Fundamenta Informaticae XX (2005) 1–37
IOS Press

Redundant Call Elimination via Tupling

Wei-Ngan Chin*
Department of Computer Science
National University of Singapore
Singapore
`chinwn@comp.nus.edu.sg`

Siau-Cheng Khoo*
Department of Computer Science
National University of Singapore
Singapore
`khoosc@comp.nus.edu.sg`

Neil Jones
DIKU (Compute Science Department)
University of Copenhagen
Denmark
`neil@diku.dk`

Abstract. Redundant call elimination has been an important program optimisation process as it can produce super-linear speedup in optimised programs. In this paper, we investigate use of the tupling transformation in achieving this optimisation over a first-order functional language. Standard tupling technique, as described in [6], works excellently in a restricted variant of the language; namely, functions with single recursion argument. We provide a semantic understanding of call redundancy, upon which we construct an analysis for handling the tupling of functions with multiple recursion arguments. The analysis provides a means to ensure termination of the tupling transformation. As the analysis is of polynomial complexity, it makes the tupling suitable as a step in compiler optimisation.

1. Introduction

Source-to-source transformation can achieve global optimisation through specialisation for recursive functions. Two well-known techniques are *partial evaluation* [18] and *deforestation* [33]. Both techniques have been extensively investigated [29, 17] to discover automatic methods and supporting analyses that can ensure correct and terminating program optimisations.

Tupling is a less known but equally powerful transformation technique. The basic technique works by grouping calls with common arguments together, so that their multiple results can be computed si-

*This research was supported in part by the National University of Singapore research grant R-252-000-138-112.

multaneously. When successfully applied, redundant calls can be eliminated, and multiple traversals of data structures combined.

As an example, consider the Tower of Hanoi function.

```
hanoi(0,a,b,c)     = []
hanoi(n+1,a,b,c)   = hanoi(n,a,c,b)++[(a,b)]++hanoi(n,c,b,a)
```

The call `hanoi(n,a,b,c)` returns a list of moves to transfer `n` discs from pole `a` to `b`, using `c` as a spare pole. (Note that ++ denotes list catenation.) The first parameter is a *recursion* parameter which strictly decreases, while the other three parameters are *permuting* parameters which are *bounded* in values. (A formal classification of parameters will be given later in Section 4.) This definition contains *redundant calls*. For instance, the following two call sequences initiating from the call `hanoi(4,'A','B','C')` will reach an identical call:

$$
\begin{array}{ccc}
 & \texttt{hanoi(4,'A','B','C')} & \\
\swarrow & & \searrow \\
\texttt{hanoi(3,'A','C','B')} & & \texttt{hanoi(3,'C','B','A')} \\
\searrow & & \swarrow \\
 & \texttt{hanoi(2,'A','B','C')} &
\end{array}
$$

It would be desirable to eliminate such call redundancy. By gathering each set of *overlapping calls* (which will be defined formally in Section 4.2) appearing in the definition of `hanoi`, the tupling method introduces two new functions `ht2` and `ht3` to capture tuples of function calls, as follows:

```
ht2(n,a,c,b) = (hanoi(n,a,c,b), hanoi(n,c,b,a))
ht3(n,a,b,c) = (hanoi(n,a,b,c), hanoi(n,b,c,a), hanoi(n,c,a,b))
```

It then transforms `hanoi` to the following :

```
hanoi(n+1,a,b,c) = let (u,v) = ht2(n,a,c,b) in u++[(a,b)]++v
ht2(0,a,c,b)     = ([],[])
ht2(n+1,a,c,b)   = let (u,v,w) = ht3(n,a,b,c)
                   in (u++[(a,c)]++v,w++[(c,b)]++u)
ht3(0,a,b,c)     = ([],[],[])
ht3(n+1,a,b,c)   = let (u,v,w) = ht3(n,a,c,b) in
                   (u++[(a,b)]++v,w++[(b,c)]++u,v++[(c,a)]++w)
```

The transformed function can run in linear time, assuming a constant-time implementation of the ++ operation [23].

Despite a significant loss in modularity and clarity, the resulting tupled function is *desirable* as all call redundancies have been eliminated, and their better performance can be mission critical. Though the benefits of tupling are clear, its wider adoption is presently hampered by the difficulties of ensuring that its transformation always terminates. This problem is crucial since it is possible for tupling to meet infinitely many different tuples of calls, which can cause infinite number of tuple functions to be introduced.

Consider the following contrived function definition:

```
f(n+2,m+4,y) = C2(f(n+1,m+2,C(y)),f(n,m+1,C(C(y))))
f(n,m,y)     = y
```

Here, `C` and `C2` are unary and binary data constructor respectively. The first two parameters of the nested `f`-calls are sub-expressions of their formal counterpart in `f`; this makes these two parameter recursion arguments. The third parameters of both the nested `f`-calls are syntactically larger than the corresponding third formal parameter; we call the third parameter an *accumulating* parameter. Redundant calls exist during invocation of `f`, but typical tupling process fails to terminate when it is performed on `f`. Specifically, tupling process will encounter the following tuples of calls, which are increasing in size:

```
1.  (f(n+1,m+2,C(y)),f(n,m,C(C(y))))
2.  (f(n1+1,m1+2,C(C(y))),f(n1,m1,C(C(C(C(y))))))
3.  (f(n2+1,m2+2,C(C(C(y)))),f(n2,m2,C(C(C(C(y))))),
        f(n2+1,m2+2,C(C(C(C(y))))))
4.  (f(n3+1,m3+2,C(C(C(C(y))))),f(n3,m3,C(C(C(C(y))))),
        f(n3+1,m3+2,C(C(C(C(C(y)))))), f(n3,m3,C(C(C(C(C(C(y))))))))
          ⋮
```

Why does typical tupling fail to stop in this case? Informally, it is because the calls of `f` *overlap*, but tupling fails to capture the *synchronisation* between their recursion parameters `n` and `m` and the accumulating parameter `y`.

To avoid the need for parameter synchronisation, previous proposals in [7, 16] restrict tupling to only functions with a single recursion parameter, and without any accumulating parameters. However, this blanket restriction also rules out many useful functions with multiple recursion and/or accumulating parameters that could be tupled. Consider:

```
repl(Leaf(n),xs)     = Leaf(head(xs))
repl(Node(l,r),xs)   = Node(repl(l,xs),repl(r,sdrop(l,xs)))
sdrop(Leaf(n),xs)    = tail(xs)
sdrop(Node(l,r),xs)  = sdrop(r,sdrop(l,xs))
```

Functions `repl` and `sdrop` are used to replace the contents of a tree by the items from another list, without any changes to the shape of the tree. Redundant `sdrop` calls exist, causing `repl` to have a time complexity of $\mathbf{O}(\mathtt{n}^2)$ where `n` is the size of the input tree. Each of the two functions has a recursive first parameter and an accumulating second parameter. For the calls which overlap, the two parameters synchronise with each other (see Section 6 later). Hence, we can gather `repl(l,xs)` and `sdrop(l,xs)` to form the following function:

```
rstup(l,xs)   = (repl(l,xs),sdrop(l,xs))
```

Applying tupling algorithm to `rstup` yields an efficient $\mathbf{O}(\mathtt{n})$ definition:

```
rstup(Leaf(n),xs)    = (Leaf(head(xs)),tail(xs))
rstup(Node(l,r),xs)  = let (u,v)=rstup(l,xs) ; (a,b)=rstup(r,v)
                       in (Node(u,a),b)
```

In this paper, we begin our investigation by describing, in Section 3, a standard tupling technique which works well on function with single recursion argument. We call this technique *SRP-tupling*. Through the standard tupling, we illustrate the termination issue pertaining to the tupling technique, and provide the reader with the first treatment of such issue. Next, we describe, in Section 4, an extension of SRP-tupling to handle functions with multiple recursion arguments and accumulating arguments. We call the extended technique *MRP-tupling*. In Section 5, we elevate the issue of call redundancy to the semantics level, and investigate the scope and safeness of any analysis that aims at detecting call redundancy. The result lays a semantics foundation upon which a *synchronisation analysis* is built (Section 6). With this new analysis, we show the solution to the termination problem pertaining to MRP-tupling. Specifically, we show how the function `f` defined above can be transformed into the following tupled version:

```
f(n+2,m+4,y)     = let z = C(y) ; (u,v) = f_tup(n,m,z) in C2(u,v)
f(n,m,y)         = y
f_tup(n+1,m+2,y) = let z = C(y) ; (u,v) = f_tup(n,m,z)
                   in (C2(u,v),u)
f_tup(n,m,y)     = (y,C(y))
```

In Section 7, we compare our work to the state of the art research in call-redundancy elimination, before concluding our presentation in Section 8.

For lack of space, we do not include proofs to various theorems in this paper. Readers may refer to the technical report [10] for further detail.

2. Language and Notation

We consider a simple (uncurried) first-order functional language with call-by-value semantics. To simplify our presentation, we consider a program to be a single set of mutual-recursive functions:

Definition 2.1. (A Simple Language)
A program in our simple language comprises a set of mutual-recursive functions:

$$
\begin{array}{lll}
\mathtt{P} & \in & \mathit{Prog} \quad \text{— Program is a set of mutual-recursive functions} \\
\mathtt{P} & ::= & \mathtt{D}_1 \,;\, \cdots \,;\, \mathtt{D}_\mathtt{n} \\
\mathtt{D} & \in & \mathit{Defn} \quad \text{— Function definition is a set of equations} \\
\mathtt{D} & ::= & \mathtt{f}(\mathtt{p}_{11},\ldots,\mathtt{p}_{1\mathtt{m}}) \;=\; \mathtt{t}_\mathtt{k} \,;\, \cdots \,;\, \mathtt{f}(\mathtt{p}_{\mathtt{k}1},\ldots,\mathtt{p}_{\mathtt{km}}) \;=\; \mathtt{t}_\mathtt{k} \\
\mathtt{t} & \in & \mathit{Expr} \quad \text{— Expressions} \\
\mathtt{t} & ::= & \mathtt{v} \mid \mathtt{C}(\mathtt{t}_1,\ldots,\mathtt{t}_\mathtt{n}) \mid \mathtt{f}(\mathtt{t}_1,\ldots,\mathtt{t}_\mathtt{n}) \mid \mathtt{let\ p} \;=\; \mathtt{t\ in\ t}_1 \mid \mathtt{Error} \\
\mathtt{p} & \in & \mathit{Pat} \quad \text{— Patterns} \\
\mathtt{p} & ::= & \mathtt{v} \mid \mathtt{C}(\mathtt{p}_1,\ldots,\mathtt{p}_\mathtt{n})
\end{array}
$$

Functions are defined by sets of *equations*, each of which are distinguished by the set of parameters it can receive. Parameters are written in *pattern* format. By treating booleans as patterns of 0-ary constant `true` and `false`, we can express conditional expressions using equations.

