Enhanced Parallelization via Constraints

Wei-Ngan CHIN
National University of Singapore

Zhenjiang HU
University of Tokyo

Masato TAKEICHI
University of Tokyo

Akihiko TAKANO
Hitachi Advanced Research Laboratory

Abstract

Systematic parallelization of sequential programs remains a major challenge in parallel computing. Traditional approaches using program schemes are somewhat narrow in scope, as the properties which enable parallelism are difficult to capture via ad-hoc schemes. We propose a more systematic approach to parallelization based on the notion of preserving the *context* of recursive sub-terms. This approach can be used to derive a class of divide-and-conquer programs. To enhance the methodology further, we advocate the use of *required constraints* to widen the class of programs that could be handled. A unique feature of our approach is that it supports both *reusability* and *efficiency*. In particular, both general and specialised contraints are gathered to make this marriage possible.
Keywords: *Parallelization, Context Preservation, Conditional & Tupled Recurrences, Hierarchy of 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 simplify the program design and debugging processes, and allows better portability to be achieved.

We shall support this call by proposing an enhanced form of parallelization to enable the derivation of parallel programs from sequential code. A particularly important form of parallelization involves the exploitation of associativity/distributivity present in sequential codes to allow transformation into divide-and-conquer style algorithms. Consider the following two generic list-type functions, parameterised by function-type meta-variables: $\mathcal{A}, \mathcal{B}$, $\oplus$ and $\otimes$.

$$\begin{array}{ll} \mathit{homo}([x]) & = \mathcal{A}(x) \\ \mathit{homo}(x{:}xs) & = \mathcal{A}(x) \otimes \mathit{homo}(xs) \\ \mathit{lrec}([x]) & = \mathcal{B}(x) \\ \mathit{lrec}(x{:}xs) & = \mathcal{B}(x) \oplus (\mathit{lrec}(xs) \otimes \mathcal{A}(x)) \end{array}$$

Both functions are expressed sequentially through singleton *[x]* as base case, and list constructor *(x:xs)* as its recursive case. (Note that the functions are expressed using ML-like pattern-matching equations.) In both definitions, a single self-recursive call is embedded within one or more outer operators, such as $\oplus$ and $\otimes$. By exploiting suitable properties on these outer operators, we could derive divide-and-conquer style algorithms for their sequential functions.

For example, if $\otimes$ is associative we can guarantee the following parallel equation for *homo*.

$$\mathit{homo}(xr +\!\!+ xs) \quad = \mathit{homo}(xr) \otimes \mathit{homo}(xs)$$

This class of functions is commonly known as *join-list homomorphism* [Bir87, Ski90] (herewith abbreviated as *homomorphism*). Note that $+\!\!+$ denotes list-splitting when used in the LHS of an equation, but denotes list-catenation when used in the RHS.

As for *lrec*, it has two outer operators $\oplus$ and $\otimes$. To provide for its parallel equations below, $\oplus$ and $\otimes$ should be associative, with $\otimes$ distributing backwards over $\oplus$ via $(a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c)$. This class of functions is typically known as *first-order recurrence*[1] [Sto75].

[1] Though recurrences are typically used to describe parameter-less equations over arrays, we have generalized them to refer to recursive functions with one or more parameters in this paper.

$lrec(cs,x)$ $= let\ (u,\_)=ptup(cs,x)\ in\ u$
$ptup([x])$ $= (\mathcal{B}(x),\mathcal{A}(x))$
$ptup(xr{+\!\!+}xs)$ $= comb2(ptup(xr),ptup(xs))$
$where\ comb2((p_1,u_1),(p_2,u_2)) = (p_1 \oplus (p_2 \otimes u_1), u_2 \otimes u_1)$

Note the use of a parallel tupled function *ptup* which essentially computes $ptup(xs)=(lrec(xs),homo(xs))$. In this parallel definition, *lrec* uses the results of *homo* as auxiliary calls, even though this is not specified in the original sequential definition. Such dependance on unspecified auxiliary functions complicates the parallelization process, since inventive insights may be required. Even more problematic is the fact that program schemes, such as homomorphism and first-order (or even higher-order) recurrences, are still inadequate for directly capturing certain classes of sequential programs that could be parallelized.

Particularly problematic are functions whose outer operators involve conditional expressions and/or tuple constructs. A simple example is the single bracketing problem (abbreviated as *sbp*) to check if brackets in a given string are properly paired. Defined below is such a function which returns *0* for *properly paired brackets*, a negative result for *too many close brackets*, and *1* for *too many open brackets*. For example, $sbp(\text{“a+(b*(c/d))”}) \Rightarrow 0$, while $sbp(\text{“a*(((b+c)”}) \Rightarrow 1$ and $sbp(\text{“a*(b+c))+(c))”}) \Rightarrow -2$.

$sbp([x])$ $= conv(x)$
$sbp(x{:}xs)$ $= if\ (sbp(xs) \leq 0)\ then\ sbp(xs)+conv(x)\ else\ 1$
$conv(x)$ $= if\ x==')'\ then\ -1\ else\ if\ x=='('\ then\ 1\ else\ 0$

Notice that there are two recursive calls to *sbp(xs)* in the RHS of the recursive equation. The first call lies in the conditional's test, while the second call has both + and *if* as its outer operators. To make the outer operators of these calls explicit, we can rewrite the recursive equation as:

$sbp(x{:}xs)$ $= (\lambda\ (\alpha_1,\alpha_2,\alpha_3,r).\ if\ (r \leq \alpha_1)\ then\ r+\alpha_2\ else\ \alpha_3)\ (0,\ conv(x),\ 1,\ sbp(xs))$

The lambda expression explicitly captures the outer operators of the recursive *sbp(xs)*. This definition lies outside of both join-list homomorphism[2], and first-order recurrence. To support its parallelization, we require the associativity of + and also the distributive and flattening properties of the *if* conditional. Sadly, these properties are still not enough for parallelization, and we also require an extra constraint, namely $\alpha_3 \geq \alpha_1 + \alpha_2$, on arguments $\{\alpha_i\}_{i \in 1..3}$ of the outer lambda operator. We refer to this as a *required (invariant) constraint*, for parallelization, and shall elaborate why it is needed in Sec 4.

This paper is about an enhanced methodology for deriving parallel programs directly from sequential counterparts. We focus on an important extension which relies on *required constraints* for successful parallelization. Such constraints are related to *loop invariants* which are typically associated with the optimization and reasoning of iterative loops. Like loop invariants, our constraints must be kept invariant across successive recursive calls. To the best of our knowledge, this is the first time that invariants are being exploited for parallelization. The main contributions of our paper are:

- We propose an *enhanced* approach to parallelization which could handle a wide class of functions, not restricted to conventional program schemes. A novel aspect of our approach is that it can be augmented with *required constraints* to achieve more parallelization. (Sec 2 & 4)
- We provide a set of *systematic* techniques to handle recursive functions with conditional and tupled constructs. Such functions are harder to parallelize, but this difficulty can be overcome by designing a set of suitable normalisation rules. (Sec 3)
- We advocate a *modular* approach to parallelization which supports both reusability and efficiency. Generic schemes/constraints are developed where possible for wider reusability, but specialised schemes/constraints are also explored for better coding. (Sec 5)

We shall consider a non-trivial problem, called maximum segment product, whose parallelization is quite complex but can nevertheless be carried out systematically with our approach. (Appendix A)

2 A Theorem for Parallelization

Most optimisation techniques, such as partial evaluation[JGS93] and deforestation[Wad88], are based on the notion of program specialisation, whose goal is to *specialise* programs to their specific contexts of use.

[2] In [Col95], this function is referred to as a *near-homomorphism* since there exists a more general tupled function (with *sbp* as a component) that is a homomorphism, but not *sbp* itself. In fact, this applies to the *lrec* function as well.

However, parallelization is a different beast as the goal is to obtain more *general* parallel equations from their sequential counterparts. Due to this difference, we have developed a new method for parallelization based on generalizing from sequential examples [CTH98].

Consider a version of prefix computation, *ascan*, with an accumulating parameter. Given an input list $xs=[x_1, x_2, \ldots, x_n]$, *ascan(xs,w)* is defined to compute a corresponding list of prefix operations, $[w \otimes x_1,\ w \otimes x_1 \otimes x_2, \ldots,\ w \otimes x_1 \otimes x_2 \otimes \cdots \otimes x_n]$.

