus x)] +\!\!+ scan\ (\oplus)\ (xs, w \oplus x)
\end{array}
$$

$$
\begin{array}{ll}
segscan\ (\oplus)\ ([], w) & = [] \\
segscan\ (\oplus)\ ((x, b) : xs, w) & = \textbf{if } b == 1 \textbf{ then } [(x, b)] +\!\!+ segscan\ (\oplus)\ (xs, x) \\
& \quad \textbf{else } [(w \oplus x, b)] +\!\!+ segscan\ (\oplus)\ (xs, w \oplus x)
\end{array}
$$

(ASIDE : Our functions are defined using Haskell/ML style pattern-matching equations. Notationwise, [] denotes an empty sequence, $x : xs$ denotes an infix *Cons* data node with x as its head and xs as its tail. Also, $+\!\!+$ is the sequence concatenation operator.)

Note that operator $\oplus$ needs to be semi-associative[2]. This is required to help distribute/parallelize the above *scan* and *segscan* functions.

The process of *matching* a given program to a set of higher-order primitives is non-trivial, particularly for recursive functions with conditional and tupled constructs. As an example, consider a sequential program to find the minimum sum from all possible segments of elements.

$$
\begin{array}{ll}
mss([x]) & = x \\
mss(x : xs) & = min2(mis(x : xs), mss(xs))
\end{array}
$$

$$
\begin{array}{ll}
mis([x]) & = x \\
mis(x : xs) & = \textbf{if } mis(xs) + x \ \leq \ x \textbf{ then } mis(xs) + x \textbf{ else } x
\end{array}
$$

How may one match this function to an appropriate parallel primitive (*eg.* *scan*)? Alternatively, could we *directly derive* a parallel equivalent of this function?

While the sequential version of *mss* is straightforward to formulate, the same cannot be said for its parallel counterpart. Part of the complication stems from the presence of the conditional construct and the need to *invent* new auxiliary functions to circumvent its sequentiality.

In this paper, we propose a semi-automatic methodology for systematic derivation of parallel programs directly from their sequential equations. We focus on transforming recursive functions containing one or more conditional constructs, and make use of an *invariant*, where appropriate, to guide the transformation steps. In the case of *mis*, we first convert its sequential definition body

[1] The functions defined in this paper are written in Haskell syntax. Particularly, the parenthesis notation $(\cdot)$ converts an infix operator (such as $\oplus$) into its equivalent prefix function.

[2] An operator $\oplus$ is said to be semi-associative if there exists a companion operator $\tilde{\oplus}$ that is fully associative, such that $((a \oplus b) \oplus c) = (a \oplus (b \tilde{\oplus} c))$ holds. For example, $-$ and $\div$ are semi-associative operators with $+$ and $*$ as their respective companions. Note that *scan* and *segscan* are usually defined using the more restricted full-associative property whereby $\oplus = \tilde{\oplus}$.

(the second equation) into a contextual form: $\textbf{if}\ \underline{\bullet} \le \alpha_1\ \textbf{then}\ \underline{\bullet} + \alpha_2\ \textbf{else}\ \alpha_3$, where $\underline{\bullet} = mis(xs)$, $\alpha_1 = 0$, $\alpha_2 = x$, and $\alpha_3 = x$. By using the invariant $\alpha_3 \ge \alpha_1 + \alpha_2$, we can derive a parallel version of *mis* (and *mss*) as follows:[3]

$$
\begin{array}{lcl}
mss([x]) & = & x \\
mss(xr{+\!\!+}xs) & = & min2(mss(xr), min2(uT(xr) + mis(xs), mss(xs))) \\
\\
mis([x]) & = & x \\
mis(xr{+\!\!+}xs) & = & \textbf{if}\ mis(xs) \le uH(xr)\ \textbf{then}\ mis(xs) + uG(xr)\ \textbf{else}\ mis(xr) \\
\\
uH([x]) & = & 0 \\
uH(xr{+\!\!+}xs) & = & min2(uH(xs), uH(xr) - uG(xs)) \\
\\
uT([x]) & = & x \\
uT(xr{+\!\!+}xs) & = & min2(uT(xr) + uG(xs), uT(xs)) \\
\\
uG([x]) & = & x \\
uG(xr{+\!\!+}xs) & = & uG(xs) + uG(xr)
\end{array}
$$

(ASIDE : The concatenation operator $+\!\!+$ also serves as a sequence splitting operation when used as a pattern in the LHS of equation. Operationally, a sequence may be implemented using a vector for easier data access and distribution. However, this paper shall focus on high-level parallelization issues, and omit detailed implementation concerns, such as data-type conversion & task distribution.)

The main contributions of this paper are:

- We *extend* the parallelization methodology proposed in [CTH98] to cover recursive functions with conditional and tupled constructs. These functions are hard to parallelize, but we circumvent this difficulty by using an appropriate normalization process. (Section 3)
- A novelty of our approach is the introduction of *invariants* to the parallelization process. To the best of our knowledge, this is the first use of invariants for the derivation of parallel programs. Such an effort significantly widens the class of parallelizable functions. (Section 4)
- Instead of relying on user-defined invariants to improve parallelization opportunity, we demonstrate *how invariants can be systematically generated and verified* through high-level constraint-solving techniques. (Section 4.3 with details in Appendix A)
- We demonstrate how the sequential code for maximum segment product problem can be systematically parallelized using our methodology. To the best of our knowledge, this is the first account of systematic parallelization for this particular problem.

[3] Here, we have used an inequality invariant during our parallelization. Alternatively, we can employ a stronger *equality* invariant $\alpha_3 = \alpha_1 + \alpha_2$. This will produce a very compact parallel code for *mis*.

The outline of the paper is as follows: Section 2 describes the syntax of the language processed by our method. Section 3 details our parallelization methodology. Section 4 discusses how invariants are discovered and verified through constraint solving. We then apply our methodology to the parallelization of maximum segment product problem in Section 5. Finally, we relate our work to existing works done in the research community, before concluding the paper.

2 Language

We are mainly interested in deriving a class of divide-and-conquer algorithms with simple split operators (e.g. $+\!\!+$). For simplicity, we shall focus on a strict first-order functional language. (Limited support for passing function names as parameters will also be provided through program schemes. For example, *ssc* in Figure 1 is essentially a function scheme with $\oplus$ as its function-type variable.)

Definition 1: *A First-Order Language*

$$\begin{array}{lcl} F & ::= & \{f(p_{i,1}, \ldots, p_{i,n_i}) = t_i\}_{i=1}^{m} \\ t & ::= & v \ \mid \ c(t_1, \ldots, t_n) \ \mid \ \mathbf{if}\ t_1\ \mathbf{then}\ t_2\ \mathbf{else}\ t_3 \\ & & \mid \ f(t_1, \ldots, t_n) \ \mid \ \mathbf{let}\ (v_1, \ldots, v_n) = t_1\ \mathbf{in}\ t_2 \\ p & ::= & v \ \mid \ c(p_1, \ldots, p_n) \\ v & \in & \text{Variables}\ ; \quad c \ \in \ \text{Data constructors}\ ; \quad f \ \in \ \text{Function names} \end{array}$$ □

This language contains the usual data constructor terms, function calls, **let** and **if** constructs. Also, each function f is defined by a set of pattern-matching equations.

We shall focus on sequential functions that are inductively defined over linear data types with an associative decomposition operator. For practical reasons, we restrict the linear data type to *List* with $+\!\!+$ as its decomposition operator. We shall only perform parallelization for a special sub-class of sequential functions, called *linear self-recursive* functions.

