Parallelizing Conditional Recurrences

W.N. CHIN[1] and J. DARLINGTON[2] and Y. GUO[2]

[1] DISCS, National University of Singapore
[2] DoC, Imperial College of Science, Technology and Medicine

Abstract. Recursive functions which use conditional constructs are common in functional (and imperative) programs. We present a collection of techniques for handling such functions for a parallel synthesis method. These techniques can help us enlarge the class of sequential functions which could be systematically transformed to parallel equivalent.

1 Introduction

Most programs are more easily expressed via sequential code. Functional programs are no exception. As a simple example, consider the following program to find the product of a list.

```
data List a = Nil  | Cons(a,List a) ;
product(Nil)       = 1;
product(x:xs)      = x*product(xs);
```

The first statement declares a *List* data type with two constructors *Nil* and *Cons*. This *List* type has a shorthand notation $[a_1, a_2, .., a_n]$ for a list of n-elements where $n \geq 0$, and an infix notation *(x:xs)* for *Cons(x,xs)*. The next two statements are pattern-matching equations of *product* for a base case and a recursive case. In the latter case, the result is computed sequentially via *x*product(xs)*. By using the fact that the operator * is associative, the parallel synthesis method proposed in [Chi95] can derive the following definition for *product*.

```
product(Nil)       = 1;
product([x])       = x;
product(xr++xs)    = product(xr) * product(xs);
```

Note the ++ operator is to split an input list into two halves. Details of our proposed parallelization methodology is described in [Chi95]. It comprises four main stages, which shall be briefly summarised below using *product* function as an example.

Procedure 1: *Four-Stage Parallel Synthesis Method*

Stage 1: Determine a desired *pre-parallel form* for the initial recursive equation.
Stage 2: Obtain a second recursive equation with the *same* pre-parallel form.
Stage 3: Use *second-order generalisation* to obtain an *equation template* from the two pre-parallel equations. This template may have one or more unknown functional variable(s).
Stage 4: *Derive* the unknown functional variable(s).

Stage 1 tries to obtain a desired pre-parallel form for the initial recursive equation. This is assisted by two simple heuristics:

Definition 1: *Heuristics for Desired Pre-Parallel Form*

(H1) All function calls (e.g. *, *product*), leading to the recursion variables (e.g. *xs*) should be either *associative* or *distributive*.
(H2) The depth of recursion variables (e.g. *xs*) from the root of its equation should be as *shallow* as possible.

The first heuristic is needed to match the associativity of the * operator in the LHS. The second heuristic is to reduce the number of function operators leading to the recursion variables. As these have to be associative (or distributive), the fewer the better. Guided by the two heuristics, we can obtain:

$$product(\underline{[x]}{+}{+}xs) \quad = \underline{x} * product(xs)$$

The underlined sub-expressions are known as *holes* of the pre-parallel equation. Formally, each maximal sub-term, not including any recursion variables, shall be known as a *hole* of the pre-parallel equation. Given an expression e, we can decompose it via $\widehat{e^P}\langle h_i\rangle_{i\in 1..n}$ where $\widehat{e^P}\langle\rangle$ is a *pre-parallel context* (or *form*), and $\{h_i\}_{i\in 1..n}$ are the *pre-parallel holes*.

Stage 2 is to obtain a second recursive equation with *identical* pre-parallel context as the first equation. We unfold the self-recursive call followed by transformations (guided by the earlier two heuristics), to obtain:

$$product(\underline{([x]{+}{+}[y])}{+}{+}xs) \quad = \underline{(x*y)} * product(xs)$$

