Synchronization Analyses for Multiple Recursion Parameters

Wei-Ngan CHIN and Siau-Cheng KHOO
Dept of Info. Systems & Computer Sc.
National University of Singapore
Singapore 119260

Peter THIEMANN
Wilhelm-Schickard-Institut
University of Tuebingen
Germany D-72076

(Preliminary Report)

Abstract

Tupling is a transformation tactic to obtain new functions, without redundant calls and/or multiple traversals of common inputs. In [Chi93], we presented an automatic method for tupling functions with a single recursion parameter each.

In this paper, we propose a new family of parameter analyses, called *synchronization analyses*, to help extend the tupling method to functions with multiple recursion parameters. To achieve better optimisation, we formulate three different forms of tupling optimisations for the elimination of intra-call traversals, the elimination of inter-call traversals and the elimination of redundant calls. We also guarantee the safety of the extended method by ensuring that its transformation always terminates.

1 Introduction

Tupling is a powerful program transformation technique that is capable of obtaining super-linear speedup. A classic example is the naive *fib* function where the presence of redundant calls cause its definition to have exponential time-complexity.

```
fib(0)      = 1;
fib(1)      = 1;
fib(n+2)    = fib(n+1) + fib(n);
```

The automated tupling transformation method of [Chi93] can be used to optimise this function. Initially, a new tuple function of the following form is introduced.

```
fib_tup(n)  = (fib(n+1), fib(n));
```

This can then be transformed to obtain the following program with linear time-complexity.

```
fib(0)      = 1;
fib(1)      = 1;
fib(n+2)    = let (u,v) = fib_tup(n) in u+v;
fib_tup(0)  = (1,1) ;
fib_tup(n+1) = let (u,v)=fib_tup(n) in (u+v,u) ;
```

One serious restriction of the present tupling technique (as proposed in [Chi93]) is that it can only safely handle functions with a single *recursion parameter* each, such as *fib*. (A recursion parameter is a parameter whose size could progressively decrease via recursive calls of pattern-matching equations. A formal definition is given later.) This limitation excludes many interesting functions with multiple recursion parameters, such as functions *take* and *drop* below:

```
data List a = Nil | Cons(x,xs)
split(n,xs)      = (take(n,xs),drop(n,xs));
take(0,xs)       = Nil;
take(n+1,x:xs)   = x:take(n,xs);
drop(0,xs)       = xs;
drop(n+1,x:xs)   = drop(n,xs);
```

Note that *:* is an infix notation for *Cons*. Both functions *take* and *drop* have two recursion parameters each. Two of their calls, *take(n,xs),drop(n,xs)*, appear in the RHS of function *split* causing the two input parameters *n,xs* to be traversed twice. Such multiple traversals are referred to as *inter-call* traversals because the common recursion variables (e.g. *n,xs*) occur across separate recursive calls (e.g. *take* and *drop*). If tupling transformation could be safely extended, these traversals could be eliminated by transforming the definition of *split* (using unfold/fold rules of [BD77]), as follows:

```
Instantiate n=0
split(0,xs)                = (take(0,xs),drop(0,xs))          ; unfold take,drop calls
                           = (Nil,xs);
Instantiate n=n+1 and xs=x:xs
split(n+1,x:xs)            = (take(n+1,x:xs),drop(n+1,x:xs))  ; unfold take,drop calls
                           = (x:take(n,xs),drop(n,xs))        ; abstract take,drop
                           = let (ys,zs)=(take(n,xs),drop(n,xs))
                                 in (x:ys,zs)                 ; fold split
                           = let (ys,zs)=split(n,xs) in (x:ys,zs)  ;
```

Another example with multiple recursion parameters is function *zip* below.

```
dup(xs)          = zip(xs,xs);
zip(Nil,Nil)     = Nil;
zip(Nil,y:ys)    = Nil;
zip(x:xs,Nil)    = Nil;
zip(x:xs,y:ys)   = (x,y):zip(xs,ys);
```

Note that *zip* is a function with two *list*-type recursion parameters. One of its calls, *zip(xs,xs)*, appear in the RHS of *dup* with two overlapping recursion arguments of common variable, *xs*. Here, the multiple traversals occur within the same *zip* function call, as opposed to separate function calls for the previous example. We refer to such multiple traversals as *intra-call* traversals because the common variable (e.g. *xs*) occur within the same recursive call (e.g. *zip*). The two recursion parameters each consumes a *Cons* cell per recursive cycle around the *zip* function. This synchronized consumption of the two parameters allows the intra-call traversals of *zip(xs,xs)* to be eliminated. We can therefore transform the single *zip* call in the RHS of *dup*, as follows:

```
Instantiate xs=Nil
dup(Nil)         = zip(Nil,Nil)      ; unfold zip call
                 = Nil;
Instantiate xs=x:xs
dup(x:xs)        = zip(x:xs,x:xs)    ; unfold zip call
                 = (x,x):zip(xs,xs)  ; fold dup
                 = (x,x):dup(xs)     ;
```

The main difficulty of extending the tupling method is that not all functions with multiple recursion parameters can be safely transformed. As a counter-example, consider a slightly different *zip2* function:

```
dup2(xs)         = zip2(xs,xs);
zip2(Nil,Nil)    = Nil;
zip2(Nil,y:ys)   = Nil;
```

zip2(x:xs,Nil) = *Nil;*
zip2(x:xs,y:ys) = *zip2'(xs,ys,x);*
zip2'(Nil,ys,x) = *Nil;*
zip2'(x':xs,ys,x) = *(x+x',y):zip2(xs,ys);*

The two mutual recursive functions *zip2* and *zip2'* have two recursion parameters per function. (Though there are three parameters in *zip2'*, only two of them are recursion parameters.) In each cycle around the recursive functions *zip2* $\rightarrow$ *zip2'* $\rightarrow$ *zip2*, two *Cons* cells are consumed by the first recursion parameter, and one *Cons* cell is consumed by the second recursion parameter. Hence, the two recursion parameters do not have intra-call synchronized consumption of their inputs for the recursive cycle. If we apply tupling transformation to *zip2(xs,xs)*, we encounter successively larger recursive calls as follows:

$$zip2(xs,xs) \rightsquigarrow zip2(xs,x1:xs) \rightsquigarrow zip2(xs,x1:x2:xs) \rightsquigarrow zip2(xs,x1:x2:x3:xs) \rightsquigarrow \ldots$$

This can cause the tupling method to go into a loop because an infinite number of different *zip2* calls (often called specialisation/folding points) with overlapping recursion arguments are encountered when we attempt to eliminate their intra-call traversals. Ensuring termination of transformation methods is important if we are to have automatic optimisation techniques that can be safely incorporated into compilers. (Note that symbol $\rightsquigarrow$ represents a sequence of one or more unfolding steps.)

To solve this problem, this paper proposes a new family of parameter analyses, called *synchronization analyses*, for analysing the interactions between multiple recursion parameters. The proposed analyses can help decide in advance whether it is safe to perform tupling on given functions with multiple recursion parameters. With it, we can extend the basic tupling method of [Chi93] to safely handle such functions.

The proposed extension is also an improvement over an earlier proposal made in [CK93], where some syntactic classification of functions with multiple recursion parameters were introduced. A problem with our earlier proposal is that the syntactic-based approach is rather conservative (restrictive). Minor syntactic perturbations can cause the older syntactic-based analysis to fail unnecessarily. For example, the following alternative definition of *zip* is not recognised as safe by the older syntactic-based analysis [CK93], but it can be accepted by the new semantics-based analysis to be proposed in this paper.

zip(Nil,ys) = *Nil;*
zip(x:xs,ys) = *zip'(xs,ys,x);*
zip'(xs,Nil,x) = *Nil;*
zip'(xs,y:ys,x) = *(x,y):zip(xs,ys);*

Section 2 reviews the automated tupling method for functions with a single recursion parameter per function. Section 3 considers functions with multiple recursion parameters, together with a framework for synchronization analyses. Section 4 presents the new tupling method in three phases to perform the eliminations of *intra-call traversals*, *inter-call traversals* and *redundant calls*, respectively. Splitting the tupling method into three phases allows more optimisation to be achieved. Related work and conclusion are discussed in Section 5.

2 Tupling Overview

We consider a first-order functional language[1] defined as follows.

[1]Though our method is formulated for a first-order language, it is quite simple to extend it to a higher-order language. Briefly speaking, we could use the higher-order removal method of [Chi90] to obtain an intermediate higher-order representation with only two additional constructs, namely $v(t^*)$ and $\lambda(v^*) \rightarrow t$. These constructs would then be suitably handled by the tupling method in a manner similar to data constructors.