Definition 2: *Linear Self-Recursive Functions*
A function f is said to be *self-recursive* if it contains only self-recursive calls. In addition, it is said to be *linear self-recursive* (LSR) if its definition contains exactly one self-recursive call. □

Though our methodology can support LSR-functions with multiple recursion parameters and nested patterns, this paper shall restrict the sub-class of functions considered to the following form.

$$\begin{array}{lcl} \text{---}\ f([], \lfloor v_j \rfloor_{j=1}^{n}) & = & Ez \\ \text{---}\ f(x : xs, \lfloor v_j \rfloor_{j=1}^{n}) & = & Er\langle f(xs, \lfloor Dr_j\langle v_j \rangle \rfloor_{j=1}^{n}\,)\rangle \end{array}$$

$Er\langle\ \rangle$ denotes the *context* for the self-recursive call of f in the rhs of the definition, while $\lfloor Dr_j\langle\ \rangle \rfloor_{j=1}^{n}$ denote the *contexts* for the *accumulative* parameters $v_{j\,\in\,1..n}$ appearing in the call to f. These expression contexts are referred to as *Recurring Contexts* (or *R-contexts* in short), as they capture the recurring subterms for

$$
\begin{array}{ll}
msp([x]) & = x \\
msp(x : xs) & = max2(mip(x : xs), \\
 & \qquad msp(xs)) \\
mip([x]) & = x \\
mip(x : xs) & = \\
\quad \textbf{if } x > 0 \textbf{ then } max2(x, x{*}mip(xs)) & \\
\quad \textbf{else } max2(x, x{*}mipm(xs)) & \\
mipm([x]) & = x \\
mipm(x : xs) & = \\
\quad \textbf{if } x > 0 \textbf{ then } min2(x, x{*}mipm(xs)) & \\
\quad \textbf{else } min2(x, x{*}mip(xs)) &
\end{array}
$$

$$
\begin{array}{ll}
sbp([x]) & = conv(x) \\
sbp(x : xs) & = \textbf{if } sbp(xs) \leq 0 \textbf{ then} \\
 & \qquad sbp(xs) + conv(x) \\
 & \quad \textbf{else } 1 \\
conv(x) & = \textbf{if } x ==')' \textbf{ then } -1 \\
 & \quad \textbf{else } (\textbf{ if } x ==' (' \textbf{ then } 1 \text{)} \\
 & \qquad \textbf{else } 0 \\
ssc\ (\oplus)\ ([], c) & = c \\
ssc\ (\oplus)\ (x : xs, c) & = \textbf{if } p(x) \textbf{ then } c \\
 & \quad \textbf{else } ssc\ (\oplus)\ (xs, c) \oplus x
\end{array}
$$

Fig. 1. Sample Sequential Programs

each self-recursive function. Note that there exists exactly one self-recursive f call. $Er\langle\ \rangle$ and $\lfloor Dr_j\langle\ \rangle\rfloor_{j=1}^{n}$ do not contain other self or mutual-recursive calls. Also, x, xs are allowed to occur freely in E_r and $\lfloor Dr_j\rfloor_{j=1}^{n}$.

Several example functions are available in Fig. 1. Here, function *msp* computes the maximum segment product; function *sbp* checks a string for properly paired bracket; and *ssc* is a higher-order program scheme. Notice that both *ssc* and *msp* are LSR functions. However, *mip*, *mipm*, and *sbp* are not. Fortunately, these functions can be converted easily using the following two pre-processing techniques.

Firstly, *tupling* [Chi93,HITT97] may be used to eliminate multiple recursive calls. For simple cases such as duplication of calls in *mis* and *sbp*, tupling eliminates duplication via **let** abstraction. For more complicated cases, such as multiple mutual-recursive calls of *mip* and *mipm*, it generates an LSR-function *miptup* such that for any list l, $miptup(l) = (mip(l),\ mipm(l))$. The generated *miptup* function is:

$$
\begin{array}{lll}
miptup([x]) & = & (x, x) \\
miptup(x : xs) & = & \textbf{let } (u, v) = \ miptup(xs) \\
 & & \textbf{in if } x \ > \ 0 \textbf{ then } (max2(x, x{*}u), min2(x, x{*}v)) \\
 & & \quad \textbf{else } (max2(x, x{*}v), min2(x, x{*}u))
\end{array}
$$

Secondly, a *conditional normalization* procedure [CDG96] can be applied to combine recursive calls in different conditional branches together. Our conditionals are first expressed in a more general guarded form:

$$
(\textbf{ if } b \textbf{ then } e_1 \textbf{ else } e_2 \ ) \ \Leftrightarrow \ \textbf{if} \begin{cases} b & \rightarrow \ e_1 \\ \neg\, b & \rightarrow \ e_2 \end{cases}
$$

After that, the following duplicate elimination rule is applied to combine multiple self-recursive calls in different branches together.

$$\textbf{if}\ \{\ b_i\ \rightarrow\ E\langle e_i\rangle\ \}_{i\,\in\,N}\ \ \Rightarrow\ E\langle\textbf{if}\ \{\ b_i\ \rightarrow\ e_i\}_{i\,\in\,N}\rangle$$

For example, with *segscan*, we obtain:

$$\begin{array}{lcl} segscan\ (\oplus)\ ([], w) & = & [] \\ segscan\ (\oplus)\ ((x,b):xs, w) & = & (\ \textbf{if}\ b == 1\ \textbf{then}\ [(x,b)]\ \textbf{else}\ [(w{\oplus}x, b)]\ \ ) +\!\!+ \\ & & \quad segscan\ (\oplus)\ (xs,\ \textbf{if}\ b == 1\ \textbf{then}\ x\ \textbf{else}\ w{\oplus}x\ \ ) \end{array}$$

3 Parallelization Methodology

Our methodology for parallelization is based on the notion of generalizing from sequential examples. Given a LSR-function, f, we attempt to obtain a more *general* parallel equation directly from its sequential equation.

An earlier version of this methodology was presented in [CTH98]. However, this earlier work did not handle recursive functions, with conditional and tupled constructs, well. A major innovation of this paper is the use of *invariants* to facilitate parallelization. We shall first see why such invariants are *required*, and later show how they may be systematically *discovered and verified*.

The four steps of our parallelization methodology are now elaborated, using *mis* (the auxiliary function of *mss*) as our running example.

Step 1 : *R-Contexts Extraction & Normalization*

The first step is to extract out R-contexts for both the recursive call and each of the accumulative arguments. In the case of *mis*, we have the following R-context.

$$(\hat{\lambda}\langle\underline{\bullet}\rangle.\ \textbf{if}\ \alpha_0 + \underline{\bullet} \le \alpha_1\ \textbf{then}\ \underline{\bullet} + \alpha_2\ \textbf{else}\ \alpha_3\ \ )\textbf{where}\ \ \alpha_i\ =\ x\ \ \forall\, i\ \in\ 0\ \ldots\ 3$$

We use the notation $\hat{\lambda}\langle\underline{\bullet}\rangle.\cdots$ to represent an R-context parameterized by a special variable $\underline{\bullet}$, is known as the *R-hole*. $\underline{\bullet}$ captures the position of the recursive call/accumulative parameter. Also, *context variables* (e.g. $\{\alpha_i\}_{i\,\in\,0..3}$) are used to capture each maximal subterm that does not contain $\underline{\bullet}$.

To obtain a simpler R-context, we apply a normalization process to the original R-context. This process performs a series of semantic-preserving transformation to the R-context, aiming at reducing the *depths* & *occurrences* of the R-hole, and the *occurrences* of context variables. The heuristics used by the transformation is as follows.

Definition 3: *Heuristics for Context Normalization*
Consider a R-context with one or more R-holes. Our normalization attempts to:

– Minimise the *depth*[4] of the *R-holes* or their *proxies*. (A *proxy* is a local variable which denotes either a R-hole or its component – if the R-hole is a tuple.)
– Minimise the *number of occurrences* of the *context variables*, where possible.
– Minimise the *number of occurrences* of the *R-holes* or their *proxies*. □

A variety of laws, such as *associativity* and *distributivity*, will be employed in the normalization process. These laws are assumed to be supplied by the user beforehand. In the case of *mis*, we can normalize to obtain the following simpler R-context:

$$\hat{\lambda}\langle\underline{\bullet}\rangle.\ \textbf{if}\ \underline{\bullet} \leq \alpha_1\ \textbf{then}\ \underline{\bullet} + \alpha_2\ \textbf{else}\ \alpha_3 \quad \textbf{where}\ \alpha_1 = 0;\ \alpha_2 = x;\ \alpha_3 = x$$

Finally, we parameterise the R-context with respect to context variables, and obtain a *skeletal R-context*, $\mathcal{R}_{mis}$, defined by:

$$\mathcal{R}_{mis}(\alpha_1, \alpha_2, \alpha_3) \overset{def}{\equiv} \hat{\lambda}\langle\underline{\bullet}\rangle.\ \textbf{if}\ \underline{\bullet} \leq \alpha_1\ \textbf{then}\ \underline{\bullet} + \alpha_2\ \textbf{else}\ \alpha_3$$

Such a normalization is helpful in the following ways:

– It can faciliate *context preservation* in Step 2.
– It can minimise the occurrences of context variables. This can help reduce unnecessary auxiliary functions that are synthesized in Step 3.