With two similar pre-parallel equations, Stage 3 will attempt a matching cum generalisation procedure to obtain a *parallel template equation*. The matching process will apply (possibly second-order) generalisations to those pre-parallel holes which *mismatch*. In the LHS, a pair of mismatched holes, *[x]* (first equation) and *[x]++[y]* (second equation), is resolved by replacing the two sub-terms by a new variable, *xr*, generalising the LHS to *product(xr++xs,c)*. The new LHS parameter, *xr++xs*, has two variables, *xr* and *xs*, called *active* and *passive* recursion variables, respectively. In the RHS, a mismatch occurs for the expressions: *x* (first equation) and *(x*y)* (second equation). This mismatch is resolved by supplying a generalised expression *H(xr)* where *H* is a function-type variable and *xr* is the active recursion variable. This generalisation is second-order since a function variable, *H*, is used. The resulting parallel template is:

$$product(xr{+}{+}xs) = H(xr) * product(xs);$$

Lastly, Stage 4 is used to derive inductive definitions for the unknown functions. This stage involves *instantiation* of the active recursion variable; followed by *simplification* and *induction* to obtain identical pre-parallel forms (for both the LHS and RHS), before *unification* to obtain new definitions for the unknown functions. In the case of *product*, we can obtain:

$$\begin{array}{ll} H(Nil) & = 1; \\ H(x{:}xr) & = x * H(xr); \end{array}$$

This definition of H is syntactically identical to the definition of *product*. Hence, we can assert that $H \equiv product$ holds.

This paper proposes an extension, to handle conditional recurrences, for our parallelization methodology. We introduce some notations and a key idea on conditional normalisation (in Sec 2). This is followed by a collection of techniques to handle different sub-classes of conditional recurrences (in Sec 3, 4 and 5). Lastly, related work are described before a conclusion (in Sec 6).

2 Conditional Recurrence and its Normalisation

We consider a first-order functional language. Valid expressions include variables (v), data constructors (C), functions (f), *let* and *if* statements. We shall abbreviate a list of terms $t_1, \ldots, t_n$ by $\vec{t}$, so that a function call $f(t_1, \ldots, t_n)$ could be abbreviated as $f(\vec{t}\,)$, or more precisely as $f(t_i)_{i \in 1..n}$. We introduce a special context notation with multiple holes.

Definition 2: *Context with Multiple Holes*
A *hole*, $\langle\rangle_m$, is a special variable labelled with a number, m.

A *context*, $\widehat{e}\langle\rangle$, is an expression with a finite number of holes, defined by the grammar:

$$\begin{array}{ll} \widehat{e}\langle\rangle & ::= v \;\mid \langle\rangle_m \;\mid C(\widehat{e}\langle\rangle^*) \;\mid\; f(\widehat{e}\langle\rangle^*) \;\mid \mathit{let}\ \{p = \widehat{e}\langle\rangle\}^+\ \mathit{in}\ \widehat{e}\langle\rangle \\ & \;\mid \mathit{if}\ \widehat{e}\langle\rangle\ \mathit{then}\ \widehat{e}\langle\rangle\ \mathit{else}\ \widehat{e}\langle\rangle\ ; \end{array}$$

Note that S^* denotes zero or more occurrences of S, while S^+ denotes one or more occurrences.

Each expression e can be decomposed into a context $\widehat{e}\langle\rangle$ and a sequence of sub-terms $[t_i]_{i \in 1..n}$ using the notation $e \equiv \widehat{e}\langle t_i\rangle_{i \in 1..n}$. This stands for the substitutions of sub-terms, $t_1, \ldots, t_n$, into their respective holes, $\langle\rangle_1, .., \langle\rangle_n$, for context $\widehat{e}\langle\rangle$. We define self-recursive function calls and variables, as follows:

Definition 3: *Self-Recursive Function Calls and Variables*
A function, f_i, is said to be *self-recursive* if its (mutual) recursive set of functions consists of just $\{f_i\}$. Each occurrence of a self-recursive call, $f_i(\vec{t}\,)$, in the RHS of f_i shall be annotated as $f_{i\#}(\vec{t}\,)$.
Consider $\mathit{let}\ (\vec{v}\,){=}f_{i\#}(\vec{t}\,)\ \mathit{in}\ e$. The variables $\vec{v}$ will be called *self-recursive variables*, and shall be annotated as $\vec{v}_\#$.