The expressions allowed at the right-hand side (RHS) of an equation include variable ($\mathtt{v}$), construction of data via a constructor ($\mathtt{C}$), call to a function ($\mathtt{f}$), `let` construct, and a keyword (`Error`) for undefined/error value. Note that the tuple constructor, $(\mathtt{t_1}, \ldots, \mathtt{t_n})$, is regarded as an instance of the more general data constructor, $\mathtt{C}(\mathtt{t_1}, \ldots, \mathtt{t_n})$.

Each function must have a "complete" set of equations in the following sense:

1. Pattern parameters from all the equations cover all eventualities (*ie.*, they are exhaustive);

2. Pattern parameters between any pair of the equations do not overlap (*ie.*, they are exclusive) unless the pattern parameter is a pattern variable.

As an illustration, we show a "complete" definition for `null` function.

```
null(Nil)          = true
null(Cons(x,xs))   = false
```

We can also express a function definition in a more succinct way, as follows:

$$\mathtt{f} \stackrel{\mathrm{def}}{=} \{(\mathtt{p_{i1}}, \mathtt{p_{i2}}, \ldots, \mathtt{p_{im}}) \Rightarrow \mathtt{t_i}\}_{\mathtt{i=1}}^{\mathtt{k}}$$

We shall abbreviate a list of expressions $\mathtt{t_1}, \ldots, \mathtt{t_n}$ by $\tilde{\mathtt{t}}$, so that a function call $\mathtt{f}(\mathtt{t_1}, \ldots, \mathtt{t_n})$ could be abbreviated as $\mathtt{f}(\tilde{\mathtt{t}})$. Moreover, multiply nested `let` expressions can be abbreviated:

$$\begin{aligned}&(\mathtt{let}\ \mathtt{v}_1\ \mathtt{=}\ \mathtt{e}_1\ \mathtt{in}\ \cdots\ \mathtt{in}\ \mathtt{let}\ \mathtt{v}_n\ \mathtt{=}\ \mathtt{e}_n\ \mathtt{in}\ \mathtt{e})\\ &\qquad\qquad \equiv (\mathtt{let}\ \mathtt{v}_1\ \mathtt{=}\ \mathtt{e}_1\ ;\ \cdots\ ;\ \mathtt{v}_n\ \mathtt{=}\ \mathtt{e}_n\ \mathtt{in}\ \mathtt{e})\end{aligned}$$

Given two expressions $\mathtt{t_1}$ and $\mathtt{t_2}$, we write $(\mathtt{t_1} \sqsubseteq \mathtt{t_2})$ to express the fact that $\mathtt{t_1}$ is a sub-expression of $\mathtt{t_2}$. A predicate *IsVar* defined over expressions checks if an expression is a variable. Similarly, a predicate *IsConst* checks if an expression is a constant (*ie.*, a variable-free constructor expression).

We distinguish between two classes of function definitions: instantiating and non-instantiating functions. A *non-instantiating* function (abbr. NI-function) is defined using only a single equation without pattern matching:

$$\mathtt{f}(\mathtt{v_1}, \mathtt{v_2}, \ldots, \mathtt{v_n}) = \mathtt{t}$$

Other functions have patterns as the first parameter in their equation, and are referred to as *instantiating* functions (abbr. I-functions). A predicate $IsNI$ determines if a function is an NI-function.

Multiple-Hole Context We introduce a special context notation with multiple holes. To begin with, we define a *hole*, $\#m$, as a special variable labelled with a number, m.

Definition 2.2. (Context with Multiple Holes)
A *context*, ζ, is an expression with holes, defined by the following grammar:

$$\zeta\ \mathtt{::=}\ \mathtt{v}\ \mid \#m\ \mid \mathtt{C}(\zeta_1\ \ldots\ \zeta_n)\ \mid \mathtt{f}(\zeta_1\ \ldots\ \zeta_n)\ \mid \mathtt{let}\ \mathtt{p} = \zeta\ \mathtt{in}\ \zeta_1$$

Each expression $\mathtt{t}$ can be decomposed into a context ζ and a sequence of sub-expressions $\mathtt{t_1}, \ldots, \mathtt{t_n}$ using the notation $\zeta\langle \mathtt{t_1}, \ldots, \mathtt{t_n}\rangle$. This notation is equivalent to $[\mathtt{t_1}/\#1, \ldots, \mathtt{t_n}/\#n](\zeta)$ which stands for the substitution of sub-expressions, $\mathtt{t_1}, \ldots, \mathtt{t_n}$, into their respective holes, $\#1, \ldots, \#n$, in the context ζ.

In expressing our tupling transformation, we often need to pick out all calls, as opposed to arbitrary expressions, to the functions in a set. We express this by naming the set as F, and denote the extraction of calls by the following notation:

$$\mathtt{t} = \zeta^{\mathtt{F}}\langle \mathtt{f_1}(\tilde{\mathtt{t_1}}), .., \mathtt{f_n}(\tilde{\mathtt{t_n}})\rangle$$

For example, consider the expression $(\mathtt{x} + \mathtt{sum}(\mathtt{xs}), 1 + \mathtt{length}(\mathtt{xs}))$. In order to abstract out the calls to functions in the set $\mathtt{F} = \{\mathtt{sum}, \mathtt{length}\}$, we use $\zeta^{\mathtt{F}}\langle \mathtt{sum}(\mathtt{xs}), \mathtt{length}(\mathtt{xs})\rangle$, where $\zeta = (\mathtt{x}{+}\#1, 1{+}\#2)$. Note that in case of nested calls, the context notation will always pick out all outermost calls. To extract the inner calls, we have to use a `let` construct to separate out the inner calls before extraction. For simplicity, we assume that the calls are never nested.

3. SRP-Tupling

In this section, we provide a formal account of the tupling transformation which was prevalently used. We name this transformation *SRP-tupling*, for the reason that it works effectively on a set of functions classified as *SRP-functions* (*cf.*, Definition 3.1.) We describe the termination issue associated with this transformation, and highlight the challenges pertaining to the handling of multiple recursion arguments during tupling.

Tupling is a transformation based on the well-known *fold/unfold* transformation rules invented by Burstall and Darlington [5]. It can be used to merge loops together by combining multiple recursive calls and also to eliminate redundant calls for a class of functional programs. The clever (and difficult) step, frequently called a *eureka step*, of this transformation method is to find appropriate sets (in the format of tuples) of calls which would allow each set of calls to be computed recursively from its previous set.

A classical example to illustrate the ability of the tupling transformation is the transformation of fibonacci function:

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

A call to this function can cause many identical subsidiary (recursive) `fib` calls to be evaluated. Such redundant calls are often analysed using the dependency graph (DG) of function calls [4]. A DG of a function call is a *compressed* representation of the evaluation tree; all syntactically different calls occur only once in the DG. A DG for the `fib` function is illustrated in Figure 1. Nodes in the DG represent subsidiary calls, while arcs represent function calling relationships. Redundant calls can have more than one arc pointing to them in the DG.

In the case of the `fib` definition, the redundant calls cause the time-complexity (in terms of reduction steps) of `fib(n)` to be $\mathbf{O}((\frac{1+\sqrt{5}}{2})^{\mathtt{n}})$ where `n` is the initial argument value. A suitable tuple of calls to help remove redundant calls is `(fib(n+1), fib(n))`, as captured in the following new function definition.

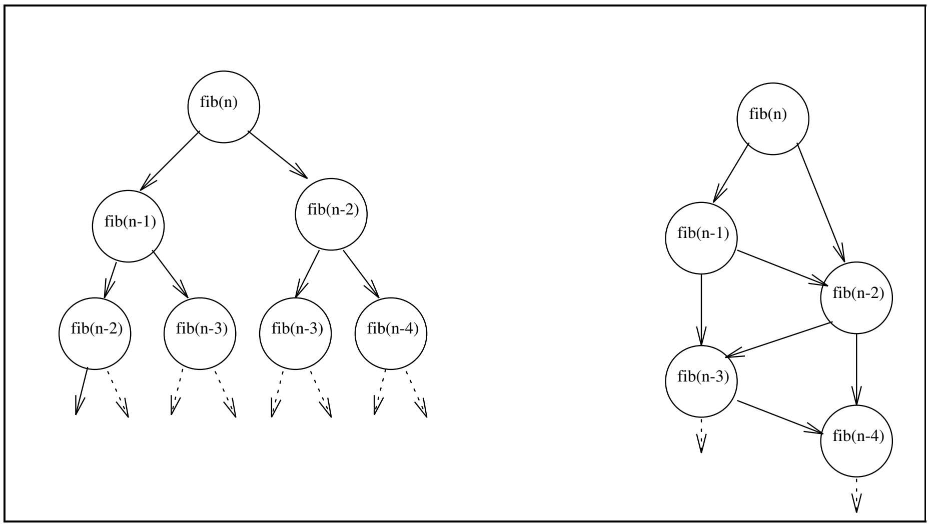

Figure 1. Dependency Graph for `fib` Calls and Its Compressed Representation

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

Unfold/fold transformation can be applied as follows:

Instantiate `n=0`

```
fib_tup(0)   = (fib(0+1), fib(0))            ; unfold fib calls
             = (1, 1)
```

Instantiate `n=n+1`

```
fib_tup(n+1) = (fib(n+2), fib(n+1))          ; unfold fib(n+2)
             = (fib(n+1)+fib(n), fib(n+1))   ; abstract fib calls
             = let (u,v)=(fib(n+1),fib(n))   ; fold fib_tup
               in (u+v,u)
             = let (u,v) = fib_tup(n) in (u+v,u)
```

The end-result is the following recursive function with $\mathbf{O}(n)$ time-complexity.

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

Duplicate calls to `fib` have been eliminated by forcing re-use, instead of re-computation, of the function calls stored in the tuple. `fib_tup` function can be used to provide a new `fib` function that has linear-time complexity, as follows:

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

3.1. SRP-Function

In order to ensure termination of our tupling, we focus on a class of functions, called *SRP-functions*. Consequently, our tupling transformation is called *SRP-tupling*.