Step 2 : *Context Preservation*

Our parallelization methodology is based on generalizing from sequential examples. This method relies on finding a second sequential equation with the same skeletal R-context as the first equation. The second equation can be obtained by unfolding the recursive call once, as follows.

$$\begin{aligned}
&f(x : (y : xs), \lfloor v_i \rfloor_{i=1}^{n}) \\
&\quad = \sigma_1 Er\langle \sigma_2 Er\langle f(xs, \lfloor \sigma_2\ Dr_i\langle \sigma_1\ Dr_i\langle v_i\rangle\rangle \rfloor_{i=1}^{n})\rangle\rangle \\
&\quad = (\sigma_1 Er\ \circ\ \sigma_2 Er)\ \langle f(xs, \lfloor (\sigma_2\ Dr_i\ \circ\ \sigma_1\ Dr_i)\langle v_i\rangle \rfloor_{i=1}^{n})\rangle \\
&\quad \textbf{where}\ \sigma_1 = [xs \mapsto y : xs];\ \sigma_2 = [x \mapsto y, \lfloor v_i \mapsto (\sigma_1\ Dr_i)\langle v_i\rangle \rfloor_{i=1}^{n}]
\end{aligned}$$

In the above, $(\sigma\ Er)$ performs substitution of variables in the context Er by σ. Also, $(\sigma_1 Er\ \circ\ \sigma_2 Er)$ denotes context composition, in the obvious manner. In the case of *mis*, we obtain:

$$\begin{aligned}
mis(x : (y : xs)) &= ((\hat{\lambda}\langle\underline{\bullet}\rangle.\ \textbf{if}\ \underline{\bullet} \leq\ 0\ \textbf{then}\ \underline{\bullet} + x\ \textbf{else}\ x\ ) \\
&\qquad \circ (\hat{\lambda}\langle\underline{\bullet}\rangle.\ \textbf{if}\ \underline{\bullet} \leq\ 0\ \textbf{then}\ \underline{\bullet} + y\ \textbf{else}\ y\ ))\ \langle mis(xs)\rangle \\
&= (\ \mathcal{R}_{mis}(0, x, x)\ \circ\ \mathcal{R}_{mis}(0, y, y))\ \langle mis(xs)\rangle
\end{aligned}$$

Note that the respective R-contexts have been *replicated* by this unfold. In order to check if the second equation has the same skeletal R-context structure

[4] Depth is defined to be the distance from the root of an expression tree. For example, the depths of variable occurrences *c,x,xs,c* in *(c+(x+sum(xs,c)))* are 1,2,3,3 respectively.

as the first equation, we must check if the following *context preservation* property holds for each of our R-contexts.

Definition 4: *Invariant-Based Context Preservation*

A R-context E is said to be *preserved modulo replication* under an *invariant* P, if there exists a skeletal R-context $\mathcal{R}$ such that the following holds:

$$
\begin{array}{l}
(E \Rightarrow_{\eta}{}^{\star}\ \mathcal{R}(t_i)_{i \in N})\ \wedge\ P(t_i)_{i \in N} \wedge \\
(((\ \mathcal{R}(\alpha_i)_{i \in N}\ \circ\ \mathcal{R}(\beta_i)_{i \in N})\ \mathbf{st}\ (P(\alpha_i)_{i \in N}\ \wedge\ P(\beta_i)_{i \in N})) \\
\quad\quad \Rightarrow_{\eta(\mathcal{R})}{}^{\star}\ (\ \mathcal{R}(\Omega_i)i\ \in\ N\ \ \mathbf{st}\ P(\Omega_i)_{i \in N}))
\end{array}
$$

where both $\Rightarrow_{\eta}{}^{\star}$ and $\Rightarrow_{\eta(\mathcal{R})}{}^{\star}$ denote respectively a series of normalization transformation. The former aims at normalizing the R-context E, as described in Step 1 above; the latter aims at transforming the replicated R-context to the desired skeletal R-context $\mathcal{R}$. α_i and β_i are context variables, and Ω_i denote subterms not containing R-hole, nor its proxies. The skeletal R-context $\mathcal{R}$ is said to be the *common context* despite replication/unfolding. Notationwise, $(e\ \ \mathbf{st}\, p)$ denotes an expression e that satisfies the invariant p. □

The term $P(t_i)_{i \in N}$ specifies that the normalized R-context $\mathcal{R}\langle t_i \rangle_{i \in N}$ satisfies the invariant P; we call it the *pre-condition* for R-context. The other three terms on P ensure that all new R-context generated during context-preserving transformation also satisfy P; we call this the *invariance condition* for the invariant across R-context replication. This condition can be independently expressed as: $P(\alpha_i)_{i \in N}\ \wedge\ P(\beta_i)_{i \in N}\ \Rightarrow\ P(\Omega_i)_{i \in N}$.

In the case of *mis*, we claim that $\mathcal{R}_{mis}(\alpha_1, \alpha_2, \alpha_3)$ is preserved modulo replication under the invariant $P_{mis}(\alpha_1, \alpha_2, \alpha_3) \stackrel{def}{\equiv}\ \alpha_3 \geq\ \alpha_1\ +\ \alpha_2$. We will describe how such an invariant is discovered (and verified) in Section 4. Here, we illustrate how P_{mis} can be used to preserve R-context during transformation. We begin with the composition $\mathcal{R}_{mis}(\alpha_1,\ \alpha_2, \alpha_3) \circ \mathcal{R}_{mis}(\beta_1, \beta_2, \beta_3)$, assuming that the composition satisfies the invariance property $P'_{mis} \stackrel{def}{\equiv}\ P_{mis}(\alpha_1, \alpha_2, \alpha_3)\ \wedge$ $P_{mis}(\beta_1, \beta_2, \beta_3)$. Note that at each transformation step, the expression under transformation satisfies P'_{mis}. Due to space constraint, we omit writing P'_{mis} in these steps, except for the first expression.

$$
\begin{array}{l}
((\hat{\lambda}\langle\underline{\bullet}\rangle.\ \mathbf{if}\ \underline{\bullet} \leq \alpha_1\ \mathbf{then}\ \underline{\bullet} + \alpha_2\ \mathbf{else}\ \alpha_3\ )\circ(\hat{\lambda}\langle\underline{\bullet}\rangle.\ \mathbf{if}\ \underline{\bullet} \leq \beta_1\ \mathbf{then}\ \underline{\bullet} + \beta_2\ \mathbf{else}\ \beta_3)\ ) \\
\quad\quad \mathbf{st}\ (\alpha_3\ \geq\ \alpha_1\ +\ \alpha_2)\ \wedge (\beta_3\ \geq\ \beta_1\ +\ \beta_2)
\end{array}
$$

$$
\begin{array}{l}
\Rightarrow_{\eta(\mathcal{R}_{mis})} \quad\quad \{\text{definition of } \circ\} \\
\hat{\lambda}\langle\underline{\bullet}\rangle.\ \mathbf{if}\ (\mathbf{if}\ \underline{\bullet} \leq \beta_1\ \mathbf{then}\ \underline{\bullet} + \beta_2\ \mathbf{else}\ \beta_3) \leq \alpha_1\ \mathbf{then}\ (\mathbf{if}\ \underline{\bullet} \leq \beta_1\ \mathbf{then}\ \underline{\bullet} + \beta_2\ \mathbf{else}\ \beta_3) + \alpha_2 \\
\quad\quad \mathbf{else}\ \alpha_3
\end{array}
$$

$$
\begin{array}{l}
\Rightarrow_{\eta(\mathcal{R}_{mis})} \quad\quad \{\text{float out } (\underline{\bullet}\ \leq\ \beta_1)\ \&\ \text{simplify}\} \\
\hat{\lambda}\langle\underline{\bullet}\rangle.\ \mathbf{if}\ \underline{\bullet} \leq \beta_1\ \mathbf{then}\ \ \mathbf{if}\ \underline{\bullet} + \beta_2 \leq \alpha_1\ \mathbf{then}\ \underline{\bullet} + \beta_2 + \alpha_2\ \mathbf{else}\ \alpha_3 \\
\quad\quad \mathbf{else\ if}\ \beta_3 \leq \alpha_1\ \mathbf{then}\ \beta_3 + \alpha_2\ \mathbf{else}\ \alpha_3
\end{array}
$$