We mark a sub-expression, e, of an equation as either e^R (*recursive*) or $e^{\overline{R}}$ (*non-recursive*), depending on whether it has self-recursive functions/variables or not. To facilitate the parallelization of conditional recurrences, we propose a guarded conditional that is to be used as intermediate representation during transformation. This conditional has the form:

$$gc \quad ::= \mathit{cond}\ \{b_i \rightarrow\ e_i\}_{i \in 1..n}$$

where *exactly* one of the tests $b_1, .., b_n$ can be true; while $e_1, .., e_n$ are their respective branches.

To obtain suitable pre-parallel form, we shall transform each recursive expression with conditional(s) into a *canonical form* during Stages 1 and 2. In this form, each self-recursive call has at most one outer conditional (and one outer *let* construct), as specified next.

Definition 4: *Normalised Conditional Form*
An expression, without any outer conditional (or *let* construct) prior to its self-recursive calls, is defined by:

$$e^F \;::=\; e^{\overline{R}} \mid v \;\mid v_{\#} \;\mid c(e^{F\,*}) \;\mid f(e^{F\,*}) \;\mid f_{\#}(e^{\overline{R}\,*})$$

Correspondingly, an expression with at most one outermost conditional and *let* for each self-recursive call is defined by:

$$e^N ::= \mathit{let}\ \{p = e^F\}^+\ \mathit{in}\ e^C \mid e^C$$
$$e^C ::= e^F \;\mid \mathit{cond}\ \{e^F \to e^F\}^*$$

An expression is said to be a *normalised conditional expression* if (i) it satisfies grammar term e^N, and (ii) the outermost conditional has at most one non-recursive branch.

Though only one outer guarded conditional is allowed prior to self-recursive calls of e^C, inner conditionals are still permitted in non-recursive expressions via $e^{\overline{R}}$ in the definition of e^F. To obtain normalised conditional expressions, we have the following procedure.

Procedure 2: *Normalisation of Conditional Expressions*

Step 1 Convert each simple (recursive) conditional to guarded conditional.
Step 2 Float out conditional (and let) constructs, where possible.
Step 3 Flatten nested conditional (and let) constructs.
Step 4 Combine all non-recursive branches of outermost conditional.
Step 5 Simplify conditionals, where possible.

The conditional normalisation rules are given in Figure 1. (Note that $\widehat{e^I}\langle\rangle$ is a special context which must not contain any conditional/*let* constructs, while $\widehat{e^L}\langle\rangle^V$ must not contain *let* constructs with bound variables from V.) To ensure that these rules are applied deterministically, we always transform the leftmost, outermost sub-term first.

The normalisation steps will be used to help parallelize three sub-classes of conditional recurrences, codenamed **LC1** with a *single recursive branch*, **LC2** with *multiple recursive branches*, and **LC3** with *recursive test(s).*

3 Single Recursive Branch (LC1)

A normalised conditional recurrence is said to have a *single recursive branch* if its outermost conditional has exactly one recursive branch. An example is:

$$\begin{array}{ll} \mathit{ssc}(\mathit{Nil},p) & = \mathit{Nil}\ ; \\ \mathit{ssc}(x{:}xs,p) & = \mathit{if}\ p(x)\ \mathit{then}\ p{:}\mathit{ssc}(xs,p)\ \mathit{else}\ \mathit{Nil}\ ; \end{array}$$

Using the conditional normalisation procedure, we can obtain the following pre-parallel equations in Stages 1 & 2, respectively.

$$\begin{array}{lll} \mathit{ssc}(\underline{[x]}{+}{+}xs,\underline{p}) & = \mathit{cond}\ \{ & \underline{p(x)} \to \underline{[x]}{+}{+}\mathit{ssc}(xs,\underline{p}); \\ & & \underline{\neg\, p(x)} \to \underline{\mathit{Nil}}\ \} \\ \mathit{ssc}(\underline{([x]{+}{+}[y])}{+}{+}xs,\underline{p}) & = \mathit{cond}\ \{ & \underline{p(x) \wedge p(y)} \to \underline{([x]{+}{+}[y])}{+}{+}\mathit{ssc}(xs,\underline{p}); \\ & & \underline{\neg\,(p(x) \wedge p(y))} \to \underline{\mathit{cond}\ \{p(x) \to [x]{+}{+}\mathit{Nil};\ \neg\, p(x) \to \mathit{Nil}\}}\} \end{array}$$