Definition 3.1. (SRP-Functions)
Consider a program comprises of a set of mutual-recursive functions $\mathtt{M}$. An equation definition of a function $\mathtt{f_i}$ is said to be an SRP-equation if it has the form:

$$\mathtt{f_i}(\mathtt{p}, \mathtt{v_1}, \ldots, \mathtt{v_n}) \ = \ \zeta^{\mathtt{M}}\langle \mathtt{f_1}(\tilde{\mathtt{t_1}}), .., \mathtt{f_m}(\tilde{\mathtt{t_m}})\rangle$$

such that

1. the pattern $\mathtt{p}$ of the first parameter of an equation is a *simple* pattern; *ie.*, $\mathtt{p}$ is of the form: $\mathtt{p} ::= \mathtt{v_0} \mid \mathtt{C}(\mathtt{v_1}, \ldots, \mathtt{v_n})$, and all remaining parameters are variables;
2. each extracted call in $\zeta^{\mathtt{M}}\langle \mathtt{f_1}(\tilde{\mathtt{t_1}}), .., \mathtt{f_m}(\tilde{\mathtt{t_m}})\rangle$ has the form $\mathtt{f_j}(\mathtt{tj_0}, \tilde{\mathtt{tj}})$ and satisfies $RpCond(\mathtt{p}, \mathtt{tj_0})$, where

$$RpCond(\mathtt{p}, \mathtt{t}) = (\mathit{IsVar}(\mathtt{t})) \wedge (\mathtt{t} \sqsubseteq \mathtt{p})$$

A function $\mathtt{f}$ is said to be an SRP-function if all its equation definitions are SRP-equations.

Note that in an SRP-function, each equation has only one non-variable pattern parameter, namely the first parameter ($\mathtt{p}$). This parameter, for convenience sake, always appears as the first parameter. Condition $RpCond$ states that the first argument ($\mathtt{tj_0}$) of each mutual-recursive call must be a variable taken from this parameter ($\mathtt{p}$). Consequently, the function has only a *single recursion parameter.*

The restriction on using *simple patterns*, with a single constructor each, does not lose generality. There exist pattern-matching translation techniques [2] which can convert functions with arbitrary constructor patterns to equivalent functions of the above restricted form. For example, the `fib` function with nested constructor patterns can be translated to the following mutual-recursive SRP-functions : (Note that we regard $(\mathtt{n}+\mathtt{k})$ where $\mathtt{k}$ is a constant as a peano-style non-negative integer constructor.)

```
fib(0)      = 1
fib(n+1)    = fib'(n)
fib'(0)     = 1
fib'(n+1)   = fib'(n) + fib(n)
```

The corresponding tupled version of $\mathtt{fib}'$ is as follows (`fib_tup` has been defined in Page 7):

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

The recursion parameter is always assumed to be in the position of the first parameter. In addition, we make explicit use of pattern-matching notation to identify recursion parameters. This facilitates the application of fold/unfold transformation, and the tupling of calls with identical recursion arguments.

Following are examples of SRP-equations. `f1` may cause non-termination during execution; `f2` and `f3` are mutually-recursive:

$$
\begin{array}{lcl}
\texttt{f1(v,x)} & = & \zeta_1\langle\texttt{f1(v,x)}\rangle \\
\texttt{f2(C1(v1,v2),x)} & = & \zeta_2\langle\texttt{f2(v1,v2),f3(v1,x)}\rangle \\
\texttt{f3(C1(v1,v2),x)} & = & \zeta_3\langle\texttt{f2(v1,C3(x))}\rangle
\end{array}
$$

but not the following (with the offending sub-expressions underlined):

$$
\begin{array}{lcl}
\texttt{f4(v,x)} & = & \zeta_4\langle\texttt{f4(}\underline{\texttt{x}}\texttt{,x)}\rangle \\
\texttt{f5(C1(v1,v2),x)} & = & \zeta_5\langle\texttt{f5(}\underline{\texttt{C2(v1)}}\texttt{,C3(v2))}\rangle
\end{array}
$$

In the case of `f4`, the recursion argument, `x`, of the recursive call is not taken from the recursion parameter, `v`; hence it does not satisfy the predicate $RpCond$. `f5` also violates $RpCond$ because it has a constructor as its recursion argument, rather than just a simple variable.

3.2. Algorithm for SRP-Tupling

We present a tupling transformation algorithm based on fold/unfold rules. Formally, it is a meta-function of the type:

$$\mathcal{T}^{\mathtt{F}} :: \mathit{Expr} \to \mathit{Prog} \to \mathit{Defn} \to (\mathit{Expr}, \mathit{Eqns})$$

$\mathcal{T}^{\mathtt{F}}$ is parameterised by a set of functions `F`, the calls of which are to be abstracted for tupling. *Prog* is the original program, *Defn* is the set of new function definitions introduced, *Expr* is the expression to be transformed, while (*Expr, Eqns*) contains a transformed expression and new equations generated. To simplify our presentation, we shall frequently focus on the main input and output (ignoring the auxiliary parameters and result) by regarding $\mathcal{T}$ as type:

$$\mathcal{T} :: \mathit{Expr} \to \mathit{Expr}$$

Associated with $\mathcal{T}^{\mathtt{F}}$ is a mutual-recursive counterpart $\mathcal{T}^{\circ\,\mathtt{F}}$, which takes a sequence of calls (of type $[Expr]$) to transform:

$$\mathcal{T}^{\circ\,\mathtt{F}} :: [\mathit{Expr}] \to \mathit{Prog} \to \mathit{Defn} \to (\mathit{Expr}, \mathit{Eqns})$$

The major operations performed by these two functions are identified by labels; these are: $\{u,\ g\}$ for $\mathcal{T}^{\mathtt{F}}$ and $\{t,\ d,\ f\}$ for $\mathcal{T}^{\circ\,\mathtt{F}}$. The functions are defined in Figures 2 and 3. Specifically, $\mathcal{T}^{\mathtt{F}}$ transforms a single expression, while $\mathcal{T}^{\circ\,\mathtt{F}}$ handles a tuple of I-function calls with common recursion argument.

The first step of $\mathcal{T}^{\mathtt{F}}$ identifies the existence of calls to NI-functions. It calls unf (labelled (u)) when there exist NI-function calls; otherwise, it calls tup (labelled (g)) to handle calls to I-functions.

$$
\begin{array}{ll}
& \mathcal{T}^{\mathtt{F}}[\![\zeta^{\mathtt{F}}\langle \mathtt{f_1}(\tilde{\mathtt{t_1}}),\ldots,\mathtt{f_n}(\tilde{\mathtt{t_n}})\rangle]\!]\ \mathit{fs}\ \mathit{ps}\ = \\
& \quad \mathit{let}\ J\ =\ \{\mathtt{j} \mid \mathtt{j} \in \mathit{1..n},\ \mathit{IsNI}(\mathtt{f_j})\} \\
& \quad \mathit{in}\ \ \mathit{if}\,(J \neq \emptyset)\ \mathit{then}\ \mathit{unf}\,([\![\zeta^{\mathtt{F}}\langle \mathtt{f_1}(\tilde{\mathtt{t_1}}),\ldots,\mathtt{f_n}(\tilde{\mathtt{t_n}})\rangle]\!], J, \mathit{fs}, \mathit{ps}) \\
& \qquad \mathit{else}\ \mathit{tup}\,([\![\zeta^{\mathtt{F}}\langle \mathtt{f_1}(\tilde{\mathtt{t_1}}),\ldots,\mathtt{f_n}(\tilde{\mathtt{t_n}})\rangle]\!], \mathit{fs}, \mathit{ps}) \\
\\
(u) & \mathit{unf}\,([\![\zeta^{\mathtt{F}}\langle \mathtt{f_1}(\tilde{\mathtt{t_1}}),\ldots,\mathtt{f_n}(\tilde{\mathtt{t_n}})\rangle]\!], J, \mathit{fs}, \mathit{ps})\ = \\
& \quad \mathit{let}\ \ (\tilde{\mathtt{v_j}}, \mathtt{e_j})\ =\ (\mathit{par}(\mathtt{f_j}), \mathit{rhs}(\mathtt{f_j})) \qquad\qquad \forall \mathtt{j} \in \{1..n\} \\
& \qquad \mathtt{d_j}\ =\ \ \mathit{if}\,(\mathtt{j} \in J)\ \mathit{then}\ [\tilde{\mathtt{t_j}}/\tilde{\mathtt{v_j}}](\mathtt{e_j})\ \mathit{else}\ \mathtt{f_j}(\tilde{\mathtt{t_j}}) \qquad \forall \mathtt{j} \in \{1..n\} \\
& \quad \mathit{in}\ (\mathcal{T}^{\mathtt{F}}[\![\zeta\langle \mathtt{d_1},\ldots,\mathtt{d_n}\rangle]\!]\ \mathit{fs}\ \mathit{ps},\ \emptyset) \\
\\
(g) & \mathit{tup}\,([\![\zeta^{\mathtt{F}}\langle \mathtt{f_1}(\tilde{\mathtt{t_1}}),\ldots,\mathtt{f_n}(\tilde{\mathtt{t_n}})\rangle]\!], \mathit{fs}, \mathit{ps})\ = \\
& \quad \mathit{let}\ \ (\mathit{dcalls}, \rho_d) \qquad\qquad = \ \mathit{MkDistinct}\ [\mathtt{f_1}(\tilde{\mathtt{t_1}}),\ldots,\mathtt{f_n}(\tilde{\mathtt{t_n}})] \\
& \qquad ([(\mathit{seq}_{\mathtt{I}}, \mathtt{vs_I})]_{\mathtt{I}=1}^{r}, \rho_s)\ \ = \ \mathit{MkSplit}\ (\mathit{dcalls}) \\
& \qquad (\tilde{\mathtt{v}}_{\mathtt{I}}, (\mathtt{e_I}, \mathit{fs}_{\mathtt{I}})) \qquad\quad = \ (\mathit{MkTuple}\ (\mathtt{vs_I}), \mathcal{T}^{\circ\,\mathtt{F}}[\![\mathit{seq}_{\mathtt{I}}]\!]\ \mathit{fs}\ \mathit{ps})\ \forall\ \mathtt{I}\ \in\ 1..r \\
& \quad \mathit{in}\ (\mathtt{let}\ \{\tilde{\mathtt{v}}_{\mathtt{I}} = \mathtt{e_I}\}_{\mathtt{I}=1}^{r}\ \mathtt{in}\ (\rho_s \circ \rho_d)(\zeta\langle \#\mathit{1},\ldots,\#\mathit{n}\rangle), \bigcup_{\mathtt{I}=1}^{r} \mathit{fs}_{\mathtt{I}})
\end{array}
$$

Note: Auxiliary functions are described in Section 3.2.1 on page 12.