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

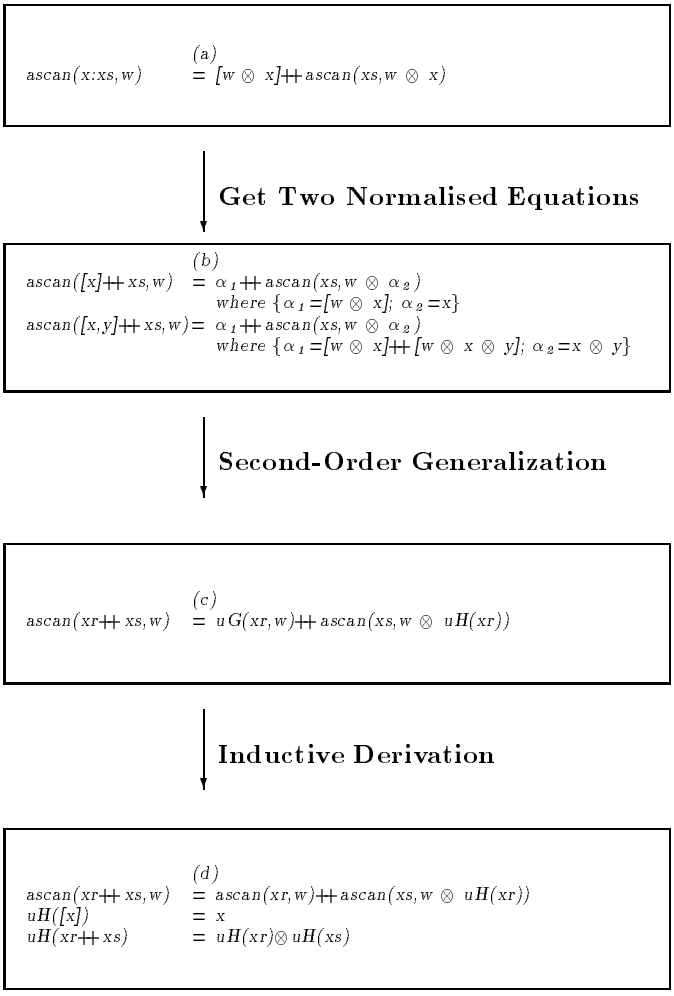

Figure 1: Steps for Example-Based Parallelization

We highlight *ascan* because its versatility for parallel computation is well-known[Ble89, BCH+93]. Our method could automatically derive its parallel implementation. Its main steps are illustrated in Figure 1. The key idea is to obtain two normalised sequential equations (see Fig. 1(b)) for *ascan*, whose contexts for the recursive call and accumulative argument(s) are identical. In particular, the context for the recursive call is $(\hat{\lambda}\langle\underline{\bullet}\rangle.\alpha_1 +\!\!+ \underline{\bullet})$, and that for the accumulative argument is $(\hat{\lambda}\langle\underline{\bullet}\rangle.\underline{\bullet} \otimes \alpha_2)$. These contexts are identical across both sequential equations of *ascan*. We refer to such contexts as *recurring contexts* (or *R-contexts*), which contain R-holes (denoted by $\underline{\bullet}$) for the positions of recursive call (or accumulative argument). Such R-contexts are *recurring* in nature since they allow their body to be replicated at the R-holes, when the recursive call is unfolded.

The presence of two similar sequential equations permits a second-order generalization process to be made where unknown functions, such as *uG* and *uH* in Fig. 1(c), are introduced. Inductive derivation

can then provide definitions for the unknown functions. These unknown functions may be equivalent[3] to existing functions, and should be *replaced* to reduce unnecessary code, if so. In our example, uG is found to have the same definition as *ascan*, while uH is given a new definition, as shown in Fig. 1(d).

To formalise this process, we have identified a powerful property which guarantees parallelization, based on the notion of preserving R-contexts across replication.

Definition 1: *Context Preservation Property*
A R-context E is said to be *preserved modulo replication* if the following holds:

$$\exists Es.\ (E \Rightarrow_T Es\langle t_i\rangle_{i\in N}) \wedge (Es\langle \alpha_i\rangle_{i\in N} \circ Es\langle \beta_i\rangle_{i\in N} \Rightarrow_T Es\langle \Omega_i\rangle_{i\in N})$$

where α_i and β_i are variables, Ω_i denote subterms not containing (includes referencing via local variables) the R-hole, and $\circ$ denotes composition.

Note that $(E \Rightarrow_T Es\langle t_i\rangle_{i\in N})$ denotes a normalisation process where the *depth* and *number of occurrences* of each R-hole is minimised[CTH98]. We refer to E as the *original R-context*, and $Es\langle t_i\rangle_{i\in N}$ as the *initial (normalised) R-context*. Also, $Es\langle \alpha_i\rangle_{i\in N}$ is referred as the *skeletal R-context* of $Es\langle t_i\rangle_{i\in N}$ in which (maximal) sub-terms, i.e. $\{t_i\}_{i\in N}$, not containing the R-hole $\underline{\bullet}$, are replaced by distinct variables, $\{\alpha_i\}_{i\in N}$. It is actually this skeletal R-context which is being preserved despite replication via $(Es\langle \alpha_i\rangle_{i\in N} \circ Es\langle \beta_i\rangle_{i\in N}) \Rightarrow_T Es\langle \Omega_i\rangle_{i\in N}$. The presence of this property is sufficient to ensure parallelization. Its presence permits both second-order generalization and the inductive derivation of unknown functions.

Example 1: Using the *ascan* example, the *original R-context* of its recursive call is $E = \hat{\lambda}\langle\underline{\bullet}\rangle.[w \otimes x] +\!\!+ \underline{\bullet}$. Normalisation could not reduce the depth of $\underline{\bullet}$, and thus we return the same expression for the *initial R-context*, namely $Es\langle t_i\rangle_{i\in N} = \hat{\lambda}\langle\underline{\bullet}\rangle.[w \otimes x] +\!\!+ \underline{\bullet}$. The subterm $[w \otimes x]$ can be replaced by a distinct variable, say α_1, giving the corresponding *skeletal R-context* as $Es\langle \alpha_i\rangle_{i\in N} = \hat{\lambda}\langle\underline{\bullet}\rangle.\alpha_1 +\!\!+ \underline{\bullet}$.

Based on this property, the parallelization process can be succinctly formalised by the following theorem, originally proposed and proven in [CTH98].

Theorem 1: *Context Preservation & Parallelization*
Consider a linear self-recursive function where $\lfloor t_j \rfloor_{j=1}^n$ is an abbreviation for $t_1, \ldots, t_n$.

$$\begin{array}{ll} f(Nil, \lfloor v_j \rfloor_{j=1}^n) & = E_z \\ f(x : xs, \lfloor v_j \rfloor_{j=1}^n) & = Er\langle f(xs, \lfloor Dr_j\langle v_j\rangle \rfloor_{j=1}^n)\rangle \end{array}$$

This function can be successfully parallelized if the context preservation property holds for:

- The R-context of the recursive call, i.e. Er.
- Each R-context of the accumulative parameters, i.e. $\{Dr_j\}_{j\in 1..n}$.

Proof : Proof given in [CTH98]. □

In the *linear self-recursive* class of functions, each recursive equation is only allowed a single self-recursive call, but can have an arbitrary number of accumulative arguments, namely $\lfloor v_j \rfloor_{j=1}^n$, which are independent of each other. (Note that mutual-dependent parameters can always be converted to this form by tupling the separate parameters together.) As we shall see, many sequential programs fall under this class or can be easily converted to it.

To illustrate our theorem, consider the *ascan* example. The skeletal R-context of the recursive call is $\hat{\lambda}\langle\underline{\bullet}\rangle.\alpha_1 +\!\!+ \underline{\bullet}$, while that for the accumulative argument is $\hat{\lambda}\langle\underline{\bullet}\rangle.\underline{\bullet} \otimes \alpha_1$. Both these two skeletal R-contexts can be preserved modulo replication, as illustrated below.

$$\begin{array}{ll} (\hat{\lambda}\langle\underline{\bullet}\rangle.\alpha_1 +\!\!+ \underline{\bullet}) \circ (\hat{\lambda}\langle\underline{\bullet}\rangle.\beta_1 +\!\!+ \underline{\bullet}) \Rightarrow_T & \{ \text{ definition of } \circ \ \} \\ \qquad \hat{\lambda}\langle\underline{\bullet}\rangle.\alpha_1 +\!\!+ (\beta_1 +\!\!+ \underline{\bullet}) & \\ \Rightarrow_T \qquad \{ \text{ associativity of } +\!\!+ \ \} & \\ \qquad \hat{\lambda}\langle\underline{\bullet}\rangle.(\alpha_1 +\!\!+ \beta_1) +\!\!+ \underline{\bullet} & \end{array}$$

[3] This is a syntactic check to determine if two functions are equivalent. Two functions are equivalent if their definitions unify with each other.

$$\Rightarrow_T \quad \{ \text{ form skeletal R-context } \}$$
$$(\hat{\lambda}\langle\underline{\bullet}\rangle.\Omega_1 +\!\!+ \underline{\bullet}) \textit{ where } \Omega_1 = (\alpha_1 +\!\!+ \beta_1)$$

$$(\hat{\lambda}\langle\underline{\bullet}\rangle.\underline{\bullet}\otimes\alpha_1) \circ (\hat{\lambda}\langle\underline{\bullet}\rangle.\underline{\bullet}\otimes\beta_1) \Rightarrow_T \quad \{ \text{ definition of } \circ \}$$
$$\hat{\lambda}\langle\underline{\bullet}\rangle.(\underline{\bullet}\otimes\beta_1)\otimes\alpha_1$$
$$\Rightarrow_T \quad \{ \text{ associativity of } \otimes \}$$
$$\hat{\lambda}\langle\underline{\bullet}\rangle.\underline{\bullet}\otimes(\beta_1\otimes\alpha_1)$$
$$\Rightarrow_T \quad \{ \text{ form skeletal R-context } \}$$
$$\hat{\lambda}\langle\underline{\bullet}\rangle.\underline{\bullet}\otimes\Omega_1 \textit{ where } \Omega_1 = (\beta_1\otimes\alpha_1)$$

Since context preservation holds, our theorem guarantees its second-order generalization and inductive derivation to yield the parallel equations shown in Fig 1(d). These parallel equations presently contain redundant calls which can be eliminated by the automated tupling method of [Chi93, HITT97]. This step is essential for obtaining work-efficient parallel programs. In the case of *ascan*, we can obtain:

$$\begin{array}{ll} \textit{ascan(xs,w)} & = \textit{let } (u,\_)=\textit{astup(xs,w) in } u \\ \textit{astup([x],w)} & = (\textit{[w}\otimes\textit{x],x)} \\ \textit{astup(xr}+\!\!+\textit{xs,w)} & = \textit{let } \{\ (u,v)=\textit{astup(xr,w); } (a,b)=\textit{astup(xs,w}\otimes\textit{v)}\ \} \textit{ in } (u +\!\!+ a,\ v \otimes b) \end{array}$$

A remarkable aspect of our parallelization method is that it is extremely general. In particular, it is equipped to handle more complex recurrences, including those with conditional and tupled constructs. In the next section, we look at a core set of normalisation rules which facilitate parallelization.