Definition 1: *First-Order Language*
The simple language is defined by the following context-free grammar:

$$\begin{array}{lll} P & ::= & [M^+] \\ M & ::= & F^+ \\ F & ::= & E^+ \\ E & ::= & f(p^*, v^*) = t \\ t & ::= & v \mid C(t^*) \mid f(t^*) \mid \mathit{let}\ \{p = t\}^+\ \mathit{in}\ t' \mid \bot \\ p & ::= & v \mid C(v^*)\ ; \end{array}$$

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

Each program P is made up of a sequence of one or more mutual-recursive (strongly-connected) sets of functions, M. Each function F is defined by one or more non-overlapping equations, E. Lastly, expressions are made up of variables (v), data constructors (C), functions (f), *let* constructs and a special symbol ($\bot$) for errors. Note that the tuple constructor, $(t_1, \ldots, t_n)$, is regarded as an instance of the more general data constructor, $C(t_1, \ldots, t_n)$.

Each function is defined using a set of complete non-overlapping equations with simple pattern parameters, p^*, and non-pattern parameters, v^*. The use of *simple patterns*, with a single constructor each, does not lose generality. The pattern-matching translation technique of [Aug85] can convert functions with arbitrary nested constructor patterns to those with only simple patterns.

For an example of *complete* non-overlapping equations with simple patterns, we have to rewrite the earlier *take* function as follows:

```
take(0,Nil)     = Nil;
take(0,x:xs)    = Nil;
take(n+1,Nil)   = ⊥;
take(n+1,x:xs)  = x:take(n,xs);
```

We abbreviate a sequence of terms $t_1, \ldots, t_n$ by $\vec{t}$, so that a function call $f(t_1, \ldots, t_n)$ can be abbreviated as $f(\vec{t}\,)$ or more precisely as $f(t_i)_{i \in 1..n}$. Also, a set of equations $\{f(\vec{p_i}, \vec{v_i}) = e_i\}_{i \in N}$ can be abbreviated as $f \stackrel{\text{def}}{=} ?\{(\vec{p_i}, \vec{v_i}) \Rightarrow e_i\}_{i \in N}$. To mark the fact that $f \in F$, where F is an arbitrary set of functions, we sometimes annotate its calls using $f^F(\vec{t}\,)$.

We introduce a special context notation with multiple holes.

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

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

$$\widehat{e}\langle\rangle^F ::= v \mid \pm m \mid C(\widehat{e_i}\langle\rangle^F)_{i \in 1..n} \mid f(\widehat{e_i}\langle\rangle^F)_{i \in 1..n} \mid \mathit{let}\ \{p_i = \widehat{e_i}\langle\rangle^F\}_{i \in 1..n}\ \mathit{in}\ \widehat{e}\langle\rangle^F$$
$$\mathit{such\ that}\ f \notin F$$

Note that the context notation is parameterised by a set of functions, F, whose calls must not appear in the context itself. Each expression e can be decomposed into a context $\widehat{e}\langle\rangle^F$ and a sequence of sub-terms $[t]_{i \in 1..n}$ using the notation $\widehat{e}\langle t_i\rangle^F_{i \in 1..n}$. This is equivalent to $[t_i/\pm i]_{i \in 1..n}(\widehat{e}\langle\rangle^F_{i \in 1..n})$ which stands for the substitutions of sub-terms, $t_1, \ldots, t_n$, into their respective holes, $\pm 1, .., \pm n$, for context $\widehat{e}\langle\rangle^F$.

With these notations, we select (pick) all calls of F from an expression e, by:

$$e = \widehat{e}\langle f_i^F(\vec{ti})\rangle^F_{i \in 1..n}$$

For example, to select the two *take/drop* calls in the expression *(x:take(n,xs),drop(n,xs))*, we use $(x : \widehat{\pm 1}, \pm 2)\langle \mathit{take}(n, xs), \mathit{drop}(n, xs)\rangle^{\{\mathit{take}, \mathit{drop}\}}$.

Some common predicates on expressions that are used later include:

Definition 3: *Useful Predicates*
The predicate $(t_1 \sqsubseteq t_2)$ is to test if t_1 is a sub-term of t_2, while the proper sub-term relationship is denoted by $(t_1 \sqsubset t_2)$.

The predicate $IsVar(t)$ returns true if t is a variable; false otherwise.

The predicate $IsConst(t)$ returns true if t is a constant expression without any free variables; false otherwise.

2.1 Transformation for SRP-Functions

The tupling method in [Chi93] was formulated for a class of functions with a *single recursion parameter* each, called SRP-functions.

Definition 4: *SRP-Functions*
Consider:

$$f^M(p, \vec{v}) \qquad = \widehat{e_f}\langle f_j^M(tj_0, \vec{tj})\rangle^M_{j \in 1..m} \ ;$$

This equation is said to be a SRP-equation if the single pattern-parameter, p, is a *recursion* parameter. The pattern-parameter, p, is said to be a *recursion* parameter if the recursion argument, tj_0, for each mutual recursive call, $f_j^M(tj_0, \vec{tj})$, is a variable taken from p, as follows:

$$\forall j \in \{1..m\}.\, RP\_Cond(p, tj_0)$$
$$\text{where } RP\_Cond(p, t) \equiv IsVar(t) \wedge (t \sqsubseteq p)$$

A function f^M is said to be a SRP-function if all the equations of its M-set are SRP-equations.

Two simple SRP-functions are *deepest* and *depth*.

```
data Tree a = Leaf a | (Tree a) @ (Tree a) ;
deepest(Leaf(a))  = [a];
deepest(l@r)      = if depth(l) > depth(r) then deepest(l)
                    else if depth(l) < depth(r)  then deepest(r)
                    else deepest(l)++deepest(r);
depth(Leaf(a))    = 0 ;
depth(l@r)        = 1+max(depth(l),depth(r)) ;
```

Redundant calls are present in the definition of *deepest*, causing the function to have a time complexity of $O(n^2)$ where n is the size of its input tree. By eliminating the redundant calls, it is possible to reduce the function's time complexity to $O(n)$. An informal tupling algorithm to achieve such an optimisation is given below. (A formal presentation shall be given in the full paper.)

Method 1: Tupling Algorithm, $\mathcal{T}$

Step 0 Decide the set of function calls to tuple, F.

Step 1 Repeatedly *unfold (without instantiation)* each function call of F.

Step 2 Split the F calls to separate tuples based on their sole recursion argument. For each tuple:

Step 3 If the tuple has only one function call, then *terminate*.

Step 4 If the tuple has appeared before, then *fold* against previous definition and *terminate*.

Step 5 Otherwise, introduce a new tuple function definition. This is a *define* step.

Step 6 *Unfold (with instantiation)* any one call of F. Goto Step 1.

There are two types of unfold operations in the above tupling algorithm, namely:

- Unfolds *without* Instantiation (Step 1)
- Unfolds *with* Instantiation (Step 6)

These unfold operations assume that the functions are defined using pattern-matching equations. For example, the call *deepest(l@r)* can be directly unfolded without any instantiation, while the call *deepest(t)* requires its argument t to be instantiated to either *Leaf(a)* or *l@r* before it can be unfolded. As there could be infinitely many different instantiations for recursive types, we shall always apply the *minimal instantiation* needed for unfolding some function call(s).

Notice that Step 6 may involve a non-deterministic choice of F calls to instantiate and unfold. However, it is always followed by Step 1 which repeatedly unfolds those F calls which did not require instantiation. This combination of the two steps ensures a deterministic outcome.

To transform the earlier example, we fix F={*deepest,depth*}, and then apply $\mathcal{T}$ to each RHS of their equations. No significant change occur in all equations, except for the second equation of *deepest*. This equation can be written more concisely (using a suitable context $\widehat{e}\langle\rangle$) as:

$$deepest(l@r) \quad = \widehat{e}\langle depth(l), deepest(l), depth(r), deepest(r)\rangle$$

Following the transformation steps in Figure 1, the above equation is transformed to:

$$\begin{array}{ll} deepest(l@r) & = let\ \{(u,v)=d\_tup(l);\ (a,b)=d\_tup(r)\}\ in\ \widehat{e}\langle u, v, a, b\rangle \\ d\_tup(Leaf(a)) & = (1,[a]) \\ d\_tup(l@r) & = let\ \{(u,v)=d\_tup(l);\ (a,b)=d\_tup(r)\}\ in\ (1+max(u,a),\widehat{e}\langle u, v, a, b\rangle) \end{array}$$

2.2 Ensuring Termination

The above tupling algorithm can eliminate both redundant calls and multiple traversals. However, it is possible for the algorithm to be non-terminating for the following reasons:

- Step 1 may unfold (without instantiation) function calls *indefinitely*.
- Step 5 may define new tuple function via *infinitely* many different tuples.