Figure 2. Tupling Transformation Algorithm – Part 1

In Rule (u), NI-function calls are *unfolded without instantiation*. This transformation continues until only I-function calls remain.

In Rule (g), subsets of I-function calls with the same recursion argument are gathered to form tuples. This is also called the *abstraction* step. The tuples formed are then subject to transformation by $\mathcal{T}^{\circ\,\mathtt{F}}$.

$\mathcal{T}^{\circ\,\mathtt{F}}$ is applied on each tuple formed in the previous rule (g). If the tuple is found to contain at most one function call, the operation on this tuple will be *terminated*, as it is only profitable to tuple two or more calls. This decision to terminate operation is labelled by (t).

If a tuple contains more than one I-function call, function *fold* (labelled (f)) is called to check if the tuple has appeared in an earlier stage of the transformation. If that is so, this tuple of calls must have been kept as a folding point in the set ps. The existing tuple is then replaced by the call to the tuple function identifying this folding point, and the operation on this tuple *terminates*.

When the tuple does not appear in any earlier stage of transformation, function *dfn* (labelled (d)) is invoked. First, it *defines* a new tuple function to replace the tuple of I-calls. This new tuple definition is then kept in ps as a new folding point. A folding pont consists of a tuple of I-function calls, and a name for the new tuple definition. Next, the new tuple definition is further transformed by simultaneously unfolding (with instantiation) all its I-function calls. Function $\mathcal{T}^{\mathtt{F}}$ is then recursively invoked to transform each new RHS of the tuple function.

Example 1. To illustrate the tupling algorithm, consider the following program with two SRP-functions `deepest` and `depth`.

data `Tree a = Leaf a  | Node(Tree a, Tree a)`

$$
\begin{array}{ll}
(t) & \mathcal{T}^{\circ\,\mathtt{F}}[\![[\mathtt{f_1}(\tilde{\mathtt{t_1}}),\ldots,\mathtt{f_n}(\tilde{\mathtt{t_n}})]]\!]\ \mathit{fs}\ \mathit{ps}\ = \\
& \quad \mathit{if}\ (\mathtt{n} \le 1)\ \mathit{then}\ (\mathit{MkTuple}\ [\mathtt{f_1}(\tilde{\mathtt{t_1}}),\ldots,\mathtt{f_n}(\tilde{\mathtt{t_n}})],\ \emptyset) \\
& \quad \mathit{else}\ \ \mathit{let}\ (\mathit{pcalls},\rho_p) = \mathit{MkPermute}\ [\mathtt{f_1}(\tilde{\mathtt{t_1}}),\ldots,\mathtt{f_n}(\tilde{\mathtt{t_n}})] \\
& \quad\quad \mathit{in}\ \ \mathit{if}\ (\exists\,(\mathit{tcalls},\mathtt{f}') \in \mathit{ps}\ .\ (\mathit{MatchTuple}\ (\mathit{pcalls},\mathit{tcalls})))\ \mathit{then} \\
& \quad\quad\quad\quad \mathit{fold}\ ([\![[\mathtt{f_1}(\tilde{\mathtt{t_1}}),\ldots,\mathtt{f_n}(\tilde{\mathtt{t_n}})]]\!],\mathit{pcalls},\mathtt{f}',\rho_p) \\
& \quad\quad\quad \mathit{else}\ \mathit{dfn}\ ([\![[\mathtt{f_1}(\tilde{\mathtt{t_1}}),\ldots,\mathtt{f_n}(\tilde{\mathtt{t_n}})]]\!],\mathit{fs},\mathit{ps})
\end{array}
$$

$$
\begin{array}{ll}
(f) & \mathit{fold}\ ([\![[\mathtt{f_1}(\tilde{\mathtt{t_1}}),\ldots,\mathtt{f_n}(\tilde{\mathtt{t_n}})]]\!],\mathit{pcalls},\mathtt{f}',\rho_p)\ = \\
& \quad \begin{array}{lll}
\mathit{let}\ \ (\mathit{pcalls},\rho_p) & = & \mathit{MkPermute}\ [\mathtt{f_1}(\tilde{\mathtt{t_1}}),\ldots,\mathtt{f_n}(\tilde{\mathtt{t_n}})] \\
\quad [\mathtt{v_1},\ldots,\mathtt{v_n}] & = & \mathit{newNames}\ (n) \\
\quad [\mathtt{u_0},\tilde{\mathtt{u}}] & = & \mathit{varList}\ \mathit{pcalls} \\
\quad \mathtt{e} & = & (([\mathtt{v}_k/\#k]_{k=1}^{n}) \circ \rho_p)(\#1,\ldots,\#n)
\end{array} \\
& \quad \mathit{in}\ (\mathtt{let}\ (\mathtt{v_1},\ldots,\mathtt{v_n}) = \mathtt{f}'\ (\mathtt{u_0},\vec{\mathtt{u}})\ \mathtt{in}\ \mathtt{e},\emptyset)
\end{array}
$$

$$
\begin{array}{ll}
(d) & \mathit{dfn}\ ([\![[\mathtt{f_1}(\tilde{\mathtt{t_1}}),\ldots,\mathtt{f_n}(\tilde{\mathtt{t_n}})]]\!],\mathit{fs},\mathit{ps})\ = \\
& \quad \begin{array}{lll}
\mathit{let}\ \ \mathit{calls} & = & [\mathtt{f_1}(\tilde{\mathtt{t_1}}),\ldots,\mathtt{f_n}(\tilde{\mathtt{t_n}})] \\
\quad [\mathtt{f_{NEW}}] & = & \mathit{newNames}\ (1) \\
\quad \mathit{ps}' & = & \mathit{ps} \cup \{(\mathit{MkTuple}\ (\mathit{calls}),\mathtt{f_{NEW}})\} \\
\quad [\mathtt{u_0},\tilde{\mathtt{u}}] & = & \mathit{varList}\ \mathit{calls} \\
\quad \{(\mathtt{p_k},\tilde{\mathtt{v_k}}) \Rightarrow \mathtt{e_k}\}_{\mathtt{k}=1}^{m} & = & \mathit{calls} \bowtie \mathit{fs} \\
\quad (\mathtt{e}'_{\mathtt{k}},\mathit{fs}'_{\mathtt{k}}) & = & \mathcal{T}^{\mathtt{F}}[\![\mathtt{e_k}]\!]\ \mathit{fs}\ \mathit{ps}'\ \forall \mathtt{k} \in 1..m \\
\quad \mathit{fs_0}' & = & \{\mathtt{f_{NEW}} \stackrel{\text{def}}{=} \{(\mathtt{p_k},\tilde{\mathtt{v_k}}) \Rightarrow \mathtt{e}'_{\mathtt{k}}\}_{\mathtt{k}=1}^{m}\}
\end{array} \\
& \quad \mathit{in}\ (\mathtt{f_{NEW}}\ (\mathtt{u_0},\tilde{\mathtt{u}}),\mathit{fs_0}' \cup (\bigcup_{k=1}^{m} \mathit{fs}'_k))
\end{array}
$$

Note: Auxiliary functions are described in Section 3.2.1 on page 12.

Figure 3. Tupling Transformation Algorithm – Part 2

```
deepest(Leaf(a))    = [a]
deepest(Node(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(Node(l,r))    = 1+max(depth(l),depth(r))
```

In addition to the fact that a quadratic number of calls to `depth` are redundant, the `Tree` data structure is also traversed multiple times by `depth` and `deepest`. Here, `F={deepest,depth}`, we can apply $\mathcal{T}^{\mathtt{F}}$ to the RHS of the second equation of `deepest`, as shown below:

$\mathtt{deepest(Node(l,r))}\ =\ \mathcal{T}^{\mathtt{F}}[\![\zeta^{\mathtt{F}}\langle\mathtt{depth(l),deepest(l),depth(r),deepest(r)}\rangle]\!]$

$$
\begin{array}{ll}
 & = (g)\text{-rule} \\
 & \quad \texttt{let (u,v)=}\mathcal{T}^{\circ\,\mathtt{F}}[\![[\mathtt{depth(l)},\mathtt{deepest(l)}]]\!] \\
 & \qquad\quad \texttt{(a,b)=}\mathcal{T}^{\circ\,\mathtt{F}}[\![[\mathtt{depth(r)},\mathtt{deepest(r)}]]\!] \\
 & \quad \texttt{in } \zeta^{\mathtt{F}}\langle\texttt{u,v,a,b}\rangle \\
 & = (d),(f)\text{-rules} \\
 & \quad \texttt{let (u,v)=d\_tup(l)} \\
 & \qquad\quad \texttt{(a,b)=d\_tup(r)} \\
 & \quad \texttt{in } \zeta\langle\texttt{u,v,a,b}\rangle
\end{array}
$$

Define

$$
\begin{array}{ll}
\texttt{d\_tup(t)} & = \texttt{(depth(t),deepest(t))}
\end{array}
$$

Case `t=Leaf(a)`

$$
\begin{array}{ll}
\texttt{d\_tup(Leaf(a))} & = \mathcal{T}^{\mathtt{F}}[\![(\mathtt{1},[\mathtt{a}])]\!] \\
 & = (t)\text{-rule} \\
 & \quad \texttt{(1,[a])}
\end{array}
$$

Case `t=Node(l,r)`

$$
\begin{array}{ll}
\texttt{d\_tup(Node(l,r))} & = \mathcal{T}^{\mathtt{F}}[\![\ (1+\mathtt{max}(\mathtt{depth(l)},\mathtt{depth(r)}), \\
 & \qquad \zeta^{\mathtt{F}}\langle\texttt{depth(l),deepest(l),depth(r),deepest(r)}\rangle)]\!] \\
 & = (g)\text{-rule} \\
 & \quad \texttt{let (u,v)=}\mathcal{T}^{\circ\,\mathtt{F}}[\![[\mathtt{depth(l)},\mathtt{deepest(l)}]]\!] \\
 & \qquad\quad \texttt{(a,b)=}\mathcal{T}^{\circ\,\mathtt{F}}[\![[\mathtt{depth(r)},\mathtt{deepest(r)}]]\!] \\
 & \quad \texttt{in (1+max(u,a),}\zeta^{\mathtt{F}}\langle\texttt{u,v,a,b}\rangle\texttt{)} \\
 & = \text{two applications of } (f)\text{-rule} \\
 & \quad \texttt{let (u,v)=d\_tup(l)} \\
 & \qquad\quad \texttt{(a,b)=d\_tup(r)} \\
 & \quad \texttt{in (1+max(u,a),}\zeta^{\mathtt{F}}\langle\texttt{u,v,a,b}\rangle\texttt{)} \qquad \square
\end{array}
$$

For convenience, we have assumed that folding points (ps) are passed from one $\mathcal{T}$ application to another to avoid the creation of any duplicate tuple function definitions. We have also ignored the plumbing caused by shuffling the auxiliary inputs and outputs of $\mathcal{T}$ (and $\mathcal{T}^{\circ}$), concentrating just on the main input and output.