3 Normalisation Rules

To facilitate parallelization, there is a need for a set of normalisation rules to transform our R-contexts into some canonical form. The main principle behind our normalisation strategy is to *preserve the location and path of each R-hole* in order to facilitate context preservation. To achieve this, we have proposed two heuristics to guide our normalisation, namely:

Definition 2: *Heuristics/Guidelines for Normalisation Rules*
Consider a R-context with one or more R-holes. Our normalisation shall attempt to:

- Minimise the *depth*[a] of the *R-holes* or their *proxies*.
- Minimise the *number of occurrences* of the *R-holes* or their *proxies*.

A *proxy* is a local variable which denotes either a R-hole or its component. □

[a] 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.

When a R-context is replicated, its R-holes (or proxies) are likely to be replicated at greater depths, and in larger numbers. The above two heuristics are aimed at transforming the replicated R-contexts back to the original form.

We shall look at a core set of transformation rules that are guided by the above heuristics. The rules are given in Figure 2. They are not meant to be exhaustive. To structure them, we organise them into five categories, namely:

- Operator Replacement
- Flattening
- Duplicate Elimination
- Distribution
- Housekeeping

Operator Replacement:
(1a) $e_1 \otimes e_2 \Rightarrow \otimes[e_1, e_2]$ *IF* $\otimes$ *is associative &* $\wedge((e_1^\bullet) \vee (e_2^\bullet))$
(1b) $e_1 \oplus e_2 \Rightarrow \oplus\{e_1, e_2\}$ *IF* $\oplus$ *is associative & commutative* $\wedge((e_1^\bullet) \vee (e_2^\bullet))$
(1c) if b *then* e_1 *else* $e_2 \Rightarrow$ *if* $\{\ b \rightarrow e_1;\ \neg\ b \rightarrow e_2\}$ *IF* $(b^\bullet) \vee (e_1^\bullet) \vee (e_2^\bullet)$
(1d) $e_1 \ominus e_2 \Rightarrow \oplus[e_1, \neg_\oplus e_2]$ *IF* $(a \ominus b) \ominus c = a \ominus (b \oplus c) \wedge \oplus$ *is associative*

Flattening :
(2a) $\otimes[t\vec{1}, \otimes[t\vec{2}], t\vec{3}] \Rightarrow \otimes[t\vec{1}, t\vec{2}, t\vec{3}]$ *IF* $\exists t \in t\vec{2}.\ t^\bullet$
(2b) $\oplus\{t\vec{1}, \oplus\{t\vec{2}\}, t\vec{3}\} \Rightarrow \oplus\{t\vec{1}, t\vec{2}, t\vec{3}\}$ *IF* $\exists t \in t\vec{2}.\ t^\bullet$
(2c) if $B \cup \{bi \rightarrow$ *if* $\{bi_j \rightarrow ei_j\}_{j \in M}\} \Rightarrow$ *if* $B \cup \{bi \wedge bi_j \rightarrow ei_j\}_{j \in M}$ *IF* $\exists t \in \{bi_j, ei_j\}_{j \in M}.\ t^\bullet$
(2d) if $B \cup \{$*if* $\{bi_j \rightarrow ei_j\}_{j \in M} \rightarrow e_i\} \Rightarrow$ *if* $B \cup \{\bigvee_{j \in M}(bi_j \wedge ei_j) \rightarrow e_i\}$ *IF* $\exists t \in \{bi_j, ei_j\}_{j \in M}.\ t^\bullet$

Distributivity:
(3a) $\oplus[\vec{a}, \otimes[b_1, \ldots, b_n]] \Rightarrow \bar{\otimes}[\bar{\oplus}[\vec{a}, b_1], .., \bar{\oplus}[\vec{a}, b_n]]$ *IF* $\forall a \in \vec{a}.\ \neg a^\bullet \ \wedge\ \exists i \in 1..n.\ b_i^\bullet$
(3b) $\otimes[\oplus[\vec{a}, b_1], .., \oplus[\vec{a}, b_n]] \Rightarrow \bar{\oplus}[\vec{a}, \bar{\otimes}[b_1, \ldots, b_n]]$ *IF* $\exists i \in 1..n.\ b_i^\bullet$
(3c) $\widehat{e^I}\langle$*if* $\{b_i \rightarrow e_i\}_{i \in N}\rangle \Rightarrow$ *if* $\{b_i \rightarrow \widehat{e^I}\langle e_i\rangle\}_{i \in N}$ *IF* $\exists i \in N.\ (b_i^\bullet) \vee (e_i^\bullet)$

Duplicate Elimination:
(4a) $\bar{\otimes}[\bar{\oplus}[\vec{a}, b_1], .., \bar{\oplus}[\vec{a}, b_n]] \Rightarrow \oplus[\vec{a}, \otimes[b_1, \ldots, b_n]]$ *IF* $\exists a \in \vec{a}.\ a^\bullet$
(4b) if $\{b_i \rightarrow \widehat{e^I}\langle e_i\rangle\}_{i \in N} \Rightarrow \widehat{e^I}\langle$*if* $\{b_i \rightarrow e_i\}_{i \in N}\rangle$ *IF* $\forall i \in N.\ (\neg e_i^\bullet) \wedge (e^{I\bullet})$
(4c) $(a \leq e_1) \wedge (a \leq e_2) \Rightarrow (a \leq min2[e_1, e_2])$ *IF* $(a^\bullet)$
(4d) $(a \leq e_1) \vee (a \leq e_2) \Rightarrow (a \leq max2[e_1, e_2])$ *IF* $(a^\bullet)$

Housekeeping:
(5a) if $\{b_i \rightarrow e\}_{i \in N} \Rightarrow e$
(5b) if $B \cup \{True \rightarrow e\} \Rightarrow e$
(5c) if $B \cup \{False \rightarrow e\} \Rightarrow$ *if* B
(5d) if $\{b_i \rightarrow e_i\}_{i \in N \cup M} \Rightarrow$ *if* $\{b_i \rightarrow e_i\}_{i \in N} \cup \{\bigvee_{j \in M} b_j \rightarrow$ *if* $\{b_i \rightarrow e_i\}_{i \in M}\}$ *IF* $\forall i \in M.\ (\neg\ b_i^\bullet \ \wedge\ \neg\ e_i^\bullet)$

Figure 2: Five Categories of Normalisation Rules

The *operator replacement* rules are aimed at giving a canonical representation to operators with associative/commutative properties. Binary operators are converted to n-ary (prefix) versions via (1a), with set-notation for operators that commute via (1b). Certain operators without full associativity may be converted to corresponding operators with the associative property. Rule (1d) shows how to change a semi-associative $\ominus$ operator. A common requirement amongst these rules is that the operators should have at least one *recurring term (or R-term)*, as their arguments. *A R-term, denoted by* $t^\bullet$*, is an expression which contains at least one R-hole or its proxy.*

The next group of rules, *flattening*, is aimed at reducing the depths of R-terms, where possible. The rules are only applied if some inner R-terms, e.g. *t2*, could be lifted outwards.

A slightly more complex rule is *distributivity*. While associativity may be used to flatten a nested pair of identical operations, *distributivity* is needed to deal with nesting from a pair of distinct operators. The distributive laws, often do not increase nor decrease the depths of the the subterms, $[b_1, \ldots, b_n]$ (see 3a and 3b). However, it can facilitate the application of a subsequent flattening rule, under two scenarios:

(A) The targetted R-term has the form $b_i = \bar{\oplus}[t_1, \ldots, t_n]$. The distributive law can then propagate the outer $\bar{\oplus}$ operator next to the inner $\bar{\oplus}$ operator, before flattening is applied.

(B) The immediate outer context has the form $\bar{\otimes}[t1, .., \{\}, .., tn]$ where $\{\}$ denotes the current expression. The distributive law can then propagate the inner $\bar{\otimes}$ operator outwards, next to the outer $\bar{\otimes}$ operator, before flattening is applied.

Note that $\otimes$ and $\bar{\otimes}$ represent possibly distinct operators. Also, the distributive laws (3a) and (3b) are the converse of each other. To prevent looping, we shall apply distribution only if one of the above two scenarios is satisfied. Distributivity must also be alternated with flattening to monotonically decrease the depth of R-terms.

The rules for *duplicate elimination* are harder to formulate. The first rule (4a) is the converse of distribution, with $\vec{a}$ containing one or more R-terms whose multiple occurrences must be eliminated. The *if* rule (4b) deals with a conditional whose branches have identical R-term as its context. The outer *if*

is pushed into the non-recursive sub-terms. More specific rules for removing duplicate occurrences must also be added (e.g. 4c,4d).

Lastly, the *housekeeping* rules are provided for simplification, and for re-grouping similar R-terms, or non-recursive subterms together.