Applying $\mathcal{T}^{\{depth,deepest\}}$

deepest(l@r) $= \widehat{e}\langle depth(l), deepest(l), depth(r), deepest(r)\rangle$

Step 2: Gather Calls (according to different recursion arg.)

$=$ *let* {(u,v)=(depth(l),deepest(l)); (a,b)=(depth(r),deepest(r))} *in* $\widehat{e}\langle u, v, a, b\rangle$

Step 5: Define Tuple Function

$=$ *let* {(u,v)=d_tup(l); (a,b)=d_tup(r)} *in* $\widehat{e}\langle u, v, a, b\rangle$

Define

d_tup(t) $=$ (depth(t),deepest(t))

Step 6: Instantiate and unfold depth call

Case t=Leaf(a)

d_tup(Leaf(a)) $=$ (1,deepest(Leaf(a)))

Step 1: Unfold deepest call (without instantiation)

$=$ (1,[a]) ;terminated *by Step 3*

Case t=l@r

d_tup(l@r) $=$ (1+max(depth(l),depth(r)),deepest(l@r))

Step 1: Unfold deepest call (without instantiation)

$=$ (1+max(depth(l),depth(r)),$\widehat{e}\langle depth(l), deepest(l), depth(r), deepest(r)\rangle$)

Step 2: Gather Calls

$=$ *let* {(u,v)=(depth(l),deepest(l)); (a,b)=(depth(r),deepest(r))} *in* (1+max(u,a),$\widehat{e}\langle u, v, a, b\rangle$)

Step 4: Fold with d_tup

$=$ *let* {(u,v)=d_tup(l); (a,b)=d_tup(r)} *in* (1+max(u,a),$\widehat{e}\langle u, v, a, b\rangle$)

Figure 1: An Example of Tupling Transformation

These two potential causes of non-termination can be resolved by two further restrictions on the sub-class of SRP-functions, called the *descending-RP* and the *bounded-arguments* restrictions.

The *descending-RP* restriction can guarantee that there will never be infinite number of unfolds without instantiation. It achieve this by requiring that the mutual-recursive definition do *not* have cyclic caller-callee transitions with non-descending recursion parameter.

Definition 5: *Descending-RP Restriction*
A M-set of SRP-functions, $f_1, .., f_n$ is said to satisfy *descending-RP* restriction if its recursion parameter will be decreased by at least one constructor per cycle around the recursive functions.

To detect this property, we build a call-graph where each caller-callee transition of $\{f_1, .., f_n\}$ is labelled as either *descending* or *unchanged*, depending on whether the recursion parameter is decreased or unchanged across its call transition[a]. The *descending-RP* restriction is satisfied iff every cycle of call transitions has at least one *descending* transition.

[a] There is no need to consider *increasing* transitions since these are disallowed by the SRP-form.

An example which violates the descending-RP restriction is:

$f(xs,y) = \widehat{e}_f\langle f(xs,y), g(xs,y)\rangle^M;$
$g(Cons(x,xs),y) = \widehat{e}_g\langle f(xs,1), g(xs,y), g(xs,2)\rangle^M;$

Note the presence of a cyclic transition $f(xs,y) \overset{\text{unchanged}}{\rightarrow} f(xs,y)$ where the recursion parameter is not decreased at all. If we apply $\mathcal{T}$ to the RHS of f, we would encounter a non-terminating

derivation involving repeated unfolding of *f(xs,y)*, without the need for parameter instantiation. All SRP-functions with the descending RP-restriction do not suffer from this shortcoming.

The bounded-arguments restriction can prevent infinite number of define steps during tupling. This is achieved by ensuring that there is a bounded number of different tuple function definitions (specialisation/folding points).

Definition 6: *Bounded-Arguments Restriction*
Consider:

$$f(p, \vec{v}) \qquad = \widehat{e_f}\langle f_j^M(tj_0, \vec{tj})\rangle_{j \in 1..m}^M \ ;$$

This equation is said to have the *bounded-arguments* restriction if the non-recursion arguments, $\vec{tj}$, from each SRP-call, $f_j^M(tj_0, \vec{tj})$, are either constants or are variables from the original set of non-recursion parameters, $\vec{v}$, as follows:

$$\forall\, j \in \{1..m\}.\, \forall\, t \in \{\vec{tj}\}.\, IsConst(t) \vee (t \in \{\vec{v}\})$$

A SRP-function is said to adhere to the *bounded-arguments* restriction if all the equations of its M-set of functions are *bounded-arguments* SRP-equations.

Using SRP-functions with the bounded-arguments restriction, we can guarantee that the number of different new tuple definitions introduced is bounded. Firstly, there is a finite number of different functions. Secondly, each tuple has only one recursion variable. Also, the number of different variables for the non-recursion arguments is finite, as no new variables are introduced by unfolding. Thus, if there are n different F-functions and the largest number of non-recursion parameters is m and the number of different variables and constants used (from the original program) is s, then the maximum number of distinct calls which could be obtained is $n \times s^m$. As the number of distinct calls is bounded, the number of different tuples (modulo variable renaming) encountered will also be bounded (at most $2^{n \times s^m}$). Hence, there will always be a bounded number of new tuple function definitions (Step 5).

These two restrictions may appear quite severe. Fortunately, it is possible to use various pre-processing techniques to transform some functions outside these restrictions to equivalent functions within (see [Chi95]).

3 Synchronization Analyses

Functions with multiple recursion parameters cause additional non-termination problems for the tupling method. Apart from the descending-RP and bounded-arguments restrictions, it is also required that multiple recursion parameters be properly synchronized across successive recursive calls. We can extend the sub-class of SRP-functions to encompass functions with multiple recursion parameters, as follows:

Definition 7: *MRP-Functions*
Consider an equation with r pattern-parameters:

$$f^M(p_1, \ldots, p_r, \vec{v}) \qquad = \widehat{e_f}\langle f_i^M(ti_1, \ldots, ti_r, \vec{ti})\rangle_{i \in 1..m}^M \ ;$$

This equation is said to be a MRP-equation if all the r pattern-parameters are *recursion* parameters. This condition is expressed as follows:

$$\forall i \in 1..m.\, \forall j \in 1..r.\, RP\_Cond(p_j, ti_j)$$

A function f^M is said to be a MRP-function if all the equations of its M-set are MRP-equations.

To handle MRP-functions safely, we shall introduce a family of synchronization analyses to determine if their multiple recursion parameters are suitable for either the eliminations of intra-call traversals, inter-call traversals or redundant calls. Our synchronization analyses are based on the function call transitions between MRP-functions. We define:

Definition 8: *Labelled Call Graph*
The *labelled call graph* of a set of MRP-function definitions, M, is a graph whereby each function name from M is a *node*; and each caller-callee transition is represented by an *arrow*, labelled with the *descent* operation on the recursion parameters.

Consider each MRP-equation with multiple recursion parameters $\vec{p}$ and descent operators, $\vec{d1}$ to $\vec{dm}$, for its m mutual recursive calls of form $f_i^M(di\vec{(p)}, \vec{ti})$.

$$f^M(\vec{p}, \vec{v}) \;=\; \widehat{e_f}\langle f_i^M(di\vec{(p)}, \vec{ti})\rangle_{i \in 1..m}^{M} \;;$$

To build the graph, we would introduce m *caller-callee* transitions, $\{f \stackrel{(\vec{di})}{\rightarrow} f_i\}_{i \in 1..m}$, labelled by the respective *descent operation* $(\vec{di})$ in tuple form.

The descent operators for each recursion argument are represented as follows. If the LHS pattern parameter is v and the corresponding argument is v, the identity function *ID* is used as the descent operator since $ID(v) = v$. If the LHS pattern parameter is $c(v_1, \ldots, v_j)$ and its argument is $v_i \in \{v_1, .., v_j\}$, the descent operator $c[i]$ is used where $c[i](c(v_1, \ldots, v_j)) = v_i$.

As an example, the labelled call graphs for the two definitions of the *zip* function (from Section 1) are given in Figure 2. Note that the call transition *zip(x:xs,y:ys)* $\rightarrow$ *zip(xs,ys)* is being labelled with the descent operation *(:[2],:[2])* where *:[2]* is a descent operator to return the second argument of a *Cons* cell.