3.2.1. Auxiliary Functions

We describe the auxiliary meta-functions used in the tupling algorithm here. Since an NI-function f is defined by just one equation, its definition can be written as: $\mathtt{f} \stackrel{\mathrm{def}}{=} \{(\vec{v}) \Rightarrow \mathtt{e}\}$. Given such a definition, we use function *par* to retrieve its parameter list $\vec{v}$ and function *rhs* to retrieve its RHS expression `e`.

Functions *MkTuple*, *MkDistinct* and *MkSplit* are used in (g)-rule. *MkTuple* converts a sequence of expressions to a tuple of expressions. *MkDistinct* ensures that all duplicate calls are detected and shared, while *MkSplit* is used to split a sequence of calls to separate sub-sequences based on common recursion arguments. As examples,

$$
\begin{array}{l}
\mathit{MkTuple}\ \texttt{[}\mathtt{e_1},\ldots,\mathtt{e_n}\texttt{] = (}\mathtt{e_1},\ldots,\mathtt{e_n}\texttt{)} \\
\mathit{MkDistinct}\ \texttt{[depth(l),depth(r),depth(l),} \\
\qquad\qquad \texttt{deepest(l),depth(r),deepest(r)]} \\
\quad \texttt{= ([depth(l),depth(r),deepest(l),deepest(r)],} \\
\qquad\quad \texttt{[}\#1/\#1, \#2/\#2, \#1/\#3, \#3/\#4, \#2/\#5, \#4/\#6\texttt{])} \\
\mathit{MkSplit}\ \texttt{[depth(l),depth(r),deepest(l),deepest(r)]}
\end{array}
$$

```
= ([([depth(l),deepest(l)],[v1,v2]),
     ([depth(r),deepest(r)],[v3,v4])],[v1/#1,v2/#3,v3/#2,v4/#4])
```

Function $MkPermute$ arranges calls in a tuple in an order that facilitates folding, while $MatchTuple$ checks if a tuple of calls *exactly-matches* (modulo variable renaming) another tuple of calls. For example, in order to allow a sequence of calls, `[fib(m),fib'(m)]`, to fold with a previous eureka tuple, say `(fib'(n),fib(n))`, we first apply:

MkPermute `[fib(m),fib'(m)] = ((fib'(m),fib(m)),[#2/#1,#1/#2])`

in order to allow $MatchTuple$`((fib'(m),fib(m)),(fib'(n),fib(n)))` to succeed.

The meta-operator $\bowtie$ is used in (d)-rule to apply simultaneous unfolding (with instantiation) of a set of I-function calls. For example, simultaneous unfolding of `sum(xs)`, `length(xs)` can be achieved by:

```
[sum(xs),length(xs)] ⋈ defn =
  { Nil ⇒ (0,0); Cons(x',xs') ⇒ (x'+sum(xs'),1+length(xs'))}
where defn = [ sum ={Nil ⇒ 0; Cons(x',xs') ⇒ x'+sum(xs')},
               length={Nil ⇒ 0; Cons(x',xs') ⇒ 1+length(xs')} ]
```

Lastly, $newName(n)$ is used to generate n new identifiers, while $(varList\ calls)$ is used to extract free variables from the tuple of calls, $calls$. In doing so, the common recursion argument is returned as the first free variable.

3.3. Tuple Derivation

The entire SRP-tupling transformation process can be viewed as a *tree of derivation* for tuples of F calls. Each node is a tuple of calls (to functions in F), coupled with a possibly empty list of variable substitution. A directed link exists between two nodes if transforming the tuple at the source node results immediately in another tuple of calls at the sink node.

Since every link indicates an application of a particular rule in $\mathcal{T}^{\mathtt{F}}$, it is labelled with the name of the rule used. In the case where the link is labelled with d, the variable substitution list associated with the sink of the link describes the substitution performed during the application of (d)-rule.

A particular path in the tree of transformation is called a *sequence of derivation*. In particular, we call a sequence of transformations an *n-step derivation* if the sequence consists of n links. An n-step derivation can be expressed diagrammatically as follows:

$$[\,](\mathtt{f}_{10}(\vec{\mathtt{t}_{10}}),\ldots,\mathtt{f}_{\mathtt{a}0}(\vec{\mathtt{t}_{\mathtt{a}0}})) \stackrel{\gamma_1}{\hookrightarrow} \rho_1(\mathtt{f}_{11}(\vec{\mathtt{t}_{11}}),\ldots,\mathtt{f}_{\mathtt{b}1}(\vec{\mathtt{t}_{\mathtt{b}1}})) \stackrel{\gamma_2}{\hookrightarrow} \cdots \stackrel{\gamma_n}{\hookrightarrow} \rho_n(\mathtt{f}_{1\mathtt{n}}(\vec{\mathtt{t}_{1\mathtt{n}}}),\ldots,\mathtt{f}_{\mathtt{dn}}(\vec{\mathtt{t}_{\mathtt{dn}}}))$$

where $\forall \mathtt{k} \in \{1,\ldots,\mathtt{n}\}.\ \rho_{\mathtt{k}} \neq [\,] \rightarrow \gamma_{\mathtt{k}} \equiv d \wedge \rho_{\mathtt{k}} = [\,] \rightarrow \gamma_{\mathtt{k}} \in \{u,g,t,f\}$. We shall omit labels attached to links when it is clear from the context, and omit the substitution list associated with a tupe when the former is empty. The above n-step derivation can be abbreviated as:

$$(\mathtt{f}_{10}(\vec{\mathtt{t}_{10}}),\ldots,\mathtt{f}_{\mathtt{a}0}(\vec{\mathtt{t}_{\mathtt{a}0}})) \stackrel{\gamma_1}{\hookrightarrow} \bullet \stackrel{\gamma_2}{\hookrightarrow} \bullet \cdots \bullet \stackrel{\gamma_n}{\hookrightarrow} \rho_n(\mathtt{f}_{1\mathtt{n}}(\vec{\mathtt{t}_{1\mathtt{n}}}),\ldots,\mathtt{f}_{\mathtt{dn}}(\vec{\mathtt{t}_{\mathtt{dn}}}))$$

Such derivation sequences have been used to model transformation algorithms by [30] in their excellent comparison of partial evaluation, deforestation and supercompilation. We use derivation sequence to model the tupling process, focusing on tuples of SRP-calls. As an example, the tree of derivation of $\mathcal{T}^{\mathtt{F}}$ on the RHS of the second equation of `fib'` function defined in Page 8 is given below.

$$
\begin{array}{l}
\mathtt{(fib(n),\ fib'(n))} \overset{g}{\hookrightarrow} \mathtt{(fib(n),\ fib'(n))} \\
\qquad \overset{d}{\hookrightarrow} [0/n]\mathtt{()} \overset{(t)}{\hookrightarrow} \bullet \\
\qquad \overset{d}{\hookrightarrow} [m+1/n]\mathtt{(fib'(m),\ fib(m),\ fib'(m))} \overset{g}{\hookrightarrow} \mathtt{(fib(m),\ fib'(m))} \overset{f}{\hookrightarrow} \bullet
\end{array}
$$

We use indentation to illustrate branching. The symbol $\bullet$ denotes the end of a particular derivation sequence. Also, we shall internalise those (g)-steps which result in only a single sub-tuple, omit sequences which result in empty tuples, and eliminate all duplicate calls; so that above derivation can be shortened to:

$$\mathtt{(fib(n),\ fib'(n))} \overset{d}{\hookrightarrow} [m+1/n]\mathtt{(fib(m),\ fib'(m))} \overset{f}{\hookrightarrow} \bullet$$

3.4. Two Restrictions to Ensure Termination

The present tupling algorithm could go into a non-terminating loop for certain SRP-functions. Two possible causes of non-termination are:

- The (u)-rule could be repeated indefinitely.
- An infinite subsequence of (d)-steps could occur (via infinitely many different tuples).

To avoid the above problems, we identify the following two simple restrictions which guarantee termination of the tupling algorithm:

- *descending-RP* restriction (to avoid infinite (u)-rule applications)
- *bounded-argument* restriction (to avoid infinite (d)-rule applications)

3.4.1. Descending RP-Restriction

(u)-rule is used to unfold (without instantiation) each NI-function call until none remains. The objective is to obtain only I-function calls for the eureka tuples.

A problem could occur when an `F`-set of functions has a subset of NI-functions which enters a cycle without ever decreasing its recursion parameter. An example is the following `M`-set in which `f` is an NI-function:

$$
\begin{array}{lcl}
\mathtt{f(xs,y)} & = & \zeta_{\mathtt{f}}\langle\mathtt{f(xs,y),g(xs,y)}\rangle \\
\mathtt{g(Nil,y)} & = & \zeta_{\mathtt{g1}}\langle\rangle \\
\mathtt{g(Cons(x,xs),y)} & = & \zeta_{\mathtt{g2}}\langle\mathtt{f(xs,1),g(xs,y),g(xs,2)}\rangle
\end{array}
$$

If we apply $\mathcal{T}^M$ to the RHS of `f`, a non-terminating derivation involving repeated applications of (u)-rule is encountered, as follows:

$$\begin{array}{ll} \texttt{(f(xs,y),g(xs,y))} & \overset{u}{\hookrightarrow} \texttt{(f(xs,y),g(xs,y),g(xs,y))} \\ & \overset{u}{\hookrightarrow} \texttt{(f(xs,y),g(xs,y),g(xs,y),g(xs,y))} \overset{u}{\hookrightarrow} \cdots \end{array}$$

To avoid this problem, we require that every SRP-function satisfies a *descending recursion parameter* (or descending-RP) restriction. This ensures that every cycle around the recursive functions has at least one I-function to decrease the recursion parameter. This helps ensure that infinite unfolding (without instantiation) of NI-function calls cannot occur.