The normalisation rules shall now be illustrated with a segmented scan operation This operation is particularly suited for specifying functions over irregular data structures. Irregular data structures, such as sparse matrixes, are typically encoded using nested lists (e.g. $nl = [[a_1, a_2], [b_1], [c_1, c_2, c_3]]$). These nested structures can be flattened into more regular lists (e.g. $fl = [a_1, a_2, b_1, c_1, c_2, c_3]$) by augmenting with boolean-tags (e.g. $bl = [True, False, True, True, False, False]$) to mark the beginning of each sublist. Using such flattened structures, we can compute prefixes for each sublist with the following function.

$$\begin{array}{l} segscan([],[],w) \quad = [] \\ segscan(x{:}xs,b{:}bs,w) = \textit{if } b{==}True \textit{ then } [x]{+\!+}segscan(xs,bs,x) \\ \qquad\qquad\qquad\quad \textit{else } [w \otimes x]{+\!+}segscan(xs,bs,w \otimes x) \end{array}$$

For example, $segscan(fl,bl,w)$ would return: $[w{\otimes}a_1, w{\otimes}a_1{\otimes}a_2, b_1, c_1, c_1{\otimes}c_2, c_1{\otimes}c_2{\otimes}c_3]$. The *segscan* function currently contains two self-recursive calls. When we apply the normalisation rules, we would convert it into a linear self-recursive function, as shown below.

$$\begin{array}{l} segscan(x{:}xs,b{:}bs,w) = \quad \{ \text{ minimise dupl. recursive calls via (4b) } \} \\ \qquad\qquad\qquad\quad (\textit{if } b{==}True \textit{ then } [x] \textit{ else } [w \otimes x]){+\!+}segscan(xs,bs,\textit{if } b{==}True \textit{ then } x \textit{ else } w \otimes x) \end{array}$$

With this normalisation, we can proceed to extract out the contexts of the recursive call, namely $(\hat{\lambda}\langle\underline{\bullet}\rangle.\alpha_1 {+\!+} \underline{\bullet})$, and the accumulative argument, $\hat{\lambda}\langle\underline{\bullet}\rangle.\textit{if } \alpha_1 \textit{ then } \alpha_2 \textit{ else } \underline{\bullet} \otimes \alpha_3$. The first R-context has already been shown to satisfy the context preservation property. The second R-context can be checked for the *context preservation* property, as illustrated below.

$$\begin{array}{l} (\hat{\lambda}\langle\underline{\bullet}\rangle.\textit{if } \alpha_1 \textit{ then } \alpha_2 \textit{ else } \underline{\bullet} \otimes \alpha_3) \circ (\hat{\lambda}\langle\underline{\bullet}\rangle.\textit{if } \beta_1 \textit{ then } \beta_2 \textit{ else } \underline{\bullet} \otimes \beta_3) \\ \quad \Rightarrow_T \quad \{ \text{ definition of } \circ \ \} \\ \qquad (\hat{\lambda}\langle\underline{\bullet}\rangle.\textit{if } \alpha_1 \textit{ then } \alpha_2 \textit{ else } (\textit{if } \beta_1 \textit{ then } \beta_2 \textit{ else } \underline{\bullet} \otimes \beta_3) \otimes \alpha_3) \\ \quad \Rightarrow_T \quad \{ \text{ operator replacements (1a), (1c) } \} \\ \qquad (\hat{\lambda}\langle\underline{\bullet}\rangle.\textit{if } \{ \alpha_1 \to \alpha_2; \neg\alpha_1 \to \otimes[\textit{if } \{ \beta_1 \to \beta_2; \neg\beta_1 \to \otimes[\underline{\bullet},\beta_3]\},\alpha_3]) \\ \quad \Rightarrow_T \quad \{ \text{ distribute inner } \textit{if} \text{ outwards (3c) } \} \\ \qquad (\hat{\lambda}\langle\underline{\bullet}\rangle.\textit{if } \{ \alpha_1 \to \alpha_2; \neg\alpha_1 \to \textit{if } \{ \beta_1 \to \otimes[\beta_2,\alpha_3]; \neg\beta_1 \to \otimes[\otimes[\underline{\bullet},\beta_3],\alpha_3]\}) \\ \quad \Rightarrow_T \quad \{ \text{ flatten } \textit{if} \text{ (2c) and } \otimes \text{ (2a) } \} \\ \qquad (\hat{\lambda}\langle\underline{\bullet}\rangle.\textit{if } \{ \alpha_1 \to \alpha_2; \neg\alpha_1 \wedge \beta_1 \to \otimes[\beta_2,\alpha_3]; \\ \qquad\qquad \neg\alpha_1 \wedge \neg\beta_1 \to \otimes[\underline{\bullet},\beta_3,\alpha_3]\}) \\ \quad \Rightarrow_T \quad \{ \text{ regroup if via (5d) } \} \\ \qquad (\hat{\lambda}\langle\underline{\bullet}\rangle.\textit{if } \{ \alpha_1 \vee \beta_1 \to \textit{if } \{ \alpha_1 \to \alpha_2; \neg\alpha_1 \to \otimes[\beta_2,\alpha_3]\}; \neg(\alpha_1 \vee \beta_1) \to \otimes[\underline{\bullet},\otimes[\beta_3,\alpha_3]]\}) \\ \quad \Rightarrow_T \quad \{ \text{ restore original operators using the converse of (1a),(1c) } \} \\ \qquad (\hat{\lambda}\langle\underline{\bullet}\rangle.\textit{if } \alpha_1 \vee \beta_1 \textit{ then } (\textit{if } \alpha_1 \textit{ then } \alpha_2 \textit{ else } \beta_2 \otimes \alpha_3) \textit{ else } \underline{\bullet} \otimes (\beta_3 \otimes \alpha_3) \ ) \ ) \\ \quad \Rightarrow_T \quad \{ \text{ form a skeletal R-context } \} \\ \qquad (\hat{\lambda}\langle\underline{\bullet}\rangle.\textit{if } \Omega_1 \textit{ then } \Omega_2 \textit{ else } (\underline{\bullet} \otimes \Omega_3)) \textit{ where } \{ \Omega_1{=}\alpha_1 \vee \beta_1; \ \Omega_3{=}\beta_3 \otimes \alpha_3; \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \Omega_2{=}\textit{if } \alpha_1 \textit{ then } \alpha_2 \textit{ else } \beta_2 \otimes \alpha_3 \ \} \end{array}$$

Since context preservations hold, we know the following second-order generalization is valid.

$$segscan(xr{+\!+}xs,br{+\!+}bs,w) = uG(xr,br,w){+\!+}segscan(xs,bs,\textit{if } uH(xr,br) \textit{ then } uJ(xr,br) \textit{ else } w \otimes uK(xr,br))$$

Inductive derivation then yields definitions for the unknown functions, resulting in:

$$\begin{array}{ll} segscan(xr{+\!+}xs,br{+\!+}bs,w) & = segscan(xr,br,w){+\!+}segscan(xs,bs,\textit{if } uH(xr) \textit{ then } uJ(xr,br,w) \textit{ else } w \otimes uK(xr)) \\ uH([b]) & = (b{==}True) \\ uH(br{+\!+}bs) & = uH(br) \wedge uH(bs) \\ uK([x]) & = x \\ uK(xr{+\!+}xs) & = uK(xr) \otimes uK(xs) \\ uJ([x],[b],w) & = \textit{if } (b{==}True) \textit{ then } x \textit{ else } w \otimes x \\ uJ(xr{+\!+}xs,br{+\!+}bs,w) & = \textit{if } uH(br) \textit{ then } uJ(xr,br,w) \textit{ else } uJ(xr,br,w) \otimes uK(xs)) \end{array}$$

4 Constraint-Enhanced Parallelization

While our Parallelization Theorem may be quite general, there are still important classes of sequential programs where it fails. The main reason is its weakness at manipulating R-holes (particularly a powerful

way to reduce the number of R-holes) to ensure context preservation. To remedy this situation, we shall add suitable constraints to strengthen the opportunites for context preservation. It turns out that some context preservations for parallelization could only be achieved *under a certain (invariant) constraint.*

A simple example is the *sbp* function given in Sec 1. The context for the recursive call is actually: $(\hat{\lambda}\langle\underline{\bullet}\rangle.\mathit{if}\ \underline{\bullet}{\leq}\alpha_1\ \mathit{then}\ \underline{\bullet}{+}\beta_2\ \mathit{else}\ \beta_3)\ \mathit{where}\ \{\beta_1{=}0;\ \beta_2{=}\mathit{conv}(x);\ \beta_3{=}1\}$

Unlike the R-context of segmented scan, the skeletal version of this R-context is harder to preserve since its R-hole appears twice in its conditional's test and one of its branches. We may attempt to normalise its replicated R-contexts, as follows:

$$
\begin{array}{l}
(\hat{\lambda}\langle\underline{\bullet}\rangle.\mathit{if}\ \underline{\bullet}{\leq}\alpha_1\ \mathit{then}\ \underline{\bullet}{+}\alpha_2\ \mathit{else}\ \alpha_3)\circ(\hat{\lambda}\langle\underline{\bullet}\rangle.\mathit{if}\ \underline{\bullet}{\leq}\beta_1\ \mathit{then}\ \underline{\bullet}{+}\beta_2\ \mathit{else}\ \beta_3)\\
\quad\Rightarrow_T\quad \{\ \text{definition of}\ \circ\ \}\\
\qquad(\hat{\lambda}\langle\underline{\bullet}\rangle.\mathit{if}\ (\mathit{if}\ \underline{\bullet}{\leq}\beta_1\ \mathit{then}\ \underline{\bullet}{+}\beta_2\ \mathit{else}\ \beta_3){\leq}\alpha_1\ \mathit{then}\ (\mathit{if}\ \underline{\bullet}{\leq}\beta_1\ \mathit{then}\ \underline{\bullet}{+}\beta_2\ \mathit{else}\ \beta_3){+}\alpha_2\ \mathit{else}\ \alpha_3)\\
\quad\Rightarrow_T\quad \{\ \text{float out}\ (\underline{\bullet}{\leq}\beta_1)\ \&\ \text{simplify}\ \}\\
\qquad\hat{\lambda}\langle\underline{\bullet}\rangle.\mathit{if}\ (\underline{\bullet}{\leq}\beta_1)\ \mathit{then}\ \mathit{if}\ \underline{\bullet}{+}\beta_2{\leq}\alpha_1\ \mathit{then}\ (\underline{\bullet}{+}\beta_2){+}\alpha_2\ \mathit{else}\ \alpha_3\\
\qquad\quad\mathit{else}\ \mathit{if}\ \beta_3{\leq}\alpha_1\ \mathit{then}\ \beta_3{+}\alpha_2\ \mathit{else}\ \alpha_3\\
\quad\Rightarrow_T\quad \{\ \text{flatten nested}\ \mathit{if}\ \}\\
\qquad\hat{\lambda}\langle\underline{\bullet}\rangle.\mathit{if}\ \{\ (\underline{\bullet}{\leq}\beta_1)\wedge(\underline{\bullet}{+}\beta_2{\leq}\alpha_1)\ \ \to(\underline{\bullet}{+}\beta_2){+}\alpha_2\ ;\\
\qquad\qquad(\underline{\bullet}{\leq}\beta_1)\wedge\neg(\underline{\bullet}{+}\beta_2{\leq}\alpha_1)\ \to\alpha_3\ ;\\
\qquad\qquad\neg(\underline{\bullet}{\leq}\beta_1)\wedge(\beta_3{\leq}\alpha_1)\ \ \to\beta_3{+}\alpha_2\ ;\\
\qquad\qquad\neg(\underline{\bullet}{\leq}\beta_1)\wedge\neg(\beta_3{\leq}\alpha_1)\ \to\alpha_3\ \}\\
\quad\Rightarrow_T\quad \{\ \text{transitivity of}\ \leq\ \&\ \text{re-group}\ \mathit{if}\ \}\\
\qquad\hat{\lambda}\langle\underline{\bullet}\rangle.\mathit{if}\ (\underline{\bullet}{\leq}\mathit{min2}(\beta_1,\alpha_1{-}\beta_2)\ \mathit{then}\ \underline{\bullet}{+}(\beta_2{+}\alpha_2)\\
\qquad\quad\mathit{else}\ \mathit{if}\ \{\ (\underline{\bullet}{\leq}\beta_1)\wedge\neg(\underline{\bullet}{+}\beta_2{\leq}\alpha_1)\to\alpha_3\ ;\\
\qquad\qquad\neg(\underline{\bullet}{\leq}\beta_1)\wedge(\beta_3{\leq}\alpha_1)\ \ \to\beta_3{+}\alpha_2\ ;\\
\qquad\qquad\neg(\underline{\bullet}{\leq}\beta_1)\wedge\neg(\beta_3{\leq}\alpha_1)\ \to\alpha_3\ \}\\
\quad\Rightarrow_T\quad \{\ \text{float non-recursive test}\ (\beta_3{\leq}\alpha_1)\ \text{and simplify}\ \ \}\\
\qquad\hat{\lambda}\langle\underline{\bullet}\rangle.\mathit{if}\ (\underline{\bullet}{\leq}\mathit{min2}(\beta_1,\alpha_1{-}\beta_2)\ \mathit{then}\ \underline{\bullet}{+}(\beta_2{+}\alpha_2)\\
\qquad\quad\mathit{else}\ \mathit{if}\ (\beta_3{\leq}\alpha_1)\ \mathit{then}\ (\mathit{if}\ \{\ (\underline{\bullet}{\leq}\beta_1)\wedge\neg(\underline{\bullet}{+}\beta_2{\leq}\alpha_1)\to\alpha_3\ ;\\
\qquad\qquad\qquad\qquad\neg(\underline{\bullet}{\leq}\beta_1)\ \ \to\beta_3{+}\alpha_2\ \})\\
\qquad\quad\mathit{else}\ \alpha_3
\end{array}
$$

Unfortunately, we are still unsuccessful as there are five occurrences of $\underline{\bullet}$, instead of two in the initial R-context. To allow context preservation to succeed, we need to simplify/normalise the following conditional by eliminating one of its two branches.

$$
\begin{array}{l}
\mathit{if}\ \{\quad(\underline{\bullet}{\leq}\beta_1)\wedge\neg(\underline{\bullet}{+}\beta_2{\leq}\alpha_1)\ \ \to\alpha_3\ ;\\
\qquad\qquad\neg(\underline{\bullet}{\leq}\beta_1)\ \ \to\beta_3{+}\alpha_2\ \}
\end{array}
\tag{1}
$$

Doing this allows us to remove three extraneous occurrences of $\underline{\bullet}$. We first normalise the predicate conditions associated with the two branches.

In the first branch, we have:

$$
\begin{array}{l}
(\beta_3{\leq}\alpha_1)\wedge(\underline{\bullet}{\leq}\beta_1)\wedge\neg(\underline{\bullet}{+}\beta_2{\leq}\alpha_1)\\
\qquad\vdash(\beta_3{\leq}\alpha_1)\wedge(\underline{\bullet}{\leq}\beta_1)\wedge(\underline{\bullet}{+}\beta_2{>}\alpha_1)\\
\qquad\vdash(\beta_3{\leq}\alpha_1)\wedge(\beta_1{>}\alpha_1{-}\beta_2)\\
\qquad\vdash(\beta_3{<}\beta_1{+}\beta_2)
\end{array}
$$

In the second branch, we have:

$$
\begin{array}{l}
\neg(\beta_3{\leq}\alpha_1)\wedge\neg(\underline{\bullet}{\leq}\beta_1)\\
\qquad\vdash(\alpha_1{<}\beta_3)\wedge(\beta_1{<}\underline{\bullet})
\end{array}
$$

The simplification carried out is basically a normalisation via SUP-INF procedure [Ble75] which attempts to remove variables, $\underline{\bullet}$ and α_1, leaving behind predicates exclusively in terms of $\{\beta_i\}_{i\in 1..3}$. Only the predicate condition for the first branch can be successfully simplified, but not the second branch. This means that the first branch can be eliminated by supplying a negated test, namely $\neg(\beta_3{<}\beta_1{+}\beta_2)$, as the *required constraint*. Adding this constraint allows the conditional of (1) to simplify to $\beta_3{+}\alpha_2$.

Note that the required constraint does not come from 'thin air' but is instead guided by the need for context preservation. For the R-context of *sbp*, we now have:

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

Note that *(e* **st** *p)* denotes an expression e under constraint p. As the constraint is assumed to be valid for one of the two R-contexts used, it shall be proven to be valid for all R-contexts encountered by the recursive parallel program. In particular, it must be valid for the initial R-context, and each R-context that is yielded by context preservation. This requirement will help ensure that all R-contexts have the required constraint. It can be expressed by the following constraint-based context preservation.

Definition 3: *Constraint-Based Context Preservation*
A R-context E is said to be *context preserved modulo replication* under *required constraint* P, if the following holds:

$$\exists Es.\,(E \Rightarrow_T Es\langle t_i\rangle_{i\in N}) \wedge P(t_i)_{i\in N} \wedge$$
$$(Es\langle \alpha_i\rangle_{i\in N} \circ Es\langle \beta_i\rangle_{i\in N})\ \mathbf{st}\ (P(\alpha_i)_{i\in N}\ \wedge\ P(\beta_i)_{i\in N}) \Rightarrow_T (Es\langle \Omega_i\rangle_{i\in N}\ \mathbf{st}\ P(\Omega_i)_{i\in N})$$

Notice that the P constraint must be shown to hold for the initial R-context, namely $Es\langle t_i\rangle_{i\in N}$ via $P(t_i)_{i\in N}$, and is again shown to hold for each new R-context generated, namely $Es\langle \Omega_i\rangle_{i\in N}$ via $P(\Omega_i)_{i\in N}$, assuming that it holds for $Es\langle \alpha_i\rangle_{i\in N}$ and $Es\langle \beta_i\rangle_{i\in N}$. As a result, there are two additional checks which must be satisfied before a constraint can be used for our enhanced parallelization. They are:

1. *Invariance* of the required constraint, i.e. $P(\alpha_i)_{i\in N}\ \wedge\ P(\beta_i)_{i\in N} \rightarrow\ P(\Omega_i)_{i\in N}$.
2. *Pre-condition* of the required constraint for the initial R-context, i.e. $P(t_i)_{i\in N}$.

We illustrate how these checks are performed using *sbp* as an example. To check if the constraint is invariant during context preservation, we attempt to simplify the *invariant* condition below:

$$\begin{aligned}
\alpha_3 \geq \alpha_1 + \alpha_2 \wedge \beta_3 \geq \beta_1 + \beta_2 \vdash\ & \Omega_3 \geq \Omega_1 + \Omega_2 \\
\vdash\ & (\text{if } (\beta_3 \leq \alpha_1) \text{ then } \beta_3 + \alpha_2 \text{ else } \alpha_3) \geq min2(\beta_1, \alpha_1 - \beta_2) + (\beta_2 + \alpha_2) \\
\vdash\ & \text{if } \{\ (\beta_3 \leq \alpha_1) \wedge (\beta_1 \geq \alpha_1 - \beta_2) \rightarrow \beta_3 \geq \alpha_1\ ; \\
& \qquad (\beta_3 \leq \alpha_1) \wedge (\beta_1 \leq \alpha_1 - \beta_2) \rightarrow \beta_3 \geq \beta_1 + \beta_2\ ; \\
& \qquad (\beta_3 > \alpha_1) \wedge (\beta_1 \geq \alpha_1 - \beta_2) \rightarrow \alpha_3 \geq \alpha_1 + \alpha_2; \\
& \qquad (\beta_3 > \alpha_1) \wedge (\beta_1 \leq \alpha_1 - \beta_2) \rightarrow \alpha_3 \geq \beta_1 + \beta_2 + \alpha_2\ \} \\
\vdash\ & \text{if } \{\ (\beta_3 \leq \alpha_1) \wedge (\beta_1 \geq \alpha_1 - \beta_2) \rightarrow \mathit{True}\ ; \\
& \qquad (\beta_3 \leq \alpha_1) \wedge (\beta_1 \leq \alpha_1 - \beta_2) \rightarrow \mathit{True}\ ; \\
& \qquad (\beta_3 > \alpha_1) \wedge (\beta_1 \geq \alpha_1 - \beta_2) \rightarrow \mathit{True}; \\
& \qquad (\beta_3 > \alpha_1) \wedge (\beta_1 \leq \alpha_1 - \beta_2) \rightarrow \mathit{True}\ \} \\
\vdash\ & \mathit{True}
\end{aligned}$$