Call transitions can be composed as follows:

Definition 9: *Composing Call Transitions*
Given two adjacent transitions $f \stackrel{D_1}{\rightarrow} g$ and $g \stackrel{D_2}{\rightarrow} h$, we can compose it as follows:

$$(g \stackrel{D_2}{\rightarrow} h)\, o\, (f \stackrel{D_1}{\rightarrow} g) = f \stackrel{D_2\, o D_1}{\rightarrow} h.$$

To express composition in a left to right manner, we use an alternative notation:

$$(f \stackrel{D_1}{\rightarrow} g); (g \stackrel{D_2}{\rightarrow} h) = f \stackrel{D_1; D_2}{\rightarrow} h.$$

Transitions are often represented by just their descent operations, e.g. $D_1; D_2$ or $D_2\, o D_1$. They are associative and can be composed as a sequence of transitions.

We define transitions for both single and multiple recursion parameters.

Definition 10: *RP- and MRP-Transitions*
The transition for a single recursion parameter (e.g. *:[2]*) is known as a *RP-transition*. It is denoted using $d, d1, d2, ..$ or $e, e1, e2, ...$ A sequence of RP-transitions (e.g. $d1; ..; dn$) is denoted using $ds, ds1, ds2, ..$ or $es, es1, es2, ...$

A tuple of RP-transitions (e.g. *(:[2],:[2])*) is known as a *MRP-transition*. It is denoted using $D, D1, D2, ..$ or $E, E1, E2, ..$ A sequence of MRP-transitions (e.g. $D1; ..; Dn$) is denoted using $DS, DS1, DS2, ..$ or $ES, ES1, ES2, ...$

Given a MRP-transition sequence, we can obtain a tuple of RP-transition sequences for the individual recursion parameters via:

$$(d1_1, .., d1_n); ..; (dr_1, .., dr_n) = ((d1_1; ..; dr_1), ..., (d1_n; ..; dr_n))$$

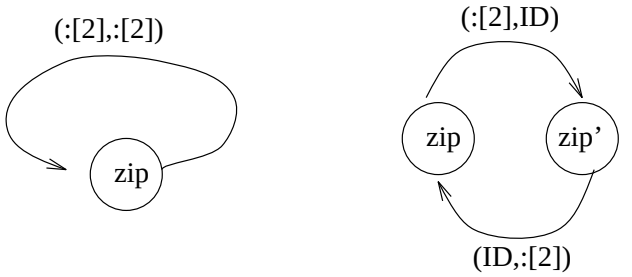

Figure 2: Labelled call graphs for two versions of the *zip* function

Transition sequences can be compacted (simplified) as follows:

Definition 11: *Compacted Transitions*
Each RP-transition sequence can be compacted using two laws, namely (i) $d; ID = d$ and (ii) $ID; d = d$.

For each RP-sequence, *ds*, we denote its compacted transition sequence by *compact(ds)*. MRP-transition sequence can be simplified by compacting its individual RP-sequences.

To analyse for parameter synchronization around recursive functions, we focus on transition sequences which are cyclic.

Definition 12: *Cyclic Transitions*
A sequence of transitions, $f1 \stackrel{D_1}{\rightarrow} f2 \stackrel{D_2}{\rightarrow} .. \stackrel{D_{n-1}}{\rightarrow} fn \stackrel{D_n}{\rightarrow} f1$, which starts from a function *f1* and returns back to the same function is called a *cyclic* transition sequence. The composed version can be abbreviated as $f1 \stackrel{D_1;..;D_n}{\rightarrow} f1$.

The cyclic transition sequences can be used to analyse how much of a recursion argument is being consumed by each cycle of recursive calls. For example, the cyclic transition $zip \rightarrow zip$ (in Figure 2) has the label *(:[2],:[2])* which shows that a *Cons* cell is being consumed by each of the two recursion parameters. Similarly, the call transitions for $zip \rightarrow zip'$ and $zip' \rightarrow zip$ have labels *(:[2],ID)* and *(ID,:[2])*. The first transition states that a *Cons* cell will be consumed (and then replaced by its second argument) for the first parameter but is unchanged for the second parameter. The second transition states the opposite situation. When both call transitions are composed, we get a cyclic transition $zip \stackrel{(:[2],ID);(ID,:[2])}{\rightarrow} zip$ which can be compacted to *(:[2],:[2])*. Hence, both definitions of the *zip* function have the same (compacted) cyclic transition sequence despite their syntactic differences.

For each cycle around the recursive function calls, the recursion parameter must have its size decreased by at least one. This can be guaranteed by ensuring that all cyclic MRP-transition sequences are *descending* sequences, defined as follows:

Definition 13: *Descending RP and MRP Transitions*
A RP-transition sequence *ds* is said to be *descending* if $compact(ds) \neq ID$.

A MRP-transition sequence $(\vec{ds})$ is said to be *descending* if
$$\exists ds \in \{\vec{ds}\}.compact(ds) \neq ID$$
Correspondingly, a M-set of MRP-functions is said to satisfy the *descending-MRP restriction* if every of its cyclic MRP-transition sequences is descending.

We must appropriately handle function calls with either identical, overlapping or disjoint multiple recursion arguments.

Definition 14: *Identical, Overlapping and Disjoint-MRPs*
Two function calls are said to have *identical-MRP* iff every pair of corresponding recursion arguments are identical.

Two function calls are said to have *overlapping-MRP* iff every pair of corresponding recursion arguments have at least one common variable.

Two function calls are said to have *disjoint-MRP* if at least one pair of corresponding recursion arguments does not have any common variable.

Consider two functions *f1, f2* with two recursion parameters each. Based on the above definitions, the calls {*f1(x1,y1),f2(x1,y1)*} have identical-MRP, while the calls {*f1(c(x1,x2),y1),f2(x2,y1)*} have overlapping-MRP, but the calls {*f1(x1,y1),f2(x2,y1)*} have disjoint-MRP.

To analyse transition sequences, we introduce the following notions of synchronization.

Definition 15: *Synchronization of Call Transitions*
Two compacted RP-transition sequences, ds and es, are said to be *level 1-synchronized* (denoted by $ds \approx_1 es$) if

$$\exists ds', es'.\ (ds; ds' = es; es')$$

Otherwise, they are said to be *incompatible* or *level 0-synchronized.*

Two compacted RP-transition sequences, ds and es, are said to be *level 2-synchronized* (denoted by $ds \approx_2 es$) if

$$\exists ds', es'.\ (ds; ds' = es; es') \wedge (ds' = (I\vec{D}) \vee es' = (I\vec{D}))$$

Two compacted RP-transition sequences, ds and es, are said to be *level 3-synchronized* (denoted by $ds \approx_3 es$) if

$$\exists cs, n, m.\ (ds = cs^n) \wedge (es = cs^m) \wedge (n, m > 0)$$

Note that cs is referred to as a *common generator* of the two RP-sequences. Level-3 synchronization is also known as *loose-synchronization.*

Two compacted RP-transition sequences, ds and es, are said to be *level 4-synchronized* (denoted by $ds \approx_4 es$) if $ds = es$. Level-4 synchronization is also be known as *tight-synchronization.*

Corresponding notions of *synchronization* apply to compacted MRP-transition sequences. Note that the synchronizations at levels 1 to 4 form a strict hierarchy, with synchronization at level i implying synchronization at level j if $i > j$. The different notions of synchronization can be used in the following way. Given two sets of calls with identical-MRP. If the two calls follow incompatible transition sequences, then their resulting calls will end up with disjoint-MRPs. If the two calls follow transition sequences that are level 4-synchronized, then the resulting calls will have identical-MRP. For level-3 synchronized transitions, the calls will have identical-MRP after traversing every sub-sequence of the common generator. With level 1-synchronized or 2-synchronized sequences, their calls will have overlapping-MRP.

4 Three-Phases of Tupling

We now present the tupling method for MRP-functions. In order to obtain better optimisation, we separate this method into three phases with different objectives, namely:

- Phase 1 - Elimination of Intra-Call Traversals

- Phase 2 - Elimination of Redundant Calls
- Phase 3 - Elimination of Inter-Call Traversals

The three types of eliminations have slightly different (synchronization) requirements. If they were lumped together into a single algorithm, we would need to impose more conservative restrictions to ensure termination. A better result can therefore be obtained if the tupling method were separated into three phases. Each of the three tupling phases will have:

1. a variant tupling algorithm, and
2. its own safe class of transformable functions/calls.