$\Rightarrow_{\eta(\mathcal{R}_{mis})}$ $\{$flatten nested *if*$\}$

$$\hat{\lambda}\langle\underline{\bullet}\rangle.\mathbf{if}\begin{cases}(\underline{\bullet}\le\beta_1)\ \wedge\ (\underline{\bullet}+\beta_2\le\alpha_1) & \to\ \underline{\bullet}+\beta_2+\alpha_2\\(\underline{\bullet}\le\beta_1)\ \wedge\ \neg(\underline{\bullet}+\beta_2\le\alpha_1) & \to\ \alpha_3\\\neg(\underline{\bullet}\le\beta_1)\ \wedge\ (\beta_3\le\alpha_1) & \to\ \beta_3+\alpha_2\\\neg(\underline{\bullet}\le\beta_1)\ \wedge\ \neg(\beta_3\le\alpha_1) & \to\ \alpha_3\end{cases}$$

$\Rightarrow_{\eta(\mathcal{R}_{mis})}$ $\{$combine $\le$ & regroup **if**$\}$

$$\begin{array}{l}\hat{\lambda}\langle\underline{\bullet}\rangle.\mathbf{if}\ \underline{\bullet}\le\ min2(\beta_1,\alpha_1-\beta_2)\ \mathbf{then}\ \underline{\bullet}+\beta_2+\alpha_2\ \mathbf{else}\\\qquad\mathbf{if}\begin{cases}(\underline{\bullet}\le\beta_1)\ \wedge\ \neg(\underline{\bullet}+\beta_2\le\alpha_1) & \to\ \alpha_3\\\neg(\underline{\bullet}\le\beta_1)\ \wedge\ (\beta_3\le\alpha_1) & \to\ \beta_3+\alpha_2\\\neg(\underline{\bullet}\le\beta_1)\ \wedge\ \neg(\beta_3\le\alpha_1) & \to\ \alpha_3\end{cases}\end{array}$$

$\Rightarrow_{\eta(\mathcal{R}_{mis})}$ $\{$float nonrec $(\beta_3\le\alpha_1)$ & simplify$\}$

$$\begin{array}{l}\hat{\lambda}\langle\underline{\bullet}\rangle.\mathbf{if}\ \underline{\bullet}\le\ min2(\beta_1,\alpha_1-\beta_2)\ \mathbf{then}\ \underline{\bullet}+\beta_2+\alpha_2\ \mathbf{else}\\\qquad\mathbf{if}\ \beta_3\le\alpha_1\ \mathbf{then}\ \boxed{\mathbf{if}\begin{cases}(\underline{\bullet}\le\beta_1)\ \wedge\ \neg(\underline{\bullet}+\beta_2\le\alpha_1) & \to\ \alpha_3\\\neg(\underline{\bullet}\le\beta_1) & \to\ \beta_3+\alpha_2\end{cases}}\ \mathbf{else}\ \alpha_3\end{array}$$

$\Rightarrow_{\eta(\mathcal{R}_{mis})}$ $\{$simplify above box to $\beta_3+\alpha_2$, using $P'_{mis}\}$

$$\begin{array}{l}\hat{\lambda}\langle\underline{\bullet}\rangle.\mathbf{if}\ \underline{\bullet}\le\ min2(\beta_1,\alpha_1-\beta_2)\ \mathbf{then}\ \underline{\bullet}+\beta_2+\alpha_2\ \mathbf{else}\\\qquad\mathbf{if}\ \beta_3\le\alpha_1\ \mathbf{then}\ \boxed{\beta_3+\alpha_2}\ \mathbf{else}\ \alpha_3\end{array}$$

$\Rightarrow_{\eta(\mathcal{R}_{mis})}$ $\{$extract$\}$

$$\begin{array}{l}\hat{\lambda}\langle\underline{\bullet}\rangle.\ \mathbf{if}\ \underline{\bullet}\ \le\ \Omega_1\ \mathbf{then}\ \underline{\bullet}\ +\ \Omega_2\ \mathbf{else}\ \Omega_3\\\quad\mathbf{where}\quad\Omega_1\ =min2(\beta_1,\ \alpha_1-\beta_2)\ ;\ \Omega_2\ =\ \beta_2\ +\ \alpha_2\ ;\\\qquad\qquad\quad\Omega_3\ =\mathbf{if}\ \beta_3\ \le\ \alpha_1\ \mathbf{then}\ \beta_3\ +\ \alpha_2\ \mathbf{else}\ \alpha_3\end{array}$$

The second last transformation step above is valid because the test $(\underline{\bullet}\le\beta_1)\ \wedge\ \neg(\underline{\bullet}+\beta_2\le\alpha_1)$ is false under the condition that $\underline{\bullet}\ >\ min2(\beta_1,\alpha_1-\beta_2), \beta_3\ \le\ \alpha_1$, and P'_{mis} are true. In the last step, we obtain $\mathcal{R}_{mis}(\Omega_1, \Omega_2, \Omega_3)$. We state without proof the validity of the invariance property:

$P_{mis}(\alpha_1,\alpha_2,\alpha_3)\ \wedge\ P_{mis}(\beta_1,\beta_2,\beta_3)\ \Rightarrow\ P_{mis}(\Omega_1,\Omega_2,\Omega_3)$.

The context-preserving transformation process described above is similar to the normalization process in that we aim to simplify the R-context. However, the former process is performed with a specific goal in mind: producing $\mathcal{R}_{mis}$. Goal-directed transformation can be effectively carried out by a technique called *rippling* [BvHSI93], that repeatedly minimises the difference between actual expression and the targeted skeletal R-context, $\mathcal{R}_{mis}$. The detail of this technique will be described in a forthcoming paper.

Step 3 : *Auxiliary Function Synthesis*

Successful context preservation ensures that a parallel equation can be derived from its sequential counterpart. This assurance was proven in [CTH98]. To synthesise the parallel equation for *mis*, we perform a second-order generalisation to obtain the following:

$$mis(xr+\!\!+xs)\ =\ \mathbf{if}\ mis(xs)\ \le uH(xr)\ \mathbf{then}\ mis(xs)+uG(xr)\ \mathbf{else}\ uJ(xr)$$

The RHS is essentially similar to R-contexts of mis, with the exception of new auxiliary functions (the second-order variables) uH, uG and uJ, to replace each of the earlier context variables, $\{\alpha_i\}_{i \in 1..3}$. Such functions are initially unknown. We apply an inductive derivation procedure to synthesize their definitions. For base case where $xr = [x]$, inductive derivation yields:

$$uH([x]) = 0 \ ; \ uG([x]) = x \ ; \ uJ([x]) = x$$

For the inductive case we set $xr = xa{+\!\!+}xb$, inductive derivation yields:

$$\begin{array}{ll} uH(xa{+\!\!+}xb) & = min2(uH(xb),\ uH(xa)\ -\ uG(xb)) \\ uG(xa{+\!\!+}xb) & = uG(xb)\ +\ uG(xa) \\ uJ(xa{+\!\!+}xb) & = \textbf{if } uJ(xb)\ \leq\ uH(xa) \textbf{ then } uJ(xb) + uG(xa) \textbf{ else } uJ(xa) \end{array}$$

The above result is essentially equivalent to the following substitutions:

$$\begin{array}{lll} \alpha_1 = uH(xa) & \alpha_2 = uG(xa) & \alpha_3 = uJ(xa) \\ \beta_1 = uH(xb) & \beta_2 = uG(xb) & \beta_3 = uJ(xb) \\ \Omega_1 = uH(xa{+\!\!+}xb) & \Omega_2 = uG(xa{+\!\!+}xb) & \Omega_3 = uJ(xa{+\!\!+}xb) \end{array}$$

The corresponding parallel definitions are:

$$\begin{array}{ll} mis([x]) & = x \\ mis(xr{+\!\!+}xs) & = \textbf{if } mis(xs)\ \leq uH(xr) \textbf{ then } mis(xs) + uG(xr) \textbf{ else } uJ(xr) \\ & \\ uH([x]) & = 0 \\ uH(xr{+\!\!+}xs) & = min2(uH(xs), uH(xr) - uG(xs)) \\ & \\ uG([x]) & = x \\ uG(xr{+\!\!+}xs) & = uG(xs)\ +\ uG(xr) \\ & \\ uJ([x]) & = x \\ uJ(xr{+\!\!+}xs) & = \textbf{if } uJ(xs)\ \leq uH(xr) \textbf{ then } uJ(xs) + uG(xr) \textbf{ else } uJ(xr) \end{array}$$

Some of the functions synthesized may be identical to previously known functions. For example, uJ is identical to mis itself. Such duplicate functions can be detected syntactically and eliminated.