$$
\begin{array}{ll}
(\mathcal{N}1) & \textit{if } b \textit{ then } e_1 \textit{ else } e_2 \equiv \textit{cond } \{\, b \to e_1;\, \neg\, b \to e_2\} \\
& \qquad \textit{IF } (b \in e^R) \vee (e_1 \in e^R) \vee (e_2 \in e^R) \\
(\mathcal{N}2a) & \widehat{e^I}\langle \textit{cond } \{b_i \to\ e_i\}_{i\,\in\,N}\rangle \\
& \equiv \textit{cond } \{b_i \to\ \widehat{e^I}\langle e_i\rangle\}_{i\,\in\,N} \\
& \qquad \textit{IF } \exists i\ \in\ N.\,(b_i\ \in\ e^R) \vee (e_i\ \in\ e^R) \\
(\mathcal{N}2b) & \widehat{e^L}\langle \textit{let } \{v_i \to\ e_i\}_{i\,\in\,N}\ \textit{ in } e\rangle^V \\
& \equiv \textit{let } \{v_i \to\ e_i\}_{i\,\in\,N}\ \textit{ in } \widehat{e^L}\langle e\rangle^V \\
& \qquad \textit{IF } (\exists i\ \in\ N.\, e_i\ \in\ e^R) \vee (e\ \in\ e^R)\ \&\ V = \bigcup_{i\in N} \textit{vars}(e_i)) \\
(\mathcal{N}3a) & \textit{cond } B \cup \{bi \to\ \textit{cond } \{bi_j \to ei_j\}_{j\in M}\}\ \equiv\ \textit{cond } B \cup \{bi\ \wedge bi_j \to ei_j\}_{j\in M} \\
(\mathcal{N}3b) & \textit{cond } B \cup \{\textit{cond } \{bi_j \to ei_j\}_{j\,\in\,M} \to e_i\}\ \equiv \textit{cond } B \cup \{\bigvee_{j\,\in\,M} (bi_j\ \wedge\ ei_j) \to e_i\} \\
(\mathcal{N}3c) & \textit{let } \{p_i = e_i\}_{i\in N}\ \textit{ in (let } \{p_i = e_i\}_{i\in M}\ \textit{ in } e)\ \equiv \textit{let } \{p_i = e_i\}_{i\in(N\cup M)}\ \textit{ in } e \\
(\mathcal{N}3d) & \textit{let } \{p_i = e_i\}_{i\in N}\ \cup\ \{v = \textit{cond } \{b_j \!\to er_j\}_{j\in M}\}\ \textit{ in } e \\
& \equiv \textit{let } \{p_i = e_i\}_{i\in N}\ \textit{ in cond } \{b_j \!\to [er_j/v]e\}_{j\in M} \\
& \qquad \textit{IF } \exists j \in M.\,(b_j \in e^R \vee er_j \in e^R) \\
(\mathcal{N}4) & \textit{cond } \{b_i \to e_i\}_{i\,\in\,N\,\cup\,M} \equiv \textit{cond } \{b_i \to e_i\}_{i\,\in\,N}\ \cup\ \{\bigvee_{j\,\in\,M} b_j \to \\
& \qquad \textit{cond } \{b_i \to e_i\}_{i\,\in\,M}\}\ \textit{IF } \forall i \in M.\,(b_i \in e^{\overline{R}} \wedge e_i \in e^{\overline{R}}) \\
(\mathcal{N}5a) & \textit{cond } \{b_i \to e\}_{i\,\in\,N} \equiv e \\
(\mathcal{N}5b) & \textit{cond } B\ \cup\ \{\textit{True} \to e\}\ \equiv\ e \\
(\mathcal{N}5c) & \textit{cond } B\ \cup\ \{\textit{False} \to e\} \equiv\ \textit{cond } B
\end{array}
$$

Fig. 1. Rules for Conditional Normalisation

In Stage 3, second-order generalisation can be applied to obtain the following parallel equation template with unknown functions H and G.