Definition 3.2. (Descending-RP Restriction)

Given an `M`-set of SRP-functions, $\texttt{f}_1, .., \texttt{f}_\texttt{n}$ with its compressed dependency graph, `G`. This `M`-set of functions is said to satisfy the *descending-RP* restriction iff there are no *cycles* of solely NI-function calls in `G`.

With the descending-RP restriction, our tupling algorithm will never encounter infinite applications of (u)-rule, as shown below.

Proposition 3.1. (Termination Ensured by the Descending-RP Restriction)
Consider a tuple of both I-function and NI-function calls from an `M`-set of descending-RP functions. It will take a finite number of (u)-steps to transform this tuple to yield a tuple of only I-function calls.

3.4.2. Bounded-Argument Restriction

It is possible for a sequence of derivation by $\mathcal{T}$ to be infinite due to the presence of infinitely many different eureka tuples encountered by the (d)-rule. Infinitely many different eureka tuples can occur when:

1. the number of distinct calls in each eureka tuple is *unbounded*, and/or
2. the size of each call is *unbounded*.

To avoid an *unbounded* growth of eureka tuples, we introduce the following restriction on the SRP-functions.

Definition 3.3. (Bounded-Argument Restriction)

Consider an `M`-set of SRP-functions whose equations are each of the form:

$$\texttt{f}_\texttt{i}(\texttt{p}, \tilde{\texttt{v}}) \quad = \zeta_\texttt{i}{}^\texttt{M}\langle \texttt{f}_1(\texttt{t1}_0, \tilde{\texttt{t1}}), \ldots, \texttt{f}_\texttt{m}(\texttt{tm}_0, \tilde{\texttt{tm}})\rangle$$

This equation is said to satisfy the *bounded-argument* restriction if for each SRP-call, $\texttt{f}_\texttt{j}(\texttt{tj}_0, \tilde{\texttt{tj}})$, occurring in its RHS, we have:

$$\forall\, \texttt{t} \in \{\tilde{\texttt{tj}}\}.\ \text{IsConst}(\texttt{t}) \vee (\texttt{t} \in \{\tilde{\texttt{v}}\})$$

A SRP-function is said to adhere to the *bounded-argument* restriction if all its equations are *bounded-argument* SRP-equations.

In the above definition, it is understood that there are only a finite number of constant arguments in an M-set of functions.

Note that each non-recursion argument (from mutual-recursive calls) must either be a constant or a variable from the original set of non-recursion parameters, $\{\tilde{\mathtt{v}}\}$. Examples of bounded-argument SRP-functions are:

$$\begin{array}{lcl} \mathtt{g1(v,x)} & = & \zeta_1\langle\mathtt{g1(v,x)}\rangle \\ \mathtt{g2(C1(v1,v2),x,y,z)} & = & \zeta_2\langle\mathtt{g2(v1,3,y,y),g2(v2,3,2,z)}\rangle \end{array}$$

but not the following:

$$\begin{array}{lcl} \mathtt{g3(C2(v),x)} & = & \zeta_3\langle\mathtt{g3(v,}\underline{\mathtt{w}}\mathtt{),g3(v,2)}\rangle \\ \mathtt{g4(C1(v1,v2),x,y,z)} & = & \zeta_4\langle\mathtt{g4(v1,}\underline{\mathtt{v2}}\mathtt{,}\underline{\mathtt{w}}\mathtt{,y),g4(v1,x,y,}\underline{\mathtt{acc(z)}}\mathtt{)}\rangle \end{array}$$

where w is a local variable. The sub-expressions which violate the bounded-argument restriction are underlined. If we apply $\mathcal{T}^{\mathtt{M}}$ to the RHS of g3, we would encounter a non-terminating derivation involving infinite sub-sequence of (d)-rule, as follows:

$$\begin{array}{l} (\mathtt{g3(v,w),g3(v,2)}) \overset{d}{\hookrightarrow} [\mathtt{C2(v_1)/v}](\mathtt{g3(v_1,w_1),g3(v_1,2),g3(v_1,w_{1a})}) \\ \qquad \overset{d}{\hookrightarrow} [\mathtt{C2(v_2)/v_1}](\mathtt{g3(v_2,w_2),g3(v_2,2),g3(v_2,w_{2a}),g3(v_2,w_{2b})}) \overset{d}{\hookrightarrow} \cdots \end{array}$$

SRP-functions with the bounded-argument restriction do not have this problem, as shown in the following proposition.

Proposition 3.2. (Termination Ensured by the Bounded-Argument Restriction)
Consider an M-set of mutual-recursive SRP-functions which adhere to the *bounded-argument* restriction. The number of different tuples formed during $\mathcal{T}^{\mathtt{M}}$ from the I-function calls is finite. Hence, the number of different (d)-rule applications in any derivation sequence is finite.

The above proposition states that the tupling algorithm will always encounter a bounded number of eureka tuples for SRP-functions with the bounded-argument restriction. However, this upper bound, $2^{n\times s^m}$, is still potentially a very large number. Though this upper bound is theoretically very large, we believe that the number of tuples actually encountered in practice will often be much smaller. This has been so for all the examples we investigated.

As an example, consider the Tower of Hanoi function shown in Section 1, the definition of which is repeated below:

$$\begin{array}{lcl} \mathtt{hanoi(0,a,b,c)} & = & \mathtt{[]} \\ \mathtt{hanoi(1+n,a,b,c)} & = & \mathtt{hanoi(n,a,c,b)} +\!\!+ \mathtt{[(a,b)]} +\!\!+ \mathtt{hanoi(n,c,b,a)} \end{array}$$

There are one I-function, three non-recursion parameters and three initial variables. Hence, the maximum number of different eureka tuples which could be encountered is $2^{1\times 3^3} = 2^{27}$. Fortunately, this is only a theoretic limit. In actual transformation, only two eureka tuples are encountered; they are underlined in the following tree of derivation.

$\underline{\texttt{(hanoi(n,a,c,b),hanoi(n,c,b,a))}} \overset{d}{\hookrightarrow}$

$\quad$ $\texttt{[m+1/n](hanoi(m,a,b,c),hanoi(m,b,c,a),hanoi(m,c,a,b),hanoi(m,a,b,c))} \overset{g}{\hookrightarrow}$

$\quad$ $\underline{\texttt{(hanoi(m,a,b,c),hanoi(m,b,c,a),hanoi(m,c,a,b))}} \overset{d}{\hookrightarrow}$

$\quad$ $\texttt{[s+1/m](hanoi(s,a,c,b),hanoi(s,c,b,a),hanoi(s,b,a,c))} \overset{f}{\hookrightarrow} \bullet$

To link the above derivation to the transformed program of `hanoi` shown in Page 2, we see that the first underlined tuple defines the function `ht2` and the second defines `ht3`. Their definitions are repeated below:

```
hanoi(n+1,a,b,c) = let (u,v) = ht2(n,a,c,b) in u++[(a,b)]++v
ht2(0,a,c,b)     = ([],[])
ht2(n+1,a,c,b)   = let (u,v,w) = ht3(n,a,b,c)
                   in (u++[(a,c)]++v,w++[(c,b)]++u)
ht3(0,a,b,c)     = ([],[],[])
ht3(n+1,a,b,c)   = let (u,v,w) = ht3(n,a,c,b) in
                   (u++[(a,b)]++v,w++[(b,c)]++u,v++[(c,a)]++w)
```

Note that function `ht2` is an intermediate non-recursive function which can be unfolded away after transformation, leaving behind the recursive function `ht3` in the optimised program.

On a different note, Liu *et al.* in [22] have derived an incremental program for `hanoi`, which consists of a recursive function definition, denoted by $\widetilde{hanoi}$. $\widetilde{hanoi}$ serves similar function as `ht3`.

3.5. Safe SRP-Tupling

We now propose a theorem which guarantees termination of the tupling transformation for a sub-class of SRP-functions with the proposed restrictions.

Theorem 3.1. (Safe SRP-Tupling Theorem)
Given an `M`-set of mutual-recursive SRP-functions which adhere to the descending-RP and the bounded-argument restrictions, the SRP-tupling transformation via $\mathcal{T}^{\texttt{M}}$ on any expression (with `M`-function calls) always terminates.

3.6. Limitation of SRP-Tupling

There are currently a number of shortcomings in the SRP-tupling method. Firstly, the method does not directly handle higher-order functions. To handle higher-order functions in general, we have proposed a *higher-order removal* technique [8] which could convert high percentage of higher-order programs to their first-order equivalent. Where successfully converted, this technique indirectly allows the tupling method to be applied to higher-order functions.

Secondly, the SRP-tupling method is unable to handle functions with multiple recursion parameters (MRP). An example is the following definition of function `f`, extracted from the definition given in Page 3:

$\texttt{f(n+2,m+4,y) = }\zeta\langle\texttt{f(n+1,m+2,C(y)),f(n,m,C(C(y)))}\rangle$

During the execution of a call such as `f(a,b,C(d))`, there can be multiple redundant call invocations. However, the SRP-tupling method can only handle SRP-functions. In the rest of the paper, we will describe a fairly sophisticated semantics-based analysis, which will extend the SRP-tupling to handle functions with multiple recursion parameters.

4. MRP-Tupling

To expand the application domain of tupling, we now discuss an extension to the standard tupling method to handle functions with multiple-recursion parameter (MRP). We call this tupling *MRP-tupling*, for obvious reason.

4.1. Operations on Arguments

Safety of MRP-tupling relies on the ability to determine systematic change in the arguments of successive function calls. Such systematic change can be described with appropriate operators, as defined below:

Definition 4.1. (Argument Operators)

1. **Descend operators**: Each data constructor C of arity n in the language is associated with n *descend operators*, which are data destructors. Notation-wise, let $\mathtt{C(t_1,\ldots,t_n)}$ be a data structure, then any of its corresponding data destructors is denoted by $\mathtt{C}^{-i}$, and defined as $\mathtt{C}^{-i}\ \mathtt{C(t_1,\ldots,t_n)} = \mathtt{t}_i$.

2. **Constant operators**: For each constant denoted by c in our program, a *constant operator* $\underline{\mathtt{c}}$ always return that constant upon application.

3. **Identity**: An *identity*, id, is the unit under function composition.