Step 4 : *Tupling*

While the equations derived may be parallel, they may be inefficient due to the presence of redundant function calls. To remove this inefficiency, we perform tupling transformation.

For mis, we remove its redundant calls by introducing the following tupled function which returns multiple results.

$$mistup(xs)\ =\ (mis(xs), uH(xs), uG(xs))$$

After tupling, we can obtain the following efficient parallel program, whereby duplicate calls are re-used, rather than re-computed. Note how m_b, h_a, g_a, g_b are used multiple times in the second recursive equation.

$$
\begin{array}{l}
mistup([x]) \quad\quad = \; (0,\; x,\; x) \\
mistup(xa{+\!\!+}xb) \; = \\
\quad \textbf{let}\ \{(m_a, h_a, g_a) = mistup(xa); (m_b, h_b, g_b) = mistup(xb)\} \\
\quad \textbf{in}\ \big(\,(\,\textbf{if}\ m_b \leq h_a\ \textbf{then}\ m_b + g_a\ \textbf{else}\ m_a\ \,), min2(h_b, h_a - g_b), g_b + g_a\,\big)
\end{array}
$$

4 Discovering Invariants

In general, it is non-trivial to preserve conditional R-context, particularly if it has multiple R-holes. This is because the number of R-holes may multiply after context replication. Our proposal to curbing such growth is to exploit *invariant* during normalization. This new technique generalises our earlier method for parallelization, and significantly widens its scope of application.

We have shown in Section 3 how an invariant can be used to achieve context presevation. It remains to be seen how an invariant can be discovered. Instead of relying on the user to provide an invariant, we achieve this eureka step by employing constraint-solving techniques to systematically generate and verify the invariant.

Invariant is originated from the need to facilitate normalization process during context preservation. Specifically, constraints may be added to achieve normalization; these constraints constitute the invariant. We have identified two scenarios under which constraints may be needed during normalization, namely: *conditional laws* and *conditional elimination*.

4.1 Conditional Laws

Some laws are conditional in nature. For example, the following four distributive laws for $*$ over $min2$ and $max2$ are conditional upon the sign of input c.

$$
\begin{array}{lcll}
c{*}max2(a,b) & = & max2(c{*}a,\, c{*}b) & \textbf{if}\ c \geq 0 \\
c{*}max2(a,b) & = & min2(c{*}a,\, c{*}b) & \textbf{if}\ c \leq 0 \\
c{*}min2(a,b) & = & min2(c{*}a,\, c{*}b) & \textbf{if}\ c \geq 0 \\
c{*}min2(a,b) & = & max2(c{*}a,\, c{*}b) & \textbf{if}\ c \leq 0
\end{array}
$$

Before these laws can be used in normalization process, we require their corresponding conditions to be satisfied. These conditions may become the invariant for our R-context. Of course, we need to verify that they can be satisfied as a pre-condition, and they obey the invariance property. An example of how such conditional laws are applied is illustrated later in Section 5.

4.2 Conditional Elimination

During context preservation process, we may wish to eliminate some conditional branches so as to reduce the number of R-holes. A branch can be eliminatd by identifying constraint that is known *not* to be true at the branch. This constraint may become the invariant for the corresponding R-context.

We would have encountered this situation in Section 3, if we tried to preserve the contextual form of *mis* function, without any knowledge of invariant. We repeat the problematic intermediate step of the transformation here:

$$\hat{\lambda}\langle\underline{\bullet}\rangle.\textbf{if } \underline{\bullet} \leq \ min2(\beta_1, \alpha_1 - \beta_2) \textbf{ then } \underline{\bullet} + \beta_2 + \alpha_2 \textbf{ else}$$
$$\textbf{if } \beta_3 \leq \alpha_1 \textbf{ then} \boxed{\textbf{if} \begin{cases} (\underline{\bullet} \leq \beta_1) \ \wedge \ \neg(\underline{\bullet} + \beta_2 \leq \alpha_1) \ \rightarrow \ \alpha_3 \\ \neg(\underline{\bullet} \leq \beta_1) \qquad\qquad\qquad\quad \rightarrow \ \beta_3 + \alpha_2 \end{cases}} \textbf{ else } \alpha_3$$

At this step, there are five occurrences of $\underline{\bullet}$, instead of two in the original R-context. The three extraneous occurrences of $\underline{\bullet}$ can be found in the *boxed* branch of the conditional shown in the last step of normalization.

A way to eliminate these redundant occurrences of the R-hole is to eliminate one of these two branches (and thus make the test in the remaining branch unnecessary). We therefore look for an invariant that enables such elimination. A technique we have devised is to gather the conditions associated with the two branches and attempt to find a constraint (exclusively in terms of either $\{\alpha_1, \alpha_2, \alpha_3\}$ or $\{\beta_1, \beta_2, \beta_3\}$) that holds for either branch. If found, the corresponding branch may be eliminated by using the *negated constraint as invariant.* This constraint-searching technique is formulated as $Ctx \vdash_B c$ where Ctx denotes the condition associated with a branch, and c represents the desired constraint expressed exclusively in terms of variables from B.

In the first branch, we obtain $\neg(\beta_3 < \beta_1 + \beta_2)$ as a candidate for invariant:

$$\begin{array}{r} \neg(\underline{\bullet} \leq \ min2(\beta_1, \alpha_1 - \beta_2)) \wedge \beta_3 \leq \alpha_1 \wedge (\underline{\bullet} \leq \beta_1) \wedge \neg(\underline{\bullet} + \beta_2 \leq \alpha_1) \\ \vdash_{\{\beta_1, \beta_2, \beta_3\}} \beta_3 < \beta_1 + \beta_2 \end{array} \qquad (1)$$

In the second branch, we find no candidate:

$$\begin{array}{r} \neg(\underline{\bullet} \leq \ min2(\beta_1, \alpha_1 - \beta_2)) \wedge \beta_3 \leq \alpha_1 \wedge \neg(\underline{\bullet} \leq \beta_1) \\ \vdash_{\{\beta_1, \beta_2, \beta_3\}} \text{ no constraint found} \end{array} \qquad (2)$$

If $\neg(\beta_3 < \beta_1 + \beta_2)$ is the invariant (and indeed it is), then the context-preservation process can proceed, yielding the desired R-context form. Invariant validation, as well as the discovery of invariant, is the job of constraint solver.

4.3 Constraint Solving via CHR

A convenient tool for solving constraints is the *Constraint Handling Rules* (CHR) developed by Frühwirth [Frü98]. Using CHR, we can build tiny but specialised constraint-solvers for operators that handle variable arguments. Currently, we run CHR on top of a Prolog system (named ECL^{i}PS^e Prolog).

In this section, we demonstrate the use of CHR to help discover and verify the invariant found in Section 4.2. The CHR rules defined for this problem are given Appendix A. To discover the invariant for *mis* through conditional elimination, we supply the context of rule (1) (*ie.*, the premises) as a prolog program to ECL^{i}PS^e Prolog (Here, `A`i and `B`i represent α_i and β_i respectively; `H` denotes $\underline{\bullet}$. The predicates used are self-explanatory):

```
br1(A1,A2,A3,B1,B2,B3,H) :- minus(A1,B2,T), min2(B1,T,U), gt(H,U),
          le(B3,A1), le(H,B1), add(H,B2,S), gt(S,A1).
```

We then ask Prolog to show us those constraints that are consistent with the context. Following is the session (shortened for presentation sake) we have with Prolog. Note that constraints `(19)`, `(21)`, and `(41)` jointly infer that $\beta_3 < \beta_1 + \beta_2$:

```
[eclipse 6]: br1(A1,A2,A3,B1,B2,B3,H).
Constraints:
(19) le(H_892, B1_754)
(21) add(H_892, B2_296, _1397)
(41) lt(B3_986, _1397)
yes.
[eclipse 7]:
```

To verify that $\neg(\beta_3 < \beta_1 + \beta_2)$ is an invariant, we first verify its pre-condition. Referring to the initial R-context of *mis*, we use CHR to verify the following proposition:

$$(\alpha_1 = 0) \wedge (\alpha_2 = x) \wedge (\alpha_3 = x) \vdash (\alpha_3 \geq \alpha_1 + \alpha_2). \qquad (3)$$