$$\textit{ssc(xr++xs,p)} \qquad = \textit{cond } \{H(xr,p) \to \textit{xr++ssc(xs,p)};\ \neg\, H(xr,p) \to G(xr,p)\}\ ;$$

In Stage 4, we can infer that $G \equiv \textit{ssc}$, while H has the definition:

$$
\begin{array}{ll}
\textit{H(Nil,p)} & = \textit{True}\ ; \\
\textit{H(x:xs,p)} & = p(x) \wedge H(xs,p)\ ;
\end{array}
$$

The new H function can also be subjected to the parallel synthesis method in order to obtain:

$$\textit{H(xr++xs,p)} \qquad = H(xr,p) \wedge H(xs,p)\ ;$$

The above equations are parallel but not efficient due to the presence of redundant H calls. This is indicated by the presence of calls with common recursion argument, e.g *H(xr,p)* and *ssc(xr,p)*. To eliminate this redundancy, we can use the automatic tupling method of [Chi93] to introduce a new definition:

$$\textit{tup(xs,p)} \qquad = (H(xs,p),\ \textit{ssc(xs,p)})\ ;$$

before it is transformed to:

$$
\begin{array}{ll}
\textit{tup([x],p)} & = (p(x),\ \textit{if } p(x) \textit{ then [x] else Nil})\ ; \\
\textit{tup(xr++xs,p)} & = \textit{let } \{(u,v)=\textit{tup(xr,p)};\ (a,b)=\textit{tup(xs,p)}\}\ \textit{in} \\
& \quad (u \wedge a,\ \textit{if } u \textit{ then xr++b else } v)\ ;
\end{array}
$$

Such functions are both parallel and efficient. Using a tuple of values, the results of function calls have been memoised to avoid redundant computation.

4 Multiple Recursive Branches (LC2)

The next sub-class of conditional recurrence we consider are those with *multiple recursive branches*. An example (with two recursive branches) is:

$$
\begin{array}{ll}
\mathit{filter(Nil,p)} & = \mathit{Nil}\ ; \\
\mathit{filter(x{:}xs,p)} & = \mathit{if}\ p(x)\ \mathit{then}\ x{:}\mathit{filter}(xs,p)\ \mathit{else}\ \mathit{filter}(xs,p)\ ;
\end{array}
$$

To handle conditional recurrence with multiple branches, we shall use a pre-processing technique to convert each such recurrence to one with at most a single recursive branch. The steps are outlined below:

Procedure 3: *Combining Multiple Recursive Branches*

Step 1 Find a common pre-parallel form for the different recursive branches.
Step 2 Combine and unify the multiple recursive branches.

Step 1 is to find a common pre-parallel form for all the recursive branches. This is achieved by obtaining a desired pre-parallel form for each of the branches, before attempting to unify them via suitable transformations. In the case of *filter* function, the desired pre-parallel form for the first recursive branch is $\underline{[x]}{+}{+}\mathit{filter}(xs,\underline{p})$, while that of the second branch is $\mathit{filter}(xs,\underline{p})$. To make these two pre-parallel expressions unifiable, we would transform the second recursive branch to $\underline{\mathit{Nil}}{+}{+}\mathit{filter}(xs,\underline{p})$.

With a common pre-parallel context, Step 2 would now combine all recursive branches into a single conditional. This is achieved by rule $\mathcal{N}4$ but with M being the set of recursive branches. After that, the multiple recursive branches are unified into a single branch by pushing the outer conditional into each hole of the common pre-parallel context. Assuming that $\widehat{e^P}\langle\rangle$ is a common pre-parallel context for multiple branches, the following rule is needed.

$$
(\mathcal{N}6)\quad \mathit{cond}\ \{b_i \to\ \widehat{e^P}\langle ti_j\rangle_{j\in M}\}_{i\in N}\ \equiv \widehat{e^P}\langle \mathit{cond}\{b_i \to\ ti_j\}_{i\in N}\rangle_{j\in M}
$$
$$
\mathit{IF}\ \forall i\ \in\ N.\, j\ \in\ M.\,(ti_j\ \in\ \overline{e^R})
$$