4. **Selectors**: For any n-tuple argument $(\mathtt{a_1,\ldots,a_n})$, the i^{th} *selector*, $\sharp_i$, is defined as $\sharp_i(\mathtt{a_1,\ldots,a_n}) = \mathtt{a}_i$.

5. **Accumulating operators**: An *accumulating operator* is any operator that is not defined above. For instance, a data constructor, such as C described in item 1 above, is an accumulating operator; and so is the tuple constructor $(\mathtt{op_1,\ldots,op_n})$. □

Composition of operators is defined by $(\mathtt{f} \circ \mathtt{g})\ \mathtt{x} = \mathtt{f}\ (\mathtt{g}\ \mathtt{x})$, where f and g are two argument operators. A composition of argument operators forms an *operation path*, denoted by op. It describes how an argument is changed from a caller to a callee through call unfolding. This can be determined by examining the relationship between the parameters and the call arguments appearing in the RHS of the equation. For instance, consider the equation $\mathtt{g(x)} = \zeta_{\mathtt{g}}\langle \mathtt{g(C(x,2))}\rangle$. $\mathtt{C(x,2)}$ in the RHS can be constructed from parameter x via the operation path : $\mathtt{C} \circ (\mathtt{id},\ \underline{2})$. To see this, we apply x to the path :

$$(\mathtt{C} \circ (\mathtt{id},\ \underline{2}))\ \mathtt{x} = \mathtt{C}\ ((\mathtt{id},\underline{2})\ \mathtt{x}) = \mathtt{C(x,2)}$$

Changes in an n-tuple argument can be described by an n-tuple of operation paths, called a *segment*, and denoted by $(\mathtt{op}_1,\ldots,\mathtt{op}_n)$. For convenience, we overload the notion `id` to represent an identity operation as well as a segment containing tuple of identity operation paths.

Segments can be used in function graphs to show how function arguments are changed:

Definition 4.2. (Labelled Call Graph)
The *labelled call graph* of a set of mutual-recursive functions F, denoted as $(\mathtt{N}_F,\ \mathtt{E}_F)$, is a graph whereby each function name from F is a node in $\mathtt{N}_F$; and each caller-callee transition is represented by an arrow in $\mathtt{E}_F$, labelled with the segment information. □

Figure 4 depicts the labelled call graph of the function `hanoi` defined in Section 1. Readers may refer to Figure 6 in Page 24 for an example of a labelled call graph for mutual-recursive functions.

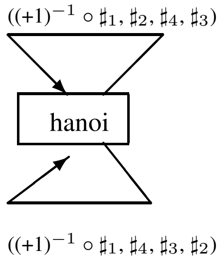

Figure 4. Labelled Call Graph of `hanoi` defined in Section 1.

We use segments to characterise function parameters. This characterisation stems from the way that parameters are changed across labelled call graph of mutually recursive functions.

Definition 4.3. (Characterising Parameters/Arguments)
Given an equation of the form $\mathtt{f(p_1,\ldots,p_n)\ =\ t}$,

1. A group of `f`'s parameters are said to be *bounded parameters* if their corresponding arguments in each recursive call in `t` are derived via either constants, identity, or application of selectors to this group of parameters.
2. The i^{th} parameter of `f` is said to be a *recursion parameter* if it is not bounded and the i^{th} argument of each recursive call in `t` is derived by applying a series of either descend operators or identity to the i^{th} parameter.
3. Otherwise, the `f`'s parameter is said to be an *accumulating parameter*. □

Correspondingly, an argument to a function call is called a *recursion* (resp. *accumulating*) *argument* if it is located at the position of a recursion (resp. accumulating) parameter.

As examples, the first parameter of function `hanoi` is a recursion parameter, whereas its other parameters are bounded. In the case of functions `repl` and `sdrop` (also defined in Section 1), both their first parameters are recursion parameters, and their second parameters are accumulating.

We partition a segment according to the kind of parameters it characterise:

Definition 4.4. (Projections of Segments)
Given a segment s characterising the parameters of an equation, we write $\pi_R(s)$/$\pi_A(s)$/$\pi_B(s)$ to denote the (sub-)tuple of s, including only those operation paths which characterise the recursion/accumulating/bounded parameters. The sub-tuple preserves the original ordering of the operation paths in the segment. Moreover, we write $\overline{\pi_B}(s)$ to denote the sub-tuple of s excluding $\pi_B(s)$. □

Our analysis of segment assumes certain restrictions on the parameters and its relationship with the arguments, as described below:

Definition 4.5. (Restrictions on Parameters)

1. A set of mutual-recursive functions (including their recursive auxiliary functions) has the same number of recursion/accumulating parameters but can have an arbitrary number of bounded parameters.
2. Given an equation, the i^{th} recursion/accumulating argument of any recursive call in the RHS is constructed from the i^{th} parameter of the equation. □

The second restriction above enables us to omit the selector operation from the operation paths derived for recursion/accumulating parameters. All examples in this paper conform to this restriction, except the following three functions:

$$\begin{array}{lcl}
\texttt{b1(C(l,r),n)} & = & \zeta_{\texttt{b1}}\langle \texttt{b1(l,n)}, \texttt{b2(l)} \rangle \\
\texttt{b2(C(l,r))} & = & \zeta_{\texttt{b2}}\langle \texttt{b2(l)}, \texttt{b2(r)} \rangle \\
\texttt{b3(C(x,xs),C(y,ys))} & = & \zeta_{\texttt{b3}}\langle \texttt{b3(xs,ys)}, \texttt{b3(ys,xs)} \rangle
\end{array}$$

Function `b1` has a recursion and a bound arguments. `b2` has a recursion argument. Both of them combined violate the first restriction. Function `b1` has two recursion arguments, and violates the second restriction. Though restrictive, these requirements can be selectively lifted by pre-processing transformation and/or improved analysis. The details are described in a technical report [10].

4.2. Algorithm for MRP-Tupling

As we have seen, the tupling technique generally works by collecting related calls in tuples. In the context of MRP-tupling, these calls have *overlapping* recursion and accumulating arguments. Redundant calls may arise during executing of a function `f` when two (or more) calls in `f`'s RHS have overlapping recursion arguments. We define the notion of overlapping below:

Definition 4.6. (Call Overlapping)

1. Two recursion arguments are said to *overlap* each other if they share some common variables.
2. Two accumulating arguments are said to *overlap* each other if one is a substructure of the other.
3. Two calls *overlap* if each corresponding pairs of recursion and accumulating arguments overlap. Otherwise, they are *disjoint*. □

For example, if two functions f1 and f2 have only recursion arguments, then $\mathtt{f1}(\mathtt{C_1}(\mathtt{x_1},\mathtt{x_2}),\ \mathtt{C_2}(\mathtt{x_4},\ \mathtt{C_1}(\mathtt{x_5},\mathtt{x_6})))$ and $\mathtt{f2}(\mathtt{C_2}(\mathtt{x_2},\mathtt{x_3}),\ \mathtt{C_2}(\mathtt{x_5},\mathtt{x_7}))$ have overlapping recursion arguments, whereas $\mathtt{f1}(\mathtt{C_1}(\mathtt{x_1},\ \mathtt{x_2}),\ \mathtt{C_2}(\mathtt{x_4},\mathtt{C_1}(\mathtt{x_5},\mathtt{x_6})))$ and $\mathtt{f2}(\mathtt{x_2},\mathtt{x_7})$ are disjoint.

If two calls overlap, the call graphs initiated from each of them may overlap, and thus may contain redundancy. Hence, during tupling transformation, we gather overlapping calls into a common function body with the intention to eventually detect and eliminate the resulting redundant calls. Once all redundant calls are eliminated, the new RHS will contain only disjoint calls.

$(u_{\mathcal{M}})$ Similar to (u), with calls to $\mathcal{T}$ replaced by calls to $\mathcal{M}$.

$$
\begin{array}{lll}
(g_{\mathcal{M}}) & \multicolumn{2}{l}{\mathit{tup}\,([\![\zeta^{\mathtt{F}}\langle \mathtt{f_1}(\tilde{\mathtt{t}}_1),\ldots,\mathtt{f_n}(\tilde{\mathtt{t}}_\mathtt{n})\rangle]\!], \mathit{fs}, \mathit{ps}) \ =} \\
& \mathit{let}\ \ (\mathit{dcalls}, \rho_d) & = \ \mathit{MkDistinct}\{[\mathtt{f_1}(\tilde{\mathtt{t}}_1),\ldots,\mathtt{f_n}(\tilde{\mathtt{t}}_\mathtt{n})]\\
& \quad ([(\mathit{seq}_\mathtt{I}, \mathtt{vs_I})]_{\mathtt{I}=1}^{r}, \rho_s) & = \ \mathit{MkSplit}\,(\mathit{dcalls})\\
& \quad \tilde{\mathtt{u}}_\mathtt{I} & = \ \mathit{MkTuple}\,(\mathtt{vs_I})\ \forall\,\mathtt{I}\ \in\ 1..r\\
& \quad (\mathit{seq}'_\mathtt{I}, \mathit{ldefs}_\mathtt{I}) & = \ \mathit{AbsAcc}\,(\mathit{seq}_\mathtt{I})\quad \forall\,\mathtt{I}\ \in\ 1..r\\
& \quad (\mathtt{e}'_\mathtt{I}, \mathit{fs}'_\mathtt{I}) & = \ \mathcal{M}^{\circ\,\mathtt{F}}[\![\mathit{seq}'_\mathtt{I}]\!]\ \mathit{fs}\ \mathit{ps}\ \forall \mathtt{I}\in 1..r\\
& \multicolumn{2}{l}{\mathit{in}\ (\ \mathtt{let}\quad \mathit{ldefs}_1;\cdots;\mathit{ldefs}_r;\tilde{\mathtt{u}}_1=\mathtt{e}'_1;\cdots;\tilde{\mathtt{u}}_r=\mathtt{e}'_r\ \ \mathtt{in}(\rho_s\circ\rho_d)(\zeta\langle \#1,\ldots,\#n\rangle),}\\
& \multicolumn{2}{l}{\quad \bigcup_{\mathtt{I}=1}^{r}\mathit{fs}'_\mathtt{I})}
\end{array}
$$

$(t_{\mathcal{M}})$ Similar to (t), with calls to $\mathcal{T}^\circ$ replaced by calls to $\mathcal{M}^\circ$.