To verify the invariance condition, we feed CHR with the following formula:

$$(\alpha_3 \geq \alpha_1 + \alpha_2) \wedge (\beta_3 \geq \beta_1 + \beta_2) \vdash (\Omega_3 \geq \Omega_1 + \Omega_2) \qquad (4)$$

Algorithmically, we prove the validity of both formulae (3) and (4) by a standard technique called *refutation*. That is, we attempt to find a condition under which the negation of a formula is true. Failing to do so, we conclude that the formula is true. Following is the Prolog program for verifying formula (4):

```
premise(V1,V2,V3) :- add(V1,V2,R), le(R,V3).   % generic premise
omega1(A1,B1,B2,R) :- minus(A1,B2,T), min2(B1,T,R).
omega2(A2,B2,R) :- add(A2,B2,R).
omega3(A1,A2,A3,B3,R) :- le(B3,A1), add(A2,B3,R).
omega3(A1,A2,A3,B3,R) :- gt(B3,A1), R=A3.
neg_inv(A1,A2,A3,B1,B2,B3,R1,R2,R3) :-          % Negated formula
 premise(A1,A2,A3), premise(B1,B2,B3), omega1(A1,B1,B2,R1),
 omega2(A2,B2,R2), omega3(A1,A2,A3,B3,R3), add(R1,R2,RR), gt(RR,R3).
```

Following is extracted from a session with ECL^{i}PS^e Prolog:

```
[eclipse 7]: neg_inv(A1,A2,A3,B1,B2,B3,R1,R2,R3).
no (more) solution.
[eclipse 8]:
```

5 MSP : A Bigger Example

Our parallelization method is not just a nice theoretical result. It is also practically useful for parallelizing more complex programs. In particular, we could systematically handle recursive functions with conditional and tupled constructs that are often much harder to parallelize. Let us examine a little known problem, called *maximum segment product* [Ben86], whose parallelization requires deep human insights otherwise.

Given an input list $[x_1, \ldots, x_n]$, we are interested to find the maximum product of all non-empty (contiguous) segments, of the form $[x_i, x_{i+1}, \ldots, x_j]$ where $1 \le i \le j \le n$. A high-level specification of *msp* is the following generate-and-test algorithm.

$$msp(xs) \quad = \quad max(map(prod, segs(xs)))$$

Here, $segs(xs)$ returns all segments for an input list xs, while $map(prod, segs(xs))$ applies *prod* to each sublist from $segs(xs)$, before *max* chooses the largest value. While clear, this specification is grossly inefficient. However, it can be transformed by fusion [Chi92a,TM95,HIT96] to a sequential algorithm. The transformed *msp*, together with two auxiliary functions, *mip*, and *mipm*, were given in Figure 1.

Functions *mip* and *mipm* are mutually recursive and not in LSR-form. Nevertheless, we could use the automated tupling method of [Chi93,HITT97] to obtain a tupled definition of *miptup*, as elaborated earlier in Section 2.

We focus on the parallelization of *miptup* as it must be parallelized before its parent *msp* function. We could proceed to extract its initial R-context[5] (shown below) to check if it could be parallelized.

$$\begin{array}{l}
\hat{\lambda}\,\langle\underline{\bullet}\rangle.\,\textbf{let}\,(u,v) \;=\; \underline{\bullet}\;\textbf{in}\;\;\textbf{if}\;\alpha_1 \;>\; 0\;\textbf{then}\;(max2(\alpha_2,\alpha_3{*}u),\; min2(\alpha_4,\alpha_5{*}v)) \\
\qquad\qquad\qquad\qquad\qquad\qquad\;\textbf{else}\;(max2(\alpha_6,\alpha_7{*}v),\; min2(\alpha_8,\alpha_9{*}u)) \\
\quad\textbf{where}\;\alpha_1 \;=\; x\;;\;\alpha_2 \;=\; x\;;\;\alpha_3 = x\;\;;\alpha_4 \;=\; x\;;\;\alpha_5 \;=\; x\;; \\
\qquad\qquad\;\alpha_6 \;=\; x\;;\;\alpha_7 \;=\; x\;;\;\alpha_8 \;=\; x\;;\;\alpha_9 = x
\end{array}$$

Note the use of local variables u, v as proxies for the R-hole. The depth and occurrences of these proxies should thus be minimised, where possible, during normalization. Application of context preservation can proceed as follows.

$$\begin{array}{l}
(\;\hat{\lambda}\,\langle\underline{\bullet}\rangle.\,\textbf{let}\,(u,v) \;=\; \underline{\bullet}\;\textbf{in}\;\;\textbf{if}\;\alpha_1 \;>\; 0\;\textbf{then}\;(max2(\alpha_2,\alpha_3{*}u), min2(\alpha_4,\alpha_5{*}v)) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad\textbf{else}\;(max2(\alpha_6,\alpha_7{*}v), min2(\alpha_8,\alpha_9{*}u))) \\
\quad\circ\,(\;\hat{\lambda}\,\langle\underline{\bullet}\rangle.\,\textbf{let}\,(u,v) \;=\; \underline{\bullet}\;\textbf{in}\;\;\textbf{if}\;\beta_1 \;>\; 0\;\textbf{then}\;(max2(\beta_2,\beta_3{*}u), min2(\beta_4,\beta_5{*}v)) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\textbf{else}\;(max2(\beta_6,\beta_7{*}v), min2(\beta_8,\beta_9{*}u)))
\end{array}$$

[5] Note that the skeletal R-context always have its variables uniquely re-named to help support reusability and the context preservation property.

$\Rightarrow_\eta$ {tupled & conditional normalization}

$$\hat{\lambda}\,\langle\underline{\bullet}\rangle.\textbf{let}\ \ (u,v)\ =\ \underline{\bullet}\ \ \textbf{in}$$
$$\textbf{if}\begin{cases}(\beta_1>0)\wedge(\alpha_1>0)\\ \quad\rightarrow\ (\ max2(\alpha_2,\alpha_3{*}max2(\beta_2,\beta_3{*}u))\ ,min2(\alpha_4,\alpha_5{*}min2(\beta_4,\beta_5{*}v))\ )\\ (\beta_1>\ 0)\wedge\neg\,(\alpha_1>0)\\ \quad\rightarrow\ (max2(\alpha_6,\alpha_7{*}min2(\beta_4,\beta_5{*}v))\ ,min2(\alpha_8,\alpha_9{*}max2(\beta_2,\beta_3{*}u))\ )\\ \neg\,(\beta_1\ >\ 0)\wedge(\alpha_1\ >\ 0)\\ \quad\rightarrow\ (max2(\alpha_2,\alpha_3{*}max2(\beta_6,\beta_7{*}v))\ ,min2(\alpha_4,\alpha_5{*}min2(\beta_8,\beta_9{*}u))\ )\\ \neg\,(\beta_1\ >\ 0)\wedge\neg(\alpha_1\ >\ 0)\\ \quad\rightarrow\ (max2(\alpha_6,\alpha_7{*}min2(\beta_8,\beta_9{*}u))\ ,min2(\alpha_8,\alpha_9{*}max2(\beta_6,\beta_7{*}v))\ )\end{cases}$$

To normalize further, we need to distribute $*$ into $max2$ and $min2$. This could only be done with the set of distributive laws provided in Section 4.1.

Each of these laws has a *condition* attached to it. If this condition is not present in the R-context, we must add them as required constraint before the corresponding distributive law can be applied (Sec 4.1). Doing so results in the following successful context preservation.