These tupling phases are applied in the above sequence to each set of mutual-recursive functions according to the bottom-up order of the function calling hierarchy. Given mutual-recursive sets of functions, $[M_1, M_2, .., M_n]$, in bottom-up order. We subject a set of function definitions, M_i, to each of the above three phases before proceeding with M_{i+1}. We call this decomposed method, the *three-phase tupling* method.

Phases 1 and 3 are applied to appropriate *auxiliary* recursive calls, M_j, in the definition of M_i where $j < i$. Phase 1 is used to eliminate intra-call traversals for certain function calls with common (or overlapping) recursion arguments. Phase 3 is used to eliminate inter-call traversals among separate auxiliary recursive calls with overlapping-MRPs. Phase 2 is to eliminate redundant *mutual* (and *auxiliary*) recursive function calls. Details of the three phases are described in the ensuing sub-sections.

4.1 Eliminating Intra-Call Traversals

This phase is meant solely for the elimination of intra-call multiple traversals. In order to formulate a sub-class of functions which can be safely transformed in this phase, we introduce the following enhanced form of synchronization:

> **Definition 16:** *Rotate Synchronization*
> Two compacted RP cyclic transition sequences, ds, es, are said to *rotate-synchronize* at level-n (denoted by $ds \approx_{n,rot} es$) if:
> $$\exists ds_1, ds_2, es_1, es_2 . (ds_1; ds_2 = ds) \wedge (es_1; es_2 = es) \wedge (ds_2; ds_1 \approx_n es_2; es_1)$$

The notion of *rotate-synchronization* is to cater to function calls which start at different points of the cyclic transition sequences and yet may synchronise. With this notion, we define the sub-class of functions which are safe for Phase 1 tupling to be:

> **Definition 17:** *Phase 1 Functions*
> A M-set of mutual-recursive functions is said to be *Phase 1* (or *P1*) functions if they satisfy the *descending-MRP* restriction and share at least one sub-set (R) of two or more tightly-synchronizing parameters that are *rotate-synchronized at level 4* for every cyclic MRP-transition sequences, ($\vec{ds}$), as specified by:
> $$\forall i, j \in R . (ds_i \approx_{4,rot} ds_j)$$

Such a property would allow each cycle of recursive functions to consume exactly the same number of constructors from each sub-set R of TS recursion parameters. As a result, a common variable placed among such sub-sets of recursion parameters might re-occur (synchronize) after each cycle of unfolding a recursive function call.

The elimination of intra-call traversals is guaranteed to be safe (terminating) for each P1-function call (or their intermediate) with common variables among its subset of TS parameters. An informal procedure is outlined below.

Method 2: Tupling to Eliminate Intra-Call Traversals

Step 0 Decide the set of safe P1-functions to tuple, F.

Step 1 Repeatedly *unfold without instantiation* all P1-function calls of F.

Step 2 Gather each P1-function call into a single-call tuple. (This is an *abstraction* step.) For each P1-function call:

Step 3 If the P1-call does not have any overlapping TS-recursion argument, *terminate*.

Step 4 If the P1-call has appeared before, then *fold* against previous definition and *terminate*.

Step 5 Otherwise, generalise the non-TS arguments before introducing a new function definition. This is a *define* step.

Step 6 *Unfold* (with minimal instantiations) the P1-function call. Goto Step 1.

The above algorithm allows each argument from TS parameters to remain in-tact but generalise away the other non-TS arguments at each define step. Also, each tuple consists of only a single call with a subset of common (overlapping) TS-arguments. These two tasks are in line with the aim of eliminating intra-call traversals among the TS-arguments. To illustrate the tupling procedure, consider the following set of mutual-recursive P1-functions.

$$
\begin{array}{ll}
f1(x1@x2,ys,z1@z2) & = \widehat{e_{f1}}\langle f2(x2, ys, z2)\rangle \\
f2(x1@x2,ys,z1@z2) & = \widehat{e_{f2}}\langle f3(x1, ys, z2)\rangle \\
f3(xs,y1@y2,z1@z2) & = \widehat{e_{f3}}\langle f4(xs, y1, z2), f2(xs, y1, z1)\rangle \\
f4(xs,y1@y2,zs) & = \widehat{e_{f4}}\langle f1(xs, y2, zs)\rangle
\end{array}
$$

There are two cyclic MRP-sequences, namely $f1 \rightarrow f2 \rightarrow f3 \rightarrow f4 \rightarrow f1$ with transition sequence $(@[2], ID, @[2]); (@[1], ID, @[2]); (ID, @[1], @[2]); (ID, @[2], ID)$ and cycle $f2 \rightarrow f3 \rightarrow f2$ with transition sequence $(@[1], ID, @[2]); (ID, @[1], @[1])$. The compacted representation of these two cycles are $((@[2]; @[1]), (@[1]; @[2]), (@[2]; @[2]; @[2]))$ and $(@[1], @[1], (@[2]; @[1]))$. We can see that the individual transitions of the first and second parameters are rotate-synchronized at level 4 but not the third parameter. This is so because $@[2]; @[1] \approx_{4,rot} @[1]; @[2]$ and $@[1] \approx_{4,rot} @[1]$ holds for the two cycles.

Hence, the first two parameters of this M-set of functions could *potentially* allow the elimination of intra-call traversals. As an example consider:

$$
\begin{array}{ll}
main(xs) & = \widehat{e_{main}}\langle f1(xs, xs, zs), m(xs, xs, zs)\rangle \\
m(xs,y1@y2,zs) & = \widehat{e_{m}}\langle f1(xs, y2, zs)\rangle
\end{array}
$$

In the RHS of *main*, there are two auxiliary calls with common TS recursion arguments where m is considered to be an intermediate function to the $\{f1,f2,f3,f4\}$ M-set. Both calls are safe to transform. The Phase 1 tupling algorithm would define:

$$
\begin{array}{ll}
am(xs,zs) & = m(xs,xs,zs) \\
n1(xs,zs) & = f1(xs,xs,zs)
\end{array}
$$

followed by transformation to obtain:

$$
main(xs) = \widehat{e_{main}}\langle n1(xs, zs), am(xs, zs)\rangle
$$

$n1(x1@x2,z1@z2) = \widehat{e_{f1}}\langle n2(x2,z2,x1)\rangle$
$n2(y1@y2,z1@z2,x1) = \widehat{e_{f2}}\langle n3(y1,z2,x1,y2)\rangle$
$n3(y1,z1@z2,x1,y2) = \widehat{e_{f3}}\langle f4(y1,x1,z2), f2(y1,x1,z1)\rangle$
$am(x1@x2,zs,x1) = \widehat{e_m}\langle a1(x2,zs,x1)\rangle$
$a1(x2,z1@z2,x1) = \widehat{e_{f1}}\langle a2(x2,z2)\rangle$
$a2(x1@x2,z1@z2) = \widehat{e_{f2}}\langle a3(x1,z2,x2)\rangle$
$a3(x1,z1@z2,x2) = \widehat{e_{f3}}\langle a4(x1,z2), a2(x1,z1)\rangle$
$a4(x1@x2,zs) = \widehat{e_{f4}}\langle a1(x2,zs)\rangle$

Note that the P1-class of functions only guarantees *safe* but not necessarily *effective* elimination of intra-call traversals. In the above example, the intra-call traversals of call *m(xs,xs,zs)* is effectively eliminated for all the recursive calls. However, the intra-call traversals of call *f1(xs,xs,zs)* did not actually synchronise across the recursive calls since *f2* and *f4* calls are still present in the RHS of *n3*. This is so because the rotate-synchronization at level-4 has resulted in calls with *disjoint* (instead of *overlapping*) TS recursion arguments. Hence, the *actual* synchronization at level-4 did not materialise for the *f1(xs,xs,zs)* call. Nevertheless, both P1-calls are safe to transform.

Another point worth noting is that our procedure requires that all non-TS arguments be generalised. As an example, consider the call *f2(xs,xs,xs)* with three identical recursion arguments. As only the first two arguments are tightly-synchronized (via $\approx_{4,rot}$), the procedure requires that the last argument be generalised (during Phase 1 tupling) to:

let zs=xs in f2(xs,xs,zs)

This generalisation can help avoid potential non-terminating due to infinite specialisation points (ie. define steps). An example of such non-termination is the *zip2* function given in Section 1.

4.1.1 Termination Proof

There are two potential causes of non-termination.

- Infinite number of unfolding without instantiation (Step 1).
- Infinite number of define steps (Step 5).