From the initial R-context of *sbp*, the required constraint is satisfied as its precondition since:

$$\begin{aligned}
\beta_3 \geq \beta_1 + \beta_2 \vdash\ & 1 \geq 0 + conv(x) \\
\vdash\ & 1 \geq (\text{if } x == \text{`)'} \text{ then } -1 \text{ else if } x == \text{`('} \text{ then } 1 \text{ else } 0) \\
\vdash\ & (\text{if } x == \text{`)'} \text{ then } 1 \geq -1 \text{ else if } x == \text{`('} \text{ then } 1 \geq 1 \text{ else } 1 \geq 0) \\
\vdash\ & (\text{if } x == \text{`)'} \text{ then } \mathit{True} \text{ else if } x == \text{`('} \text{ then } \mathit{True} \text{ else } \mathit{True}) \\
\vdash\ & \mathit{True}
\end{aligned}$$

Both checks can be systematically carried out via our normalisation procedure for conditional expressions given in Sec 3, in conjunction with simplification of inequalities. They help us confirm that the required constraint is maintained before and throughout context preservation, and could therefore be used for the parallelization of the *sbp* function.

We now present a corresponding theorem for our constraint-enhanced parallelization.

Theorem 2: *Constraint-Enhanced Parallelization*
Consider a linear self-recursive function.

$$\begin{aligned}
f(Nil, \lfloor v_j \rfloor_{j=1}^n) &= E_z \\
f(x : xs, \lfloor v_j \rfloor_{j=1}^n) &= Er\langle f(xs, \lfloor Dr_j\langle v_j\rangle \rfloor_{j=1}^n)\rangle
\end{aligned}$$

This function can be successfully parallelized if the constraint-based context preservation property holds for:

- The R-context of the recursive call, Er, under an required constraint P.
- Each R-context of the accumulative parameters, $\{Dr_j\}_{j\in 1..n}$, under required constraints $\{P_j\}_{j\in 1..n}$, respectively.

Proof : Follows from Theorem 1 with the addition of required constraints. □

As the conditions of Theorem 2 are satisfied, our method can derive the parallel algorithm below.

$$
\begin{array}{ll}
sbp([x]) & = conv(x) \\
sbp(xr{+\!\!+}xs) & = \textit{if } sbp(xs){\leq}uH(xr) \textit{ then } sbp(xs){+}uG(xr) \textit{ else } uK(xr) \\
uH([x]) & = 0 \\
uH(xr{+\!\!+}xs) & = min2(uH(xs), uH(xr){-}uG(xs)) \\
uG([x]) & = conv(x) \\
uG(xr{+\!\!+}xs) & = uG(xs){+}uG(xr) \\
uK([x]) & = 1 \\
uK(xr{+\!\!+}xs) & = \textit{if } uK(xs){\leq}uH(xr) \textit{ then } uK(xs){+}uG(xr) \textit{ else } uK(xr)
\end{array}
$$

None of the functions are equivalent to another. For a more work-efficient parallel algorithm, all four functions must be tupled together.

5 Benefits of Modularity

The overall context preservation task is non-trivial, and may involve substantial normalisation efforts. To keep things simple, we have an approach that is *modular* since the R-contexts for recursive calls and accumulative arguments can be checked separately. Our approach is unique in that it could support both reusability and efficiency – two key topics to be described next.

5.1 Support for Reusability

We shall support reusability by keeping each R-context which satisfies context preservation in a library. Such R-contexts could be reused, when applicable. For example, the recursive calls of *ascan* and *segscan* shares the same R-context, $(\hat{\lambda}\langle\underline{\bullet}\rangle.\alpha_1 {+\!\!+} \underline{\bullet})$, which need only be checked once for context preservation and stored in a library for subsequent reuse.

Likewise, we may also reuse R-contexts with required constraints, where applicable. As a simple example, consider the minimum segment sum problem[Col95].

$$
\begin{array}{ll}
mss([x]) & = x \\
mss(x{:}xs) & = min2(mis(x{:}xs), mss(xs)) \\
mis([x]) & = x \\
mis(x{:}xs) & = \textit{if } x{+}mis(xs){\leq}x \textit{ then } mis(xs){+}x \textit{ else } x
\end{array}
$$

To parallelize *mss*, we must first parallelize its auxiliary recursive function *mis*. Its initial R-context can be normalised to: $(\hat{\lambda}\langle\underline{\bullet}\rangle.\textit{if } \underline{\bullet}{\leq}\beta_1 \textit{ then } \underline{\bullet}{+}\beta_2 \textit{ else } \beta_3) \textit{ where } \{\beta_1{=}0;\ \beta_2{=}x;\ \beta_3{=}x\}$.

The skeletal R-context obtained is identical to the corresponding one for *sbp*. As a result, we could reuse the R-context of *sbp* with its required constraint. Both context preservation and invariance check for the required constraint have been proven, and need not be repeated. However, the initial R-context for *mis* differs slightly from *sbp*. We must therefore check to see if the pre-condition for its required constraint holds, as follows:

$$
\begin{array}{ll}
\beta_3 {\geq} \beta_1 {+} \beta_2 \vdash & x{\geq}0{+}x \\
\vdash & True
\end{array}
$$

As its pre-condition holds, our parallelization theorem could be used to assert the following parallel equations for *mis* (after discovering that one of the newly introduced function is equivalent to *mis*).

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

With *mis* parallelized, we could proceed to transform its parent function *mss* using the associative property of *min2*, and its distributivity over $+$. Due to space constraint, we shall not show its derivation and normalisation, but simply give its final parallel equations below.

$$
\begin{array}{ll}
mss([x]) & = x \\
mss(xr{+}{+}xs) & = min[mss(xr), uT(xr){+}mis(xs), mss(xs)] \\
uT([x]) & = x \\
uT(xr{+}{+}xs) & = min2(uT(xr){+}uG(xs), uT(xs))
\end{array}
$$

5.2 Support for Efficiency

There is an on-going tension between *generality/reusability* on the one end, and *efficiency* on the other. Often, techniques/programs that are general tend to be more widely applicable, but are likely to yield less efficient code, and vice versa. We shall show how we could minimise this tension by searching to exploit specialised scenarios, where possible.

During constraint-based context preservation, we often end up with *inequality* constraints which may represent the weakest requirement for parallelization, but are helpful towards reusability of the corresponding R-contexts. However, it is sometimes beneficial to also exploit stronger *equality* constraints under applicable conditions.

Consider the two initial R-contexts for *sbp* and *mis* below.

$$
\begin{array}{c}
(\hat{\lambda}\langle\underline{\bullet}\rangle.if\ \underline{\bullet} \leq \beta_1\ then\ \underline{\bullet}+\beta_2\ else\ \beta_3)\ where\ \{\beta_1=0;\ \beta_2=conv(x);\ \beta_3=1\} \\
(\hat{\lambda}\langle\underline{\bullet}\rangle.if\ \underline{\bullet} \leq \beta_1\ then\ \underline{\bullet}+\beta_2\ else\ \beta_3)\ where\ \{\beta_1=0;\ \beta_2=x;\ \beta_3=x\}
\end{array}
$$

The inequality constraint $(\beta_3 \geq \beta_1 + \beta_2)$ is applicable to both R-contexts, but we may wish to exploit a *specialised* equality constraint $(\beta_3 == \beta_1 + \beta_2)$ that is only applicable for the R-context of *mis*, since its pre-condition is satisfied as follows:

$$
\begin{array}{ll}
\beta_3 == \beta_1 + \beta_2 & \vdash x == 0 + x \\
& \vdash True
\end{array}
$$

This equality constraint should be exploited where possible, as we may obtain more efficient parallel algorithm (with fewer new functions introduced) as a side-benefit. Before that, we must also check if the *invariance* of this equality constraint can be maintained for the skeletal R-context, as follows:

$$
\begin{array}{ll}
\alpha_3 == \alpha_1 + \alpha_2 \wedge \beta_3 == \beta_1 + \beta_2 & \vdash \Omega_3 == \Omega_1 + \Omega_2 \\
& \vdash (if\ (\beta_3 \leq \alpha_1)\ then\ \beta_3 + \alpha_2\ else\ \alpha_3) == min2(\beta_1, \alpha_1 - \beta_2) + (\beta_2 + \alpha_2) \\
& \vdash if\ \{\ (\beta_3 \leq \alpha_1) \wedge (\beta_1 \geq \alpha_1 - \beta_2) \rightarrow \beta_3 == \alpha_1\ ; \\
& \qquad (\beta_3 \leq \alpha_1) \wedge (\beta_1 \leq \alpha_1 - \beta_2) \rightarrow \beta_3 == \beta_1 + \beta_2\ ; \\
& \qquad (\beta_3 > \alpha_1) \wedge (\beta_1 \geq \alpha_1 - \beta_2) \rightarrow \alpha_3 == \alpha_1 + \alpha_2; \\
& \qquad (\beta_3 > \alpha_1) \wedge (\beta_1 \leq \alpha_1 - \beta_2) \rightarrow \alpha_3 == \beta_1 + \beta_2 + \alpha_2\ \} \\
& \vdash if\ \{\ (\beta_3 \leq \alpha_1) \wedge (\beta_1 \geq \alpha_1 - \beta_2) \rightarrow True\ ; \\
& \qquad (\beta_3 \leq \alpha_1) \wedge (\beta_1 \leq \alpha_1 - \beta_2) \rightarrow True\ ; \\
& \qquad (\beta_3 > \alpha_1) \wedge (\beta_1 \geq \alpha_1 - \beta_2) \rightarrow True; \\
& \qquad False \rightarrow \alpha_1 == \beta_1 + \beta_2\ \} \\
& \vdash True
\end{array}
$$