$\Rightarrow_\eta$ {add selected constraints & normalize further}

$$\hat{\lambda}\,\langle\underline{\bullet}\rangle.\textbf{let}\ \ (u,v)\ =\ \underline{\bullet}\ \ \textbf{in}$$
$$\textbf{if}\begin{cases}(\beta_1\ >\ 0)\wedge(\alpha_1>0)\\ \quad\rightarrow\begin{pmatrix}max2(max2(\alpha_2,\alpha_3{*}\beta_2),(\alpha_3{*}\beta_3){*}u)),\\ min2(min2(\alpha_4,\alpha_5{*}\beta_4),(\alpha_5{*}\beta_5){*}v)\end{pmatrix}\\ (\beta_1\ >\ 0)\wedge\neg\,(\alpha_1\ >\ 0)\\ \quad\rightarrow\begin{pmatrix}max2(max2(\alpha_6,\alpha_7{*}\beta_4),(\alpha_7{*}\beta_5){*}v)),\\ min2(min2(\alpha_8,\alpha_9{*}\beta_2),(\alpha_9{*}\beta_3){*}u)\end{pmatrix}\\ \neg\,(\beta_1\ >\ 0)\wedge(\alpha_1\ >\ 0)\\ \quad\rightarrow\begin{pmatrix}max2(max2(\alpha_2,\alpha_3{*}\beta_6),(\alpha_3{*}\beta_7){*}v)),\\ min2(min2(\alpha_4,\alpha_5{*}\beta_8),(\alpha_5{*}\beta_9){*}u)\end{pmatrix}\\ \neg\,(\beta_1\ >\ 0)\wedge\neg(\alpha_1\ >\ 0)\\ \quad\rightarrow\begin{pmatrix}max2(max2(\alpha_6,\alpha_7{*}\beta_8),(\alpha_7{*}\beta_9){*}u)),\\ min2(min2(\alpha_8,\alpha_9{*}\beta_6),(\alpha_9{*}\beta_7){*}v)\end{pmatrix}\end{cases}$$
$$\textbf{st}\ \{\alpha_3\ \geq\ 0;\ \alpha_5\ \geq\ 0;\ \alpha_7\ \leq\ 0;\ \alpha_9\ \leq\ 0\,\}$$

$\Rightarrow_\eta$ {re−group branches & form skeletal R−ctx }

$$\begin{aligned}&\hat{\lambda}\,\langle\underline{\bullet}\rangle.\textbf{let}\ \ (u,v)\ =\ \underline{\bullet}\ \ \textbf{in if}\ \Omega_1>0\ \textbf{then}\ (max2(\Omega_2,\Omega_3{*}u),\ min2(\Omega_4,\Omega_5{*}v))\\ &\qquad\qquad\qquad\qquad\qquad\quad\ \textbf{else}\ (max2(\Omega_6,\Omega_7{*}v),\ min2(\Omega_8,\Omega_9{*}u))\\ &\textbf{where}\ \Omega_1\ =\alpha_1{*}\beta_1\\ &\qquad\quad\ \Omega_2\ =\textbf{if}\ \alpha_1\ >\ 0\ \textbf{then}\ max2(\alpha_2,\alpha_3{*}\beta_2)\ \textbf{else}\ max2(\alpha_6,\alpha_7{*}\beta_8)\\ &\qquad\quad\ \Omega_3\ =\textbf{if}\ \alpha_1\ >\ 0\ \textbf{then}\ \alpha_3{*}\beta_3\ \textbf{else}\ \alpha_7{*}\beta_9\\ &\qquad\quad\ \Omega_4\ =\textbf{if}\ \alpha_1\ >\ 0\ \textbf{then}\ min2(\alpha_4,\alpha_5{*}\beta_4)\ \textbf{else}\ min2(\alpha_8,\alpha_9{*}\beta_6)\\ &\qquad\quad\ \Omega_5\ =\textbf{if}\ \alpha_1\ >\ 0\ \textbf{then}\ \alpha_5{*}\beta_5\ \textbf{else}\ \alpha_9{*}\beta_7\\ &\qquad\quad\ \Omega_6\ =\textbf{if}\ \alpha_1\ >\ 0\ \textbf{then}\ max2(\alpha_2,\alpha_3{*}\beta_6)\ \textbf{else}\ max2(\alpha_6,\alpha_7{*}\beta_4)\end{aligned}$$

$$\begin{aligned}
\Omega_7 &= \textbf{if } \alpha_1 > 0 \textbf{ then } \alpha_3{*}\beta_7 \textbf{ else } \alpha_7{*}\beta_5 \\
\Omega_8 &= \textbf{if } \alpha_1 > 0 \textbf{ then } min2(\alpha_4, \alpha_5{*}\beta_8) \textbf{ else } min2(\alpha_8, \alpha_9{*}\beta_2) \\
\Omega_9 &= \textbf{if } \alpha_1 > 0 \textbf{ then } \alpha_5{*}\beta_9 \textbf{ else } \alpha_9{*}\beta_3
\end{aligned}$$

We can now form an invariant from the constraints collected during transformation. To do so, we take into consideration the conditional context in which these constraints are used. We thus derive at the formula $(\alpha_1 > 0 \Rightarrow \alpha_3 \geq 0 \wedge \alpha_5 \geq 0) \wedge (\alpha_1 \leq 0 \Rightarrow \alpha_7 \leq 0 \wedge \alpha_9 \leq 0)$. We verify that this is indeed an invariant by proving its pre-condition and invariance condition. We omit the detail constraint solving in this paper.

Next, we synthesize the auxiliary functions needed for defining the parallel version of *miptup*. After eliminating duplicated functions, we obtain:

$$\begin{array}{l}
miptup([x]) \quad = (x, x) \\
miptup(xr{+\!\!+}xs) = \\
\quad \textbf{let } (u, v) = miptup(xs) \\
\quad \textbf{in if } uH1(xr) > 0 \textbf{ then} \\
\qquad\qquad (max2(uH2(xr), uH1(xr){*}u), min2(uH4(xr), uH1(xr){*}v)) \\
\qquad \textbf{else } (max2(uH2(xr), uH1(xr){*}v), min2(uH4(xr), uH1(xr){*}u))
\end{array}$$

$$\begin{aligned}
uH1([x]) &= x \\
uH1(xr{+\!\!+}xs) &= uH1(xr){*}uH1(xs)
\end{aligned}$$

$$\begin{aligned}
uH2([x]) &= x \\
uH2(xr{+\!\!+}xs) &= \textbf{if } uH1(xr) > 0 \textbf{ then } max2(uH2(xr), uH1(xr){*}uH2(xs)) \\
&\quad\ \textbf{else } max2(uH2(xr), uH1(xr){*}uH4(xs))
\end{aligned}$$

$$\begin{aligned}
uH4([x]) &= x \\
uH4(xr{+\!\!+}xs) &= \textbf{if } uH1(xr) > 0 \textbf{ then } min2(uH4(xr), uH1(xr){*}uH4(xs)) \\
&\quad\ \textbf{else } min2(uH4(xr), uH1(xr){*}uH2(xs))
\end{aligned}$$

Finally, by tupling the definitions of $uH2$ and $uH4$ together, we obtain a tupled function that is identical to the components of *miptup*. Consequently, we can derive a very compact parallel algorithm shown below.

$$\begin{array}{l}
miptup([x]) \quad = (x, x) \\
miptup(xr{+\!\!+}xs) = \\
\quad \textbf{let } (a, b) = miptup(xr);\ (u, v) = miptup(xs) \\
\quad \textbf{in if } uH1(xr) > 0 \textbf{ then } (max2(a, uH1(xr){*}u), min2(b, uH1(xr){*}v)) \\
\qquad \textbf{else } (max2(a, uH1(xr){*}v), min2(b, uH1(xr){*}u))
\end{array}$$

$$\begin{aligned}
uH1([x]) &= x \\
uH1(xr{+\!\!+}xs) &= uH1(xr){*}uH1(xs)
\end{aligned}$$

With these equations, we can proceed to parallelize the parent function *msptup* using context preservation and normalization. Its parallel equations are:

$$
\begin{array}{lcl}
msp(xr{+\!\!+}xs) & = & \textbf{let } (a,b) = miptup(xr);\ (u,v) = miptup(xs) \\
& & \textbf{in } max2(max2(max2(msp(xr), msp(xs)), \\
& & \qquad\qquad max2(mfp(xr)+a, mfp(xr)+b)), \\
& & \qquad max2(mfpm(xr)+a, mfpm(xr)+b))
\end{array}
$$

$$
\begin{array}{lcl}
mfp([x]) & = & x \\
mfp(xr{+\!\!+}xs) & = & \textbf{if } uH1(xs) > 0 \textbf{ then } max2(mfp(xr)*uH1(xs), mfp(xs)) \\
& & \textbf{else } max2(mfpm(xr)*uH1(xs), mfp(xs))
\end{array}
$$

$$
\begin{array}{lcl}
mfpm([x]) & = & x \\
mfpm(xr{+\!\!+}xs) & = & \textbf{if } uH1(xr) > 0 \textbf{ then } min2(mfpm(xr)*uH1(xs), mfpm(xs)) \\
& & \textbf{else } min2(mfp(xr)*uH1(xs), mfpm(xs))
\end{array}
$$

Tupling can again be applied to obtain a work-efficient parallel program.