Infinite unfolding without instantiation (Step 1) is not possible because the P1-functions have the *decreasing-MRP* restriction. Infinite define of Step 5 is also impossible because there are only a finite number of different single call tuples. Firstly, the non-TS arguments are always generalised into variables. Secondly, the size of the TS-arguments are bounded. On every cycle, each TS-parameter consumes exactly the same number of constructors from each of its recursion argument. Hence, the size of each TS-argument during the transition would grow by at most the length of the cyclic MRP-sequence before it is decreased by a corresponding amount in the remaining transitions. As there is a finite number of function names, the number of different tuples which could be encountered is therefore bounded. Hence, infinite number of define steps is not possible since fold transformation (Step 4) will be carried out when a matching call re-appears.

4.1.2 Preservation of Non-Strict Semantics

It is quite easy to show that this tupling phase to eliminate intra-call traversals preserves the lazy semantics. In particular, the instantiation of recursion parameters occur for only new equations with a single call on their RHS. Since such a single call lies in a strict context, its parameter instantiation always adheres to the non-strict semantics (see [RFJ89]).

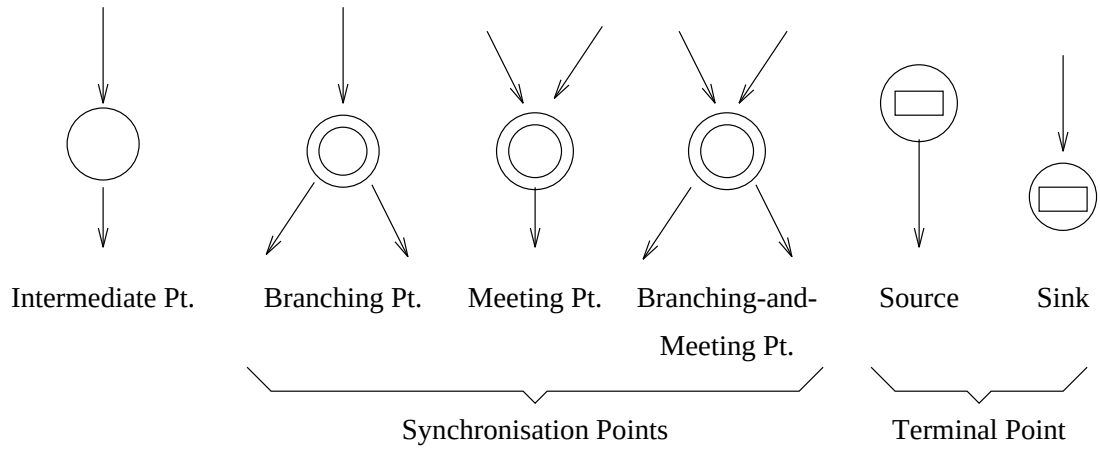

Figure 3: Classification of nodes in a labelled call graph

Though preservation of non-strict semantics is simple for this phase, it is not so straightforward for the next two tupling phases. The reason is that the next two phases have to deal with tuples of MRP-function calls. Such calls may have conflicting strictness requirement on the common recursion argument. Under this scenario, an unfold (with instantiation) on one call may indirectly cause another call to violate its non-strict semantics making the latter more strict than should be. We shall consider remedies for preserving non-strict semantics in the full paper.

4.2 Synchronization Points

Before describing the analyses and techniques used to eliminate redundant calls or inter-call traversals, we give an enhanced interpretation of the labelled call graph, defined in Definition 8.

We classify the nodes (ie., functions) in the labelled call graph based on the number of its in-coming and out-going (directed) edges. An *intermediate point* is a node that has exactly one in-coming edge and one out-going edge. A *branching point* has one in-coming edge and multiple out-going edges. A *meeting point* has multiple in-coming edges and one out-going edge. A *branching-and-meeting point* has multiple in-coming and multiple out-going edges. Such branching and/or meeting points are collectively known as *synchronization points*. Lastly, a *terminal point* is either a sink or a source in the graph. Figure 3 illustrate such a classification.

Synchronization points are potential places where redundant calls within an equation can be detected, and thus effective tupling can take place. In the case of mutual-recursive function definitions, effective tupling are made possible when synchronization (of arguments) can be maintained across recursive function calls. To describe this, we define a *segment* as a sequence of connected edges that link a synchronization point to its neighbouring synchronization point (possibly itself) without visiting any other synchronization points.

Sometimes, an intermediate point in a graph can be introduced as a *pseudo*-synchronization point, which breaks a segment (passing through it) into two sub-segments; this may enable synchronization to be attained independently at each sub-segment. This is useful when the segment itself is not suitably synchronized with any other segment in the graph.

With this classification of nodes, we can easily identify the set of functions eligible for redundant call elimination. This is described in the next section.

4.3 Eliminating Redundant Calls

In this section, we describe the task of redundant call elimination. Firstly, we characterise a sub-class of functions in which redundant calls can be safely eliminated.

Definition 18: *Phase 2 MRP-Functions*
A M-set of MRP-functions is said to be *Phase 2* (or P2) function if the following conditions are satisfied:

1. Every function satisfies the descending-MRP restriction.
2. The non-recursion parameters satisfy the bounded-arguments restriction.
3. Let $\mathcal{B}$ be the set of all the segments in the corresponding labelled call graph. Then $Sync(\mathcal{B})$ holds, where

$$Sync(\mathcal{B}) \equiv \forall i,j \in \mathcal{B}.\ (i \approx_0 j)\ \vee\ (i \approx_3 j)\ \vee\ PseudoSyn(i,j,\mathcal{B}),$$

$$\begin{aligned} PseudoSyn(i,j,\mathcal{B})\ \equiv\ & i \not\approx_3 j\ \wedge\ i \approx_2 j\ \wedge \\ & ((\exists i'.\ (i = (j;i'))\ \wedge\ Sync((\mathcal{B} - \{i\}) \cup \{i'\}))\ \vee \\ & (\exists j'.\ (j = (i;j'))\ \wedge Sync((\mathcal{B} - \{j\}) \cup \{j'\}))). \end{aligned}$$

Conditions 1 and 2 help ensure the termination of tupling process. Condition 3 ensures synchronization of the redundant calls. In contrast to the the tight-synchronization criteria needed among arguments in the case of Phase-1 tupling, loose synchronization (*ie.*, synchronization at level-3) between calls suffices for achieving redundant call elimination. In the case where two segments are synchronized at level-2 but not level-3, a pseudo-synchronization point (for the longer segment) has to be introduced, so that the opportunity for synchronization can be obtained at the two sub-segments. Thus, tranforming a tuple of any calls with overlapping-MRPs is bound to terminate because of the loose-synchronization criteria. (This will be elaborated in the termination proof below.)

Once a M-set of P2 functions is identified, we initiate the transformation by tupling those calls that are located within the same equation and having identical (or overlapping) arguments. If these calls have synchronized cyclic transitions, tupling becomes effective.

The algorithm for this tupling phase is outlined below:

Method 3: Tupling to Eliminate Redundant Calls

Step 0 Decide a set of mutual-recursive P2-function calls to tuple, F.

Step 1 Repeatedly *unfold without instantiation* each function call of F.

Step 2 Gather each subset of P2-function calls with overlapping-MRP into a tuple. (This is an *abstraction* step.) For each tuple:

Step 3 If the tuple has ≤ 1 function call, then *terminate*.

Step 4 If the tuple has appeared before, then *fold* against previous definition and *terminate*.

Step 5 Otherwise, introduce a new function definition. This is a *define* step.

Step 6 *Unfold* (with minimal instantiation) a P2-call. Goto Step 1.

As an example, consider the following contrived set of functions.