With this equality constraint, we could eliminate one of the $\{\Omega_i\}_{i \in 1..3}$ subterms by appropriate substitutions. Which should we eliminate? We choose Ω_1 to eliminate, since Ω_3 has potential to be identical to the initial R-context of *mis*, while Ω_2 is simpler in form. Doing so results in the following specialised context preservation.

$$
\begin{array}{l}
(\hat{\lambda}\langle\underline{\bullet}\rangle.if\ \underline{\bullet} \leq \alpha_3 - \alpha_2\ then\ \underline{\bullet}+\alpha_2\ else\ \alpha_3) \circ (\hat{\lambda}\langle\underline{\bullet}\rangle.if\ \underline{\bullet} \leq \beta_3 - \beta_2\ then\ \underline{\bullet}+\beta_2\ else\ \beta_3) \\
\quad \Rightarrow_T (\hat{\lambda}\langle\underline{\bullet}\rangle.if\ \underline{\bullet} \leq \Omega_3 - \Omega_1\ then\ \underline{\bullet}+\Omega_2\ else\ \Omega_3) \\
\qquad where\ \{\ \Omega_2 = \beta_2 + \alpha_2;\ \Omega_3 = if\ (\beta_3 \leq \alpha_3 - \alpha_2)\ then\ \beta_3 + \alpha_2\ else\ \alpha_3\}
\end{array}
$$

Using this specialised R-context, we could now derive the following more compact code for *mis*.

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

Due to the specialised constraints, we have used two functions rather than three used previously. Again, tupling can be applied to give a more work-efficient parallel code.

5.3 Hierarchy of Equality Constraints

To support the reuse of R-context with equality constraints, we shall organise these constraints into a hierarchy. At the base of this hierarchy will be a set of *required constraints* which have been selected to ensure context preservation. Above this will be *equality constraints*, which satisfy the invariance property. (In this paper, we are only interested in specialising via equality constraints. However, extensions to other types of constraints are also possible.)

The constraints in our hierarchy shall be connected by arrows to denote dependencies. All specialised constraints are ultimately dependent on the required constraints since they represent the minimal requirement for context preservation. The specialised constraints themselves may either be *independent*, *mutually dependent*, or *dependent* on each other. A specialised constraint C_1 is said to be dependent on another specialised constraint C_2, if it depends on the latter for its invariance check.

The hierarchy of constraints can support a family of more specialised R-contexts for use in parallelization. In particular, two *independent* constraints may be used separately to support two different R-contexts, or jointly to support a more specialised R-context. Also, *dependent* constraints may be added to provide more specialised R-contexts, where applicable. *Mutual-dependent* constraints must always be used jointly.

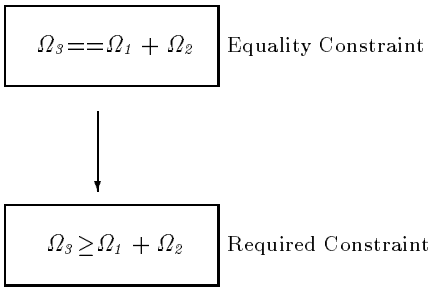

Figure 3: A Simple Hierarchy of Constraints

For example, the skeletal R-context of *sbp* and *mis* shall have a hierarchy of two constraints shown in Figure 3. Required constraint ($\Omega_3 \geq \Omega_1 + \Omega_2$) is at the base, with the equality constraint ($\Omega_3 == \Omega_1 + \Omega_2$) being its special case. It can be used to support *sbp*-like functions with just the required contraint, or to support more specialised R-contexts, such as in *mis*-like functions, with both the equality and required constraints. This is a very simple hierarchy. An example of a more complex hierarchy of constraints is given in Figure 4. There, C_1 & C_2 are mutually-dependent, and so are C_3 & C_4. Also, C_6 depends on C_5, which in turn depends on both C_1 and C_2.

Two issues remain on how the specialised/equality constraints are *collected*, and *applied*.

Definition 4: *Collection of Equality Constraints*
Specialised equality constraints can be collected in two ways:

1. If the required constraint is an inequality, we can assert its equality as a special case, after a suitable invariance check. (An example of this is shown earlier for *mis*.)

2. If two Ω sub-terms of the final R-context are syntactically unifiable, we may use their substitutions from unification as specialised equality constraints. (An example is the R-context of *miptup* to be described in the Appendix.)

Once the specialised equality constraints have been collected, we should find the largest set of equality constraints that is applicable for a given initial R-context, as follows:

Definition 5: *Applicability of Equality Constraints*
A given set of mutual-dependent equality constraints is said to be *applicable* to an initial R-context, if it is satisfied as the latter's pre-condition. The set of applicable equality constraints shall be checked in a bottom-up order according to the constraints' hierarchy.

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 significant interests in the 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 (reduced) 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 [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 constraints, their approach fails to handle certain classes of programs.

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 and be efficiently computed[Pug92]. Our work is complimentary to Pugh's in two respects. Firstly, we may take advantage of practical advances in his constraint technology to support normalisation and invariance/precondition checks. 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.

7 Concluding Remarks

We have formally presented an enhanced method for parallelizing sequential programs. The method relies on the successful preservation of the replicated R-contexts for the recursive call and each accumulative argument. The notion of context preservation is central to our parallelization method. To enhance this methodology further, we introduced a key innovation based on the use of *required and specialised constraints*. To support our extension, a set of powerful normalisation rules have been proposed.

We are currently working on implementation techniques to apply context preservation and invariance/precondition checks automatically. Apart from the heuristic of minimising both the depths and number of occurrences of R-holes, we have also turned to a technique, called *rippling* [BvHSI93], which has been very successful in automated theorem-proving. In our case, we use rippling to repeatedly minimise the difference between actual expression and targeted R-context, until context preservation is achieved. 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 *generalized* R-context to be attempted. This much enhanced potential for parallelization is made possible by our adoption of small but expressive transformation rules, together with appropriate theorems and strategies for guiding their applications.

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.

[Bir87] Richard S. Bird. An introduction to the theory of lists. In *Logic of Programming and Calculi of Discrete Design (Springer Verlag, ed M Broy)*, pages 3–42, 1987.

[Ble75] W.W. Bledsoe. A new method for proving certain Presburger formulae. In *Proc of ICJAI*, pages 15–21, 1975.

[Ble89] Guy E. Blelloch. Scans as primitive parallel operations. *IEEE Trans. on Computers*, 38(11):1526–1538, November 1989.

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

[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.

[Col95] Murray I. Cole. Parallel programming with list homomorphism. *Parallel Processing Letters*, 5(2):191–203, 1995.

[CTH98] W.N. Chin, A. Takano, and Z. Hu. Parallelization via context preservation. In *IEEE Intl Conference on Computer Languages*, Chicago, U.S.A. (submitted), May 1998. IEEE Press. `http://www.iscs.nus.edu.sg/~chinwn/iccl98.ps`.

[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.

[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*, San Diego, California, January 1998. ACM Press (to appear).

[JGS93] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall, 1993.

[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.

[Wad88] Phil Wadler. Deforestation: Transforming programs to eliminate trees. In *European Symposium on Programming*, Nancy, France, (LNCS, vol 300, pp. 344–358), March 1988.

A Maximum Segment Product : A Final Example

Our constraint-enhanced parallelization method is not merely a nice theoretical result, but also practically useful for deriving parallel algorithms for more complex programs. In particular, we could systematically handle recursive functions with conditional and tupled constructs that are often much harder to parallelize. We shall 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 $1 \leq i \leq j \leq n$, taken from the input list. The simplest way of specifying *msp* is via the following generate-and-test algorithm.

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

Here, *segs(xs)* returns all segments for *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 but could be transformed by fusion method [Chi92a, TM95, HIT96] to the following sequential recursive algorithm.

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

The functions *mip* and *mipm* are mutually recursive and do not belong to the linear self-recursive class. Nevertheless, we could use the automated tupling method of [Chi93] to introduce:

$$miptup(xs) = (mip(xs),mipm(xs))$$

before transforming it to:

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

We focus on the parallelization of *miptup* as it must be parallelized before its parent *msp* function. With this definition, we could proceed to extract its R-context[4] (shown below) to see if it could be parallelized.

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

Application of context preservation can now proceed as follows:

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

In the middle of the normalisation, we need to distribute $*$ into *max2* and *min2*, but this could only be done with the following distributive laws.

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

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

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 could be applied. Doing so results in the following successful context preservation.