6 Related Works

Generic program schemes have been advocated for use in structured parallel programming, both for imperative programs expressed as first-order recurrences through a classic result of [Sto75] and for functional programs via Bird's homomorphism [Ski90]. 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 [Roe91,GDH96] often require deep intuition and the support of ad-hoc lemmas – making automation difficult. Another approach is to provide more specialised schemes, either statically [PP91] or via a procedure [HTC98], that can be directly matched to sequential specification. Though cheap to operate, the generality of this approach is often called into question.

On the imperative language (e.g. Fortran) front, there have been interests in parallelization of reduction-style loops. A work similar to ours was independently conceived by Fischer & Ghoulum [FG94,GF95]. By modelling loops via functions, they noted that function-type values could be reduced (in parallel) via associative function composition. However, the 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 R-context under recursive call unfolding, which we discovered earlier in [Chi92b]. Being based on loops, their framework is less general and less formal. No specific techniques, other than simplification, have been offered for checking if closed template is possible. Also, without invariant handling, their approach is presently limited.

The power of constraints have not escaped the attention of traditional work on finding parallelism in array-based programs. Through the use of constraints, Pugh showed how *exact dependence analysis* can be formulated to support better vectorisation[Pug92]. Our work is complimentary to Pugh's in two respects. Firstly, we may take advantage of practical advances in his constraint technology. Secondly, we tackle a different class of reduction-style sequential algorithms, with inherent dependences across recursion. Thus, instead of checking for the absence of dependence, we transform the sequential dependences into divide-and-conquer counterparts with the help of properties, such as associativity and distributivity. We originally planned to use the Omega calculator for our constraint solving. However, some of our problems (e.g. *msp*) require constraints that fall outside the linear arithmetic class accepted by Pugh's system. This forces us to turn to CHR to build our own specialised constraint solvers.

7 Conclusion

We have presented a systematic methodology for parallelizing sequential programs. The method relies on the successful preservation of replicated R-contexts for the recursive call and each accumulative argument. The notion of context preservation is central to our parallelization method. A key innovation in this paper is the introduction of *invariants* to obtain context preservation. To support this, some constraint-solving techniques have been proposed. Finally, we demonstrated the power of our methodology by applying it to parallelize a non-trivial problem: maximum segment product.

We are currently working on an implementation system to apply context preservation and invariant verification semi-automatically. Apart from the heuristic of minimising both the depths and number of occurrences of R-holes, we have also employed the *rippling* technique [BvHSI93], which has been very popular in automated theorem-proving. It may also be possible for our method to recover from failures when a given R-context could not be preserved. In particular, the resulting context may suggest either a *new* or a *generalized* R-context that could be attempted. This much enhanced potential for parallelization is made possible by our adoption of appropriate strategies and techniques (including constraint handling) for guiding their applications.

8 Acknowledgment

The authors would like to thank the anonymous referees for their comments. This work was supported by the research grant RP3982693.

References

[BCH+93] 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.

[Ben86] Jon Bentley. *Programming Pearls.* Addison-Wesley, 1986.

[Ble90] Guy E. Blelloch. *Vector Models for Data Parallel Computing.* MIT Press, Cambridge, MA, 1990.

[BvHSI93] A. Bundy, F. van Harmelen, A. Smaill, and A. Ireland. Rippling: A heuristic for guiding inductive proofs. *Artificial Intelligence*, 62:185–253, 1993.

[CDG96] 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.

[Chi92a] Wei-Ngan Chin. Safe fusion of functional expressions. In *7th ACM LISP and Functional Programming Conference*, pages 11–20, San Francisco, California, June 1992. ACM Press.

[Chi92b] 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.

[Chi93] Wei-Ngan Chin. Towards an automated tupling strategy. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 119–132, Copenhagen, Denmark, June 1993. ACM Press.

[CTH98] 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.

[FG94] 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.

[Frü98] Thom Frühwirth. Theory and practice of constraint handling rules. *Journal of Logic Programming*, 37((1–3)):95–138, Oct 1998.

[GDH96] Z.N. Grant-Duff and P. Harrison. Parallelism via homomorphism. *Parallel Processing Letters*, 6(2):279–295, 1996.

[GF95] 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.

[HIT96] Z. Hu, H. Iwasaki, and M. Takeichi. Deriving structural hylomorphisms from recursive definitions. In *ACM SIGPLAN International Conference on Functional Programming*, pages 73–82, Philadelphia, Pennsylvannia, May 1996. ACM Press.

[HITT97] Z. Hu, H. Iwasaki, M. Takeichi, and A. Takano. Tupling calculation eliminates multiple traversals. In *2nd ACM SIGPLAN International Conference on Functional Programming*, pages 164–175, Amsterdam, Netherlands, June 1997. ACM Press.

[HTC98] 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.

[PP91] 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.

[Pug92] William Pugh. The omega test: A fast practical integer programming algorithm for dependence analysis. *Communications of ACM*, 8:102–114, 1992.

[Roe91] Paul Roe. *Parallel Programming using Functional Languages (Report CSC 91/R3)*. PhD thesis, University of Glasgow, 1991.

[Ski90] D. Skillicorn. Architecture-independent parallel computation. *IEEE Computer*, 23(12):38–50, December 1990.

[Sto75] Harold S. Stone. Parallel tridiagonal equation solvers. *ACM Transactions on Mathematical Software*, 1(4):287–307, 1975.

[TM95] A. Takano and E. Meijer. Shortcut deforestation in calculational form. In *ACM Conference on Functional Programming and Computer Architecture*, pages 306–313, San Diego, California, June 1995. ACM Press.

A A Tiny Constraint Solver in CHR

In CHR, we define both *simplification* and *propagation* rules for constraints over variables. Simplification rules are used by CHR to replace existing constraints with simpler, logically equivalent, constraints, whereas propagation rules add new constraints which are logically redundant but may cause further simplification. The CHR rules used in the case of *mis* is defined below:

```
%% Rules for Negation.
ne(A,B) <=> A=B | fail.         % "<=>" is a simplification rule
ne(A,B) ==> ne(B,A).            % "==>" is a propagation rule
%% Rules for inequalities.
le(A,B) <=> A=B | true.         % reflexive
le(A,B),le(B,A) <=> A = B.      % antisymmetry
le(A,B),le(B,C) ==> le(A,C).    % transitive

lt(A,B) <=> A=B | fail.         % non-reflexive
lt(A,B),le(B,A) <=> fail.       % asymmetry
le(A,B),lt(B,A) <=> fail.       % asymmetry
lt(A,B),lt(B,A) <=> fail.       % asymmetry
lt(A,B),le(B,C) ==> lt(A,C).    % transitive
le(A,B),lt(B,C) ==> lt(A,C).    % transitive
lt(A,B),lt(B,C) ==> lt(A,C).    % transitive
le(A,B), ne(A,B) <=> lt(A,B).
ge(A,B) <=> le(B,A).
gt(A,B) <=> lt(B,A).
%% Rules for addition.
add(X,Y,Z) <=> Y=0 | X=Z.       % zero
add(X,Y,Z) <=> X=Z | Y=0.       % zero
add(X,Y,Z) ==> add(Y,X,Z).      % commutative
add(X,Y,Z), add(Z,U,W) ==> add(Y,U,R),add(X,R,W).     % associative
add(X1,Y,Z1), add(X2,Y,Z2) ==> le(Z1,Z2) | le(X1,X2). %
add(X1,Y,Z1), add(X2,Y,Z2) ==> lt(X1,X2) | lt(Z1,Z2). %
add(X1,Y,Z1), add(X2,Y,Z2) ==> lt(Z1,Z2) | lt(X1,X2). %
add(X1,Y,Z1), add(X2,Y,Z2) ==> gt(X1,X2) | gt(Z1,Z2). %
add(X1,Y,Z1), add(X2,Y,Z2) ==> gt(Z1,Z2) | gt(X1,X2). %
%% Rules for subtraction
minus(X,Y,Z) <=> add(Y,Z,X).    % normalise
%% Rules for minimum operation.
min2(A,B,C) <=> gt(A,B) | C = B.
min2(A,B,C) <=> le(A,B) | A = C.
min2(A,B,C) ==> min2(B,A,C).      % commutative
min2(A,B,C), min2(C,D,E) ==> min2(B,D,F), min2(A,F,E). % associative
%% Rules for maximum operation.
max2(A,B,B) <=> min2(A,B,A).
```