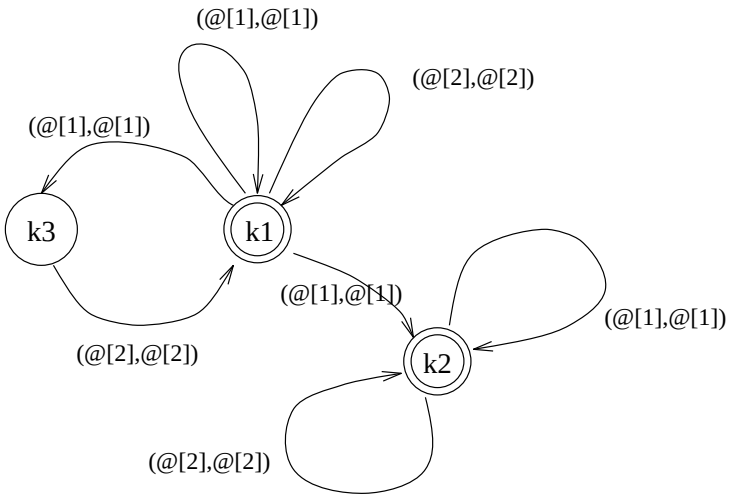

Figure 4: A Labelled Call Graph for *k1*, *k2* and *k3*.

$$k1(x1@x2,y1@y2) = \widehat{e_{k1}}\langle k1(x1,y1), k2(x1,y1), k3(x1,y1), k1(x2,y2)\rangle$$
$$k2(x1@x2,y1@y2) = \widehat{e_{k2}}\langle k2(x1,y1), k2(x2,y2)\rangle$$
$$k3(x1@x2,y1@y2) = \widehat{e_{k3}}\langle k1(x2,y2)\rangle$$

The various segments for these functions are shown in Figure 4. There are six segments altogether (collectively called $\mathcal{B}$), with two synchronization points: *k1* and *k2*. For the point *k1*, the cycle (segment) passing through the intermediate point *k3* synchronizes at level-*2* with another cycle *(@[1],@[1])*. Closer investigation reveals that it can be split into two sub-segments: a prefix, *(@[1],@[1])*, and a suffix, *(@[2],@[2])*, such that (i) the prefix is identical to the cycle *(@[1],@[1])*, and (ii) the suffix together with the rest of the segment satisfies the predicate *Sync*. Therefore, *k1* is a P2 function.

The only equation having calls with identical-MRP is the first equation, which has calls *k1(x1,y1)*, *k2(x1,y1)*, *k3(x1,y1)*. This is the place where potential duplication of function calls could occur.

Applying tupling transformation on this set of P2-functions will introduce the following two tuple-functions for the calls with overlapping- (specifically, identical-) MRP.

$$ktup1(x,y) = (k1(x,y),k2(x,y),k3(x,y))$$
$$ktup2(x,y) = (k1(x,y),k2(x,y))$$

The end-result is the following efficient transformed program:

$$\begin{aligned}
k1(x1@x2,y1@y2) &= let\ \{(a,b,c) = ktup1(x1,y1)\}\ in\ \widehat{e_{k1}}\langle a,b,c,k1(x2,y2)\rangle \\
k2(x1@x2,y1@y2) &= \widehat{e_{k2}}\langle k2(x1,y1), k2(x2,y2)\rangle \\
ktup1(x1@x2,y1@y2) &= let\ \{(a,b,c)=ktup1(x1,y1);\ (d,e)=ktup2(x2,y2)\} \\
&\quad in\ (\widehat{e_{k1}}\langle a,b,c,d\rangle,\ \widehat{e_{k2}}\langle c,e\rangle, \widehat{e_{k3}}\langle d\rangle) \\
ktup2(x1@x2,y1@y2) &= let\ \{(a,b,c)=ktup1(x1,y1);\ (d,e)=ktup2(x2,y2)\} \\
&\quad in\ (\widehat{e_{k1}}\langle a,b,c,d\rangle,\ \widehat{e_{k2}}\langle c,e\rangle)
\end{aligned}$$

The effect of tupling transformation can also be seen in the resulting labelled call graph (which is omitted here). All the synchronization points in the graph share a common charateristic: Every pair of segments starting from the same synchronization point is incompatible. This indicates that redundant calls have been eliminated.

Note that the definition of P2-functions, which was meant for MRP-functions, can also be used to characterise SRP-functions. Specifically, every segment of SRP-functions always satisfies the *Sync* predicate, as it can always be broken into sub-segments which are either incompatible ($\approx_0$) or level-3 synchronizing ($\approx_3$) without encountering level-1 synchronization ($\approx_1$). Hence, the definition of P2-functions, as well as its corresponding transformation technique, can be viewed as a proper extension of the original tupling method for SRP-functions.

4.3.1 Termination Proof

There are only two potential places of non-termination.

- Infinite application of unfolding without instantiation (Step 1).
- Infinite number of *define* steps (Step 5).

Infinite application of unfolding without instantiation at (Step 1) is not possible as all P2-functions are required to have descending-MRPs, as implied by the Condition 1 in Definition 18. Infinite *define* at (Step 5) is also impossible because there are only a finite number of different tuples: Firstly, there are finitely many different non-recursion arguments since they can only be drawn from a bounded set of variables and constants. Secondly, there are finitely many different recursion arguments because the size of the recursion arguments is bounded: If two calls come from two cyclic transition sequences which are synchronized at level-3, then (during tupling transformation) the growth of their corresponding recursion arguments is bounded by the maximum length of those synchronized cycles. On the other hand, if two calls are from incompatible cyclic transition sequences, then they will be split into separate tuples. Lastly, there is a finite number of function names. These three factors implies that the number of different tuples which can be encountered is bounded. Thus, infinite number of *define* (Step 5) is not possible since *folding* at (Step 4) will be carried out when a matching tuple re-appears.

4.4 Eliminating Inter-Call Traversals

While the Phase 2 tupling algorithm eliminates redundant calls appearing within a set of mutual-recursive functions, this section explores the techniques to eliminate multiple traversals of the same data by calls to different functions. In the following, we define a sub-class of functions that is applicable to this tupling phase.

Definition 19: *Phase 3 MRP-functions*
A M-set of mutual recursive (and auxiliary) MRP-functions is said to be *Phase 3* (or P3) functions if the following conditions are satisfied:

1. Every function satisfies the descending-MRP restriction.
2. For each synchronization point, p, in the corresponding labelled call graph, let $\mathcal{B}_p$ be the set of segments originated at point p. Then,
$$\forall i_p, j_p \in \mathcal{B}_p.\ (i_p \approx_0 j_p).$$
3. Let $\mathcal{B}$ be the set of all the segments in the corresponding labelled call graph. Then *Sync*($\mathcal{B}$) holds, where *Sync* is defined in Definition 18.

Condition 2 above stipulates that no P3-function contains redundant calls. This means a piece of data used as a recursion argument to such a function will not be traversed more than

once. We say that the function has *single-traversal* property. This property ensures that the number of calls to be grouped into a tuple never grow — a necessary condition for safe tupling in this phase. Condition 3 is similar to that defined for P2-functions: It prevents the multiple recursion arguments from growing unboundedly – another necessary condition for safe tupling.

Notice that the bounded-arguments restriction is absent from the above definition. This allows functions with accumulative parameters to be transformed by Phase-3 tupling. However, such accumulative parameters are subject to the generalisation process, to be described in later, in order to ensure safe tupling.

All P2-functions will become P3-functions after Phase-2 tupling. Furthermore, it is possible to have P3-functions that do not conform to the bounded-arguments restriction. (This is a benefit of splitting tupling method into separate phases.)

As every P3-function has single-traversal property, multiple-traversal of a piece of data will occur when the data is used as recursion arguments to more than one P3-functions. This is reflected by the existence of two or more calls to P3-functions with overlapping-MRP.

An outline of the algorithm to eliminate inter-call multiple traversals is presented next.

Method 4: Tupling to Eliminate Inter-Call Traversals

Step 0 Decide the set of P3-function calls to tuple, F.

Step 1 Repeatedly *unfold without instantiation* all P3-function calls of F.

Step 2 Gather each subset of P3-function calls with overlapping-MRP. Generalise the other non-recursion arguments. (This is an *abstraction* step.) For each tuple:

Step 3 If the tuple has ≤ 1 function call, then *terminate*.

Step 4 If the tuple has appeared before, then *fold* against previous definition and *terminate*.

Step 5 Otherwise, introduce a new function definition. This is a *define* step.

Step 6 *Unfold* (with minimal instantiation) a P3 call. Goto Step 1.

Consider the following sample program:

$$
\begin{array}{ll}
g(x,y,z) & = \widehat{e_g}\langle g1(x,y,z), g2(x,y)\rangle \\
g1(x1@x2,y1@y2,z) & = \widehat{e_{g1}}\langle g1(x1,y1,acc(z))\rangle \\
g2(x1@x2,y1@y2) & = \widehat{e_{g2}}\langle g3(x1,y1), g2(x2,y2)\rangle \\
g3(x1@x2,y1@y2) & = \widehat{e_{g3}}\langle g2(x1,y1)\rangle
\end{array}
$$

There are two sets of mutual-recusive functions that are auxiliary to g, namely: $\{g1\}$ and $\{g2,g3\}$. The function $g1$ has an accumulative parameter, z. Both sets of functions belong to the P3 form, with the cycle of $g1$, $(@1,@1)$, synchronizing at level-3 with one of the cycle of $g2$, $(@1;@1,@1;@1)$. Lastly, the first two arguments to function g will be traversed more than once via invocation of functions $g1$ and $g2$.