Applying it to *filter*, we obtain:

$$
\begin{array}{ll}
\mathit{filter}([x]{+}{+}xs,p) & = \mathit{cond}\ \{p(x) \to [x]{+}{+}\mathit{filter}(xs,p);\ \neg p(x) \to \mathit{Nil}{+}{+}\mathit{filter}(xs,p)\}\ ;\ \mathcal{N}6 \\
& = (\mathit{cond}\ \{p(x) \to [x];\ \neg p(x) \to \mathit{Nil}\}) \\
& \qquad {+}{+}\mathit{filter}(xs,\mathit{cond}\ \{p(x) \to p;\ \neg p(x) \to p\})\ ;\ \mathcal{N}5a \\
& = (\mathit{cond}\ \{p(x) \to [x];\ \neg p(x) \to \mathit{Nil}\}){+}{+}\mathit{filter}(xs,p)\ ;
\end{array}
$$

Once we have unified the multiple recursive branches into a single recursive branch, we can proceed with the earlier technique for the **LC1** sub-class.

5 Recursive Tests (LC3)

Another scenario for conditional recurrence involves multiple occurrences of a recursive call in the conditional's test, and its branches. An example is:

$$
\begin{array}{ll}
\mathit{max}([x]) & = x\ ; \\
\mathit{max}(x{:}xs) & = \mathit{if}\ x{>}\mathit{max}(xs)\ \mathit{then}\ x\ \mathit{else}\ \mathit{max}(xs)\ ;
\end{array}
$$

The recursive call *max(xs)* appears in both the test and a branch of the conditional. To make this equation into a linear recurrence, we use a *let* statement to abstract out the recursive call, as follows:

$$max([x]{+}{+}xs) \quad = let\ z_{\#}{=}max(xs)\ in\ if\ \underline{x}{>}z_{\#}\ then\ \underline{x}\ else\ z_{\#}\ ;$$

Note that variable z is a proxy for the self-recursive call *max(xs)*, and is thus marked with the recursive annotation, #. At the moment, there are two pre-parallel holes (shown underlined). However, since these holes represent the same variable x, we could reduce the number of pre-parallel holes (also an effect of heuristic H2), by abstracting it via a *let* variable, as follows:

$$max([\underline{x}]{+}{+}xs) \quad = let\ \{z{=}max(xs);\ v{=}\underline{x}\}\ cond\ \{v{>}z \rightarrow v;\ \neg\ v{>}z \rightarrow z\}\ ;$$

Subsequently, in Stage 2, we perform a further unfold of *max* before applying the conditional normalisation procedure to obtain:

$$\begin{aligned} max([x]{+}{+}([y]{+}{+}xs)) = let\ &\{z'_{\#}{=}max(xs)\}\ in \\ &cond\ \{(y{>}z'_{\#}) \wedge (x{>}y) \rightarrow x\ ; \\ &\quad (y{>}z'_{\#}) \wedge \neg(x{>}y) \rightarrow y\ ; \\ &\quad \neg(y{>}z'_{\#}) \wedge (x{>}z'_{\#}) \rightarrow x\ ; \\ &\quad \neg(y{>}z'_{\#}) \wedge \neg(x{>}z'_{\#}) \rightarrow z'_{\#}\}\ ; \end{aligned}$$

At this point, the second pre-parallel equation has four conditional branches, while the original pre-parallel equation has only two. To simplify this more complex conditional, we use another procedure with three steps: (Step 1) Choose a non-recursive test to float out, (Step 2) Simplify the inner guarded conditionals, and (Step 3) Unify the inner guarded conditionals.

The three tests in the outermost conditional are $\{y{>}z'_{\#},\ x{>}y,\ x{>}z'_{\#}\}$. Two of the tests are *recursive*, but $x{>}y$ is *non-recursive* since it did not depend on the self-recursive variable, $z'_{\#}$. Step 1 will choose this non-recursive test to float out, using the rule: $e \equiv cond\ \{(x > y) \rightarrow\ e; \neg(x > y) \rightarrow\ e\}$.