$\Rightarrow_T$ { add selected constraints & normalise further }
$(\hat{\lambda}\langle\underline{\bullet}\rangle.\ let\ (u,v)=\underline{\bullet}\ in$
if { $(\beta_1>0) \wedge (\alpha_1>0) \rightarrow (max2(max2(\alpha_2,\alpha_3*\beta_2),(\alpha_3*\beta_3)*u)),min2(min2(\alpha_4,\alpha_5*\beta_4),(\alpha_5*\beta_5)*v)));$
$(\beta_1>0) \wedge \neg(\alpha_1>0) \rightarrow (max2(max2(\alpha_6,\alpha_7*\beta_4),(\alpha_7*\beta_5)*v)),min2(min2(\alpha_8,\alpha_9*\beta_2),(\alpha_9*\beta_3)*u)))$
$\neg(\beta_1>0) \wedge (\alpha_1>0) \rightarrow (max2(max2(\alpha_2,\alpha_3*\beta_6),(\alpha_3*\beta_7)*v)),min2(min2(\alpha_4,\alpha_5*\beta_8),(\alpha_5*\beta_9)*u)));$
$\neg(\beta_1>0) \wedge \neg(\alpha_1>0) \rightarrow (max2(max2(\alpha_6,\alpha_7*\beta_8),(\alpha_7*\beta_9)*u)),min2(min2(\alpha_8,\alpha_9*\beta_6),(\alpha_9*\beta_7)*v)))$ }
st $\{\alpha_3\geq 0;\ \alpha_5\geq 0;\ \alpha_7\leq 0;\ \alpha_9\leq 0\}$
$\Rightarrow_T$ { re-group branches & form skeletal R-context }
$\hat{\lambda}\langle\underline{\bullet}\rangle.\ let\ (u,v)=\underline{\bullet}\ in\ if\ \Omega_1>0\ then\ (max2(\Omega_2,\Omega_3*u),min2(\Omega_4,\Omega_5*v))$
$else\ (max2(\Omega_6,\Omega_7*v),min2(\Omega_8,\Omega_9*u))$
where { $\Omega_1=\alpha_1*\beta_1;\ \Omega_2=if\ \alpha_1>0\ then\ max2(\alpha_2,\alpha_3*\beta_2)\ else\ max2(\alpha_6,\alpha_7*\beta_8);$
$\Omega_3=if\ \alpha_1>0\ then\ \alpha_3*\beta_3\ else\ \alpha_7*\beta_9;$
$\Omega_4=if\ \alpha_1>0\ then\ min2(\alpha_4,\alpha_5*\beta_4)\ else\ min2(\alpha_8,\alpha_9*\beta_6);$
$\Omega_5=if\ \alpha_1>0\ then\ \alpha_5*\beta_5\ else\ \alpha_9*\beta_7;$
$\Omega_6=if\ \alpha_1>0\ then\ max2(\alpha_2,\alpha_3*\beta_6)\ else\ max2(\alpha_6,\alpha_7*\beta_4)\ ;$
$\Omega_7=if\ \alpha_1>0\ then\ \alpha_3*\beta_7\ else\ \alpha_7*\beta_5;$
$\Omega_8=if\ \alpha_1>0\ then\ min2(\alpha_4,\alpha_5*\beta_8)\ else\ min2(\alpha_8,\alpha_9*\beta_2);$
$\Omega_9=if\ \alpha_1>0\ then\ \alpha_5*\beta_9\ else\ \alpha_9*\beta_3$ }

We are now left with two checks on the *pre-condition* and *invariance* of the required constraints. Both checks are valid and can be mechanically performed. The reader may like to try them out as an exercise. For example, the invariance check is:

$$(\alpha_3\geq 0 \wedge \alpha_5\geq 0 \wedge \alpha_7\leq 0 \wedge \alpha_9\leq 0) \wedge (\beta_3\geq 0 \wedge \beta_5\geq 0 \wedge \beta_7\leq 0 \wedge \beta_9\leq 0)$$
$$\rightarrow (\Omega_3\geq 0 \wedge \Omega_5\geq 0 \wedge \Omega_7\leq 0 \wedge \Omega_9\leq 0)$$

While our R-contexts have been renamed and normalised for greater reusability, they may yield somewhat less efficient codes. To improve matter, we shall search for equality constraints that could be used. We may attempt to unify some of the $\{\Omega_i\}_{i\in 1..9}$ subterms in the resulting R-context. In particular, four equality constraints may be attempted in the first round, namely $C_1 = (\Omega_3 == \Omega_5)$, $C_2 = (\Omega_7 == \Omega_9)$, $C_3 = (\Omega_2 == \Omega_6)$, $C_4 = (\Omega_4 == \Omega_8)$. The checks for invariance do hold for the above equality constraints with C_1 mutually-dependent on C_2, and C_3 mutually-dependent on C_4.

In the second round, we substitute the equality constraints collected and perform normalisation to obtain the following new sub-terms for our R-context.

$\Omega_2=if\ \alpha_1>0\ then\ max2(\alpha_2,\alpha_3*\beta_2)\ else\ max2(\alpha_2,\alpha_7*\beta_4);$
$\Omega_3=if\ \alpha_1>0\ then\ \alpha_3*\beta_3\ else\ \alpha_7*\beta_7;$
$\Omega_4=if\ \alpha_1>0\ then\ min2(\alpha_4,\alpha_3*\beta_4)\ else\ min2(\alpha_4,\alpha_7*\beta_2);$
$\Omega_7=if\ \alpha_1>0\ then\ \alpha_3*\beta_7\ else\ \alpha_7*\beta_3;$

This time round, unification is only possible for $C_5 = (\Omega_3 == \Omega_7)$[5] whose invariance can be shown to hold. In the final round, the Ω_3 sub-term is simplified, as follows:

$\Omega_3 = if\ \alpha_1>0\ then\ \alpha_3*\beta_3\ else\ \alpha_3*\beta_3;$
$= \alpha_3*\beta_3$

This now allows the equality constraint $C_6 = (\Omega_1 == \Omega_3)$ to hold via unification. The equality constraints are now gathered into a hierarchy, shown in Figure 4.

Depending on the number of applicable equality constraints, a family of parallel algorithms is being covered by the same skeletal R-context and its hierarchy of constraints. Consider a hypothetically generalized function.

$gmiptup(x{:}xs) = let\ (u,v)=gmiptup(xs)\ in$
$if\ a_1*x>0\ then\ (max2(a_2*x,a_3*x*u),min2(a_4*x,a_5*x*v))$
$else\ (max2(a_6*x,a_7*x*v),min2(a_8*x,a_9*x*u))$

[5] This equality constraint may appear to contradict with the required constraint C_0. However, as the equality constraints are being used independently of the required constraints and in checking for syntactically equivalent functions, this contradition can be safely ignored.

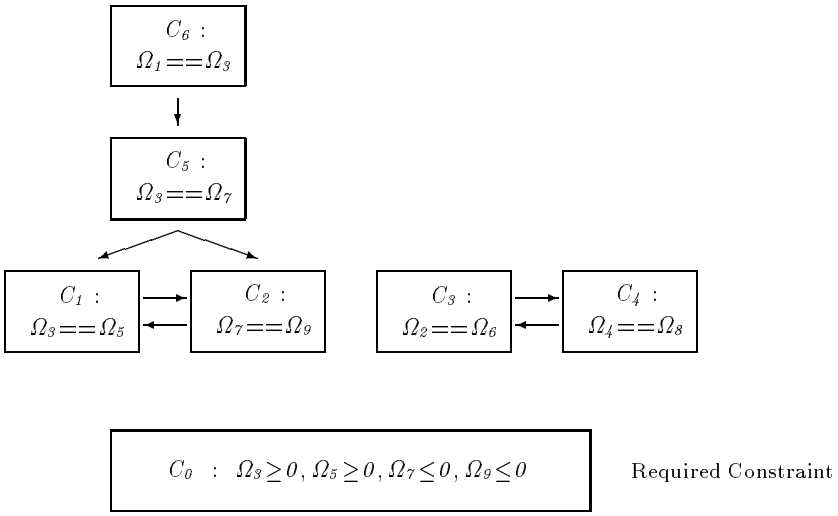

Figure 4: A More Complex Hierarchy of Constraints

This definition has the same skeletal R-context as *miptup*. If $a_3 = a_5$ and $a_7 = a_9$, we could exploit the equality constraints of C_1 and C_2. If $a_2 = a_6$ and $a_4 = a_8$, we could exploit the equality constraints of C_3 and C_4. In addition, if $a_3 = a_7$ we could apply the constraint C_5, but constraint C_6 could only be used if we also have $a_1 = a_3$ too.

The function *miptup* is actually a special case of *gmiptup*, where all the gathered equality constraints are applicable. Also, two unknown functions derived happens to be identical to the components of *miptup* and could thus be replaced. As a result, we can derive a very compact and efficient parallel algorithm shown below.

```
miptup([x])      = (x,x)
miptup(xr++xs)   = let {(a,b)=miptup(xr); (u,v)= miptup(xs) in
                     if uH(xr)>0 then (max2(a,uH1(xr)*u),min2(b,uH1(xr)*v))
                     else (max2(a,uH1(xr)*v),min2(b,uH1(xr)*u))
uH1([x])         = x
uH1(xr++xs)      = uH1(xr)*uH1(xs)
```

With these equations, we could proceed to parallelize the parent function *msptup* using context preservation and normalisation. The final equations are presented below.

```
msp(xr++xs)      = let {(a,b)=miptup(xr); (u,v)= miptup(xs) in
                     max[msp(xr),msp(xs),mfp(xr)+a,mfp(xr)+b,mfpm(xr)+a,mfpm(xr)+b]
mfp([x])         = x
mfp(xr++xs)      = if uH1(xs)>0 then max2(mfp(xr)*uH1(xs),mfp(xs))
                     else max2(mfpm(xr)*uH1(xs),mfp(xs));
mfpm([x])        = x
mfpm(xr++xs)     = if uH1(xr)>0 then min2(mfpm(xr)*uH1(xs),mfpm(xs))
                     else min2(mfp(xr)*uH1(xs),mfpm(xs))
```