Applying Phase-3 algorithm to the RHS of g will introduce the following two tuple-functions:

$$
\begin{array}{l}
gtup1(x,y,z) = (g1(x,y,z),g2(x,y)) \\
gtup2(x,y,z) = (g1(x,y,z),g3(x,y))
\end{array}
$$

The definition of g is then transformed into one that has single-traversal property:

$$
\begin{array}{l}
g(x,y,z) = let\ \{(a,b) = gtup1(x,y,z)\}\ in\ \widehat{e_g}\langle a, b\rangle \\
gtup1(x1@x2,y1@y2,z) = let\ \{(a,b) = gtup2(x1,y1,acc(z))\}\ in\ (\widehat{e_{g1}}\langle a\rangle, \widehat{e_{g2}}\langle b, g2(x2,y2)\rangle) \\
gtup2(x1@x2,y1@y2,z) = let\ \{(a,b) = gtup1(x1,y1,acc(z))\}\ in\ (\widehat{e_{g1}}\langle a\rangle, \widehat{e_{g3}}\langle b\rangle)
\end{array}
$$

4.4.1 Termination Proof

There are only two potential places of non-termination.

- Infinite application of unfolding without instantiation (Step 1).
- Infinite number of define steps (Step 5).

Infinite application of (Step 1) is not possible as all P3-functions are required to have descending-MRPs. Infinite *define* at (Step 5) is also impossible because there are only a finite number different tuples: Firstly, the number of P3-calls in each tuple is bounded. Due to the single-traversal property, each P3-call in the tuple can only result in at most one descendant call which maintains synchronization with other calls in the tuple.

Secondly, the non-recursion arguments are always generalised into variables, and thus have bounded size.

Finally, the size of the recursion arguments is also bounded: If two calls with overlapping-MRPs come from cyclic transition sequences which synchronizes at level-3, then during the transformation, the growth of their corresponding recursion arguments is bounded by the sum of two numbers – the maximum argument size of the original calls, and the maximum length of the two synchronized cycles. On the other hand, if two calls are from incompatible cyclic transition sequences, then they will be split into separate tuples.

As there is a finite number of function names, the number of different tuples which could be encountered is bounded. Thus, infinite number of *define* at (Step 5) is not possible since *folding* at (Step 4) will be carried out when a matching tuple re-appears.

5 Related Work and Conclusion

Many techniques have been proposed in the past for avoiding redundant function calls. One of the earliest techniques uses memo-functions [Mic68]. Memo-functions are special functions which remember/store either some or all of their previously computed function calls in a memo-table, so that re-occurring function calls can have their results retrieved from the memo-table rather than re-computed. Though general (with no analysis required), memo-functions suffer from large table management overheads.

Other transformation techniques (e.g. tupling and tabulation) may result in more efficient programs but they usually require program analyses and may be restricted to sub-classes of programs. Most of these techniques analyse the call dependency graphs (DGs) algebraically. The algebraic approach typically uses appropriate conditions of descent functions to infer redundancy patterns in the DGs. Descent functions are those functions which are applied to the arguments of subsidiary recursive calls. Some common relationships of descent functions which were discovered by Cohen[Coh83] are the *common generator*, *periodic commutative* and *commutative* redundancy relationships. Though fairly extensive, there are still other classes of programs (e.g. Tower of Hanoi) which are not easily addressed by this algebraic approach. This is due primarily to the difficulty of analysing all possible descent conditions without getting ad-hoc.

A second approach for detecting call redundancy is to make *direct* searches of the DGs. Richard Bird [Bir80] pointed out the possibility of using pebbling game [Pip80] as a general technique for this purpose. This game searches for a suitable pebble placement sequence, which uses the minimum number of pebbles, to visit the entire graph from the leaves to the root. Similarly, Pettorossi [Pet84] advocated a semi-formal framework for a top-down search of DG

to find eureka matching tuples. However, these past proposals are only semi-formal and semi-automatic. Our newly extended tupling method can be seen as an attempt to combine a simple yet powerful algebraic analyses (on the recursion parameters), together with an automated (search-based) transformation method.

Another related transformation is that proposed by Proietti and Pettorossi[PP91]. They formulated a transformation procedure, called the Elimination Procedure (in short), for eliminating unnecessary variables from logic programs. This procedure is also based on the unfold/fold rules, and is rather powerful because it could perform both fusion (similar to deforestation [Wad88]) and tupling transformations at the same time. The Elimination Procedure is applicable to a class of programs which can be cleanly split into two portions: a *tree-like* sub-program and a *non-ascending* sub-program. In the tree-like sub-program, each clause can be non-linear but each predicate can have at most one parameter. Recursion parameters could be captured in the tree-like sub-program. At best, these functions correspond to SRP-functions. With the synchronization analyses, our extended method now works for MRP-functions, as well.

Both our synchronization analyses and Holst's *finiteness analysis* [Hol91] are sophisticated parameter analyses based on function call transitions. Holst's finiteness analysis can detect increasing/decreasing parameters in order to determine if static (known) arguments are safe to specialise. It was formulated to ensure termination of partial evaluators for strict, first-order functional languages. We focus on a different transformation technique, namely tupling. As a result, a new form of parameter (namely, synchronization) analysis is required to ensure that our method is safe.

In this paper, a systematic extension to the tupling method has been proposed. The extended method now works for sub-classes of functions with multiple recursion parameters. To obtain more optimisation, the new method is also structured into three main phases for the elimination of intra-call traversals, inter-call traversals and redundant calls. Like partial evaluation, the extended tupling method is completely automatic with the help of suitable semantics-based synchronization analyses. Unlike partial evaluation, superlinear speedup is possible. In addition, we can guarantee the transformation's termination. This makes the tupling method a reasonable candidate for incorporation into the next generation of functional language compilers.

6 Acknowlegements

This initial ideas for this work were germinated in April-June 1993 during a visit to Chalmers University of Technology by the first author. The author benefited from fruitful discussions with John Hughes and Carsten Kehler Holst. Insightful comments by Neil Jones have also helped to sharpen the eventual trust of the paper. The support of NUS research grants, RP920614 & RP930611, are acknowledged.

References

[Aug85] Lennart Augustsson. Compiling pattern-matching. In *Conference on Functional Programming and Computer Architecture*, Nancy, France, (Lect. Notes Comput. Sc., vol 201, pp. 368–381) Berlin Heidelberg New York: Springer, 1985.

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

[Bir80] Richard S. Bird. Tabulation techniques for recursive programs. *ACM Computing Surveys*, 12(4):403–417, December 1980.

[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. Tupling: an automatic compile-time memoisation transformation. Technical report, Dept of IS/CS, NUS, May 1995.

[CK93] Wei-Ngan Chin and Siau-Cheng Khoo. Tupling functions with multiple recursion parameters. In *3rd International Workshop on Static Analysis*, Padova, Italy, (Lect. Notes Comput. Sc., vol 724, pp. 124–140) Berlin Heidelberg New York: Springer, 1993.

[Coh83] Norman H. Cohen. Eliminating redundant recursive calls. *ACM TOPLAS*, 5(3):265–299, July 1983.

[Hol91] Carsten Kehler Holst. Finiteness analysis. In *5th ACM Conference on Functional programming Languages and Computer Architecture*, pages 473–495, Cambridge, Massachusetts, August 1991.

[Mic68] Donald Michie. Memo functions and machine learning. *Nature*, 218:19–22, 1968.

[Pet84] Alberto Pettorossi. A powerful strategy for deriving programs by transformation. In *3rd ACM Lisp and Functional Programming Conference*, pages 273–281, 1984.

[Pip80] N. Pippenger. Pebbling. Rc 8258 (# 35937), IBM Thomas J Watson Research Centre, May 1980.

[PP91] M. Proietti and A. Pettorossi. Unfolding - definition - folding, in this order for avoiding unnecessary variables in logic programs. In *Proceedings of PLILP*, Passau, Germany, (Lect. Notes Comput. Sc., vol 528, pp. 347–258) Berlin Heidelberg New York: Springer, 1991.

[RFJ89] C. Runciman, M. Firth, and N. Jagger. Transformation in a non-strict language : An approach to instantiation. In *Glasgow Functional Programming Workshop*, August 1989.

[Wad88] Phil Wadler. Deforestation: Transforming programs to eliminate trees. In *European Symposium on Programming*, pages 344–358, Nancy, France, March 1988.