This step pushes all recursive tests and branches to inner conditionals. After this, the non-recursive assertions, $x{>}y$ and $\neg(x{>}y)$, can be used to simplify the inner conditionals in Step 2. Applying appropriate simplification results in:

$$\begin{aligned} max(([x]{+}{+}[y]){+}{+}xs) = let\ z'{=}max(xs)\ in\ cond \\ \{\ (x{>}y) \rightarrow\ \ let\ v{=}\underline{x}\ in\ \ cond\ \{\ (v{>}z') \rightarrow v\ ;\ (v{\leq}z') \rightarrow z'\ \}\ ; \\ (x{\leq}y) \rightarrow\ \ let\ v{=}\underline{y}\ in\ \ cond\ \{\ (v{>}z') \rightarrow v\ ;\ (v{\leq}z') \rightarrow z'\ \}\ \}\ ; \end{aligned}$$

The two inner recursive conditionals now have the same pre-parallel form. In Step 3, we unify these multiple recursive branches to obtain:

$$\begin{aligned} max(([x]{+}{+}[y]){+}{+}xs) \ = let\ &z'{=}max(xs)\ in \\ let\ &v{=}\underline{cond\ \{x{>}y \rightarrow x\ ;\ x{\leq}y \rightarrow y\}}\ in \\ &cond\ \{(v{>}z') \rightarrow v\ ;\ (v{\leq}z') \rightarrow z'\}\ ; \end{aligned}$$

This last equation has the same pre-parallel form as the first equation. Proceeding with Stages 3 & 4 can yield the expected parallel equation.

6 Related Work & Conclusion

A well-known approach for synthesizing parallel functional programs is through Bird-Meertens formalism [Ski90]. The emphasis there is to construct programs using a small set of higher-order functions (such as *map*, *reduce*) from which it is often possible to directly detect/derive the divide-and-conquer homomorphism *without induction*. However, the homomorphism sub-class is rather limiting since many programs (e.g. *ssc*) lie outside it. Cole realised this limitation and attempted to identify a larger class of so-called *near-homomorphism* [Col93]. His attempt is somewhat informal and appears difficult to automate.

In traditional imperative languages (e.g. Fortran), there are also many on-going efforts at developing sophisticated techniques for parallelizing iterative loops. Lately, we became aware of a more systematic method for parallelizing complex scans and reductions [FG94]. This method is based on a parallel reduction of *function composition* where function-type values are propagated. This is allowed because function composition is associative. However, the complexity of the functions propagated could get progressively worse unless they match a certain *template* form. The steps needed for finding such template form is similar to our technique for finding a parallel template equation. Our technique was discovered independently with the initial methodology presented in [Chi90].

As illustrated in this paper, a more systematic approach is possible by augmenting the unfold/fold rules [BD77] with induction/generalisation steps. There is no need to restrict user programs to a closed set of higher-order functions. Our method is capable of generating suitable parallel equations, even in the presence of conditional expressions. The systematic formulation presented is an initial step towards an automation of the proposed parallelization methodology.

Acknowledgement: Work supported by EEC KIT143 grant for international scientific cooperation, and NUS research grant RP920614.

References

[BD77] R.M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of ACM*, 24(1):44–67, January 1977.

[Chi90] Wei-Ngan Chin. *Automatic Methods for Program Transformation.* PhD thesis, Imperial College, University of London, March 1990.

[Chi93] Wei-Ngan Chin. Towards an automated tupling strategy. In *3rd ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation, ACM Press*, pages 119–132, Copenhagen, Denmark, ACM Press, June 1993.

[Chi95] Wei-Ngan Chin. Synthesizing efficient parallel programs from sequential specifications. Technical report, (TRA9/95) Dept of IS/CS, NUS, September 1995.

[Col93] M Cole. Parallel programming, list homomorphism and the maximum segment sum problem. CSR 25-93, University of Edinburgh, 1993.

[FG94] AL. Fischer and AM. Ghuloum. Parallelizing complex scans and reductions. In *ACM PLDI*, pages 135–136, Orlando, Florida, ACM Press, 1994.

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