$(f_{\mathcal{M}})$ Similar to (f), with calls to $\mathcal{T}^\circ$ replaced by calls to $\mathcal{M}^\circ$.

$$
\begin{array}{lll}
(d_{\mathcal{M}}) & \multicolumn{2}{l}{\mathit{dfn}\,([\![[\mathtt{f_1}(\tilde{\mathtt{t}}_1),\ldots,\mathtt{f_n}(\tilde{\mathtt{t}}_\mathtt{n})]]\!], \mathit{fs}, \mathit{ps}) \ =} \\
& \mathit{let}\ \ \mathit{calls} & = \ [\mathtt{f_1}(\tilde{\mathtt{t}}_1),\ldots,\mathtt{f_n}(\tilde{\mathtt{t}}_\mathtt{n})]\\
& \quad [\mathtt{f_{NEW}}] & = \ \mathit{newNames}\ (1)\\
& \quad \mathit{ps}' & = \ \mathit{ps}\cup\{(\mathit{MkTuple}\ (\mathit{calls}), \mathtt{f_{NEW}})\}\\
& \quad [\mathtt{u_0},\tilde{\mathtt{u}}] & = \ \mathit{varList}\ \mathit{calls}\\
& \quad \{(\mathtt{p_k},\tilde{\mathtt{v}}_\mathtt{k})\Rightarrow \mathtt{e_k}\}_{\mathtt{k}=1}^{m} & = \ \mathit{calls}\bowtie_{\mathcal{M}}\mathit{fs}\\
& \quad (\mathtt{e}'_\mathtt{k}, \mathit{fs}'_\mathtt{k}) & = \ \mathcal{M}^{\mathtt{F}}[\![\mathtt{e_k}]\!]\ \mathit{fs}\ \mathit{ps}'\ \forall \mathtt{k}\in 1..m\\
& \quad \mathit{fs_0}' & = \ \{\mathtt{f_{NEW}}\stackrel{\mathrm{def}}{=}\{(\mathtt{p_k},\tilde{\mathtt{v}}_\mathtt{k})\Rightarrow \mathtt{e}'_\mathtt{k}\}_{\mathtt{k}=1}^{m}\}\\
& \multicolumn{2}{l}{\mathit{in}\ (\mathtt{f_{NEW}}\,(\mathtt{u_0},\tilde{\mathtt{u}}), \mathit{fs_0}'\cup(\bigcup_{k=1}^{m}\mathit{fs}'_k))}
\end{array}
$$

Note: Auxiliary functions are described in Section 3.2.1 on page 12 and this section (Page 22).

Figure 5. MRP-Tupling Algorithm

The algorithm for MRP-tupling is very similar to that for SRP-tupling, except for some modifications at the rules (g) and (d). We thus provide special name for these rules: $(g_{\mathcal{M}})$ and $(d_{\mathcal{M}})$. They are depicted in Figure 5.

To recall, rule (g) is an abstraction step. It gathers each subset of I-function calls to F with the same recursion argument into a tuple. In rule $(g_{\mathcal{M}})$, similar abstraction step is performed. Because accumulating arguments are allowed in the calls, their sizes may grow as calls are being unfolded. To avoid such a growth, and to maintain these arguments at finite size, the arguments need to be abstracted. The meta-function $AbsAcc$ therefore takes in a tuple of overlapping calls, and for each accumulating argument at a specific position across all these calls, abstracts (by providing a local definition) a maximal subexpression that occurs in all these accumulating arguments. For example, consider the two overlapping calls occurring in the definition of `f` below:

$$\texttt{f(n+2,m+4,y) = }\zeta\langle\texttt{f(n+1,m+2,C(y)),f(n,m,C(C(y)))}\rangle$$

The substructure `C(y)` occurs in the accumulating argument of both the calls to `f`, and it is thus replaced by a new variable `z`.

$$\begin{array}{l} \mathit{AccAbs}\ \texttt{[f(n+1,m+2,C(y)),f(n,m,C(C(y)))] =} \\ \qquad\qquad \texttt{([f(n+1,m+2,z),f(n,m,C(z))],[z = C(y)])} \end{array}$$

Next, we consider modification of the rule (d). It defines a new tuple function to replace tuple of I-function calls. The body of this tuple function is obtained by unfolding, *with instantiation*, the calls in the tuple via the meta-operator $\bowtie$. As the SRP-tupling was defined to operate on functions with simple-pattern parameters, all recursion arguments in the tupled calls are variables. Consequently, $\bowtie$ unfold *all* the tupled calls by instantiating these variables.

In the modified rule $(d_{\mathcal{M}})$, a generalised version of $\bowtie$, named $\bowtie_{\mathcal{M}}$ is used. Here, not all the tupled calls might get unfolded with instantiation, because different calls may have different recursion arguments. Thus, instantiating the recursion arguments of a tupled call may not cause the recursion arguments of the other tupled calls to be sufficiently instantiated. Consider the tupled calls to `f` again: `f(n+1,m+2,z)` and `f(n,m,C(z))`. During the operation of $\bowtie_{\mathcal{M}}$, only the first call will be instantiated and unfolded:

$$\begin{array}{l} \texttt{[f(n+1,m+2,z), f(n,m,C(z))]} \bowtie_{\mathcal{M}} \texttt{defn =} \\ \quad \{\ \texttt{(n'+1,m'+2,y')} \Rightarrow \texttt{(}\zeta\langle\texttt{f(n'+1,m'+2,C(y')),f(n',m',C(C(y')))}\rangle\texttt{,} \\ \qquad\qquad\qquad\qquad \texttt{f(n'+1,m'+2,C(y'))) ;}\ \cdots\} \\ \text{where}\ \texttt{defn = [f = }\{\ \texttt{(n'+2,m'+4,y')} \Rightarrow \\ \qquad\qquad\qquad \zeta\langle\texttt{f(n'+1,m'+2,C(y')),f(n',m',C(C(y')))}\rangle\texttt{;}\ \cdots\}\ \texttt{]} \end{array}$$

Since each tupled call can have different data structures in its recursion arguments, it is natural to ask which call will be targeted for instantiation and unfold. Among the calls available, we choose to unfold *any* call having *maximal recursion* arguments; that is, the recursion arguments, treated as a tree-like data structure, is deepest in depth among the calls.

Example 2. A formal application of $\mathcal{M}^{\mathtt{F}}$ to a RHS of `f` is shown below, with $\mathtt{F} = \{\mathtt{f}\}$.

$$\begin{array}{rl} \texttt{f(n+2,m+4,y)} & = \mathcal{M}^{\mathtt{F}}[\![\zeta\langle\texttt{f(n+1,m+2,C(y)), f(n,m,C(C(y)))}\rangle]\!] \\ & = (g_{\mathcal{M}})\text{-rule} \\ & \quad \texttt{let z = C(y) ; (u,v) = } \mathcal{M}^{\circ\,\mathtt{F}}[\![\mathtt{[f(n+1,m+2,z),f(n,m,C(z))]}]\!] \\ & \quad \texttt{in}\ \ \zeta\langle\texttt{u,v}\rangle \\ & = (d_{\mathcal{M}}),(f_{\mathcal{M}})\text{-rules} \\ & \quad \texttt{let z = C(y) ; (u,v) = f\_tup(n,m,z) in }\zeta\langle\texttt{u,v}\rangle \end{array}$$

Define

```
 f_tup(n,m,y) = (f(n+1,m+2,y),f(n,m,C(y)))
Case (n,m)= (n+1,m+2)
 f_tup(n+1,m+2,y)) = M^F[[ (ζ⟨f(n+1,m+2,C(y)),f(n,m,C(C(y)))⟩,
                             f(n+1,m+2,C(y)) ]]
                   = (g_M)-rule
                     let  z = C(y) ;  (u,v)=M°F[[f(n+1,m+2,z),f(n,m,C(z))]]
                     in   (ζ⟨u,v⟩, u)
                   = (f_M)-rules
                     let  z = C(y) ;  (u,v)=f_tup(n,m,z) in (ζ⟨u,v⟩, u)
```

The final result after MRP-tupling is as follows:

```
 f(n+2,m+4,y)     = let z = C(y) ; (u,v) = f_tup(n,m,z) in ζ⟨u,v⟩
 f_tup(n+1,m+2,y) = let z = C(y) ; (u,v) = f_tup(n,m,z) in (ζ⟨u,v⟩,u)
```

□

The equation of f above is interesting in that it contains two recursion arguments – both being consumed at different rate – as well as an accumulating argument. The computation of call `f(n,m,C(C(y)))` is repeated in the computation of `f(n+1,m+2,C(y))`. This makes the computational complexity of f exponential.

Although effective in eliminating redundant calls, execution of the algorithm $\mathcal{M}$ may not terminate in general due to one of the following reasons: (1) repeatedly applying rule $(u_{\mathcal{M}})$ results in unfolding (without instantiation) calls indefinitely; (2) repeatedly applying rule $(d_{\mathcal{M}})$ may introduce infinitely many new tuple definitions.

We address the first termination issue (infinite unfolding without instantiation) in the following section (Section 4.3), and we leave the second termination issue (infinite generations of new tuple definitions) to Section 6, after we have understood the semantics behind call redundancy.

4.3. Preventing Indefinite Unfolding

We provide a simple condition to prevent MRP-tupling from admitting indefinite unfolding when applying rule $(u_{\mathcal{M}})$. This condition is similar to the "descending-RP restriction" defined in Definition 3.2. Here we rephrase it using segment notation, and extend it to cover multiple-recursion arguments.

We first define a *simple cycle* as a simple loop (with no repeating node, except the first one) in a labelled call graph. Example 3 illustrates the notion of *simple cycle*:

Example 3. Consider the following set of equations:

```
 f1(x1,y1)              = ζ1⟨f2(x1,y1),f3(x1,y1)⟩
 f2(C(x1,x2),C(y1,y2))  = ζ2⟨f2(x2,y1),f3(C(x1,x2),C(y1,y2))⟩
 f3(x1,C(y1,y2))        = ζ3⟨f1(x1,C(y1,y2)),f1(x1,y1),f3(x1,y2)⟩
```

The labelled call graph is depicted in Figure 6. A simple loop in a labelled call graph is called a *simple cycle*. For the above set of equation, the simple cycles as well as their corresponding (compacted) segments are as follows:

$$
\begin{array}{ll}
\texttt{f1} \xrightarrow{\text{a}} \texttt{f2} \xrightarrow{\text{d}} \texttt{f3} \xrightarrow{\text{e}} \texttt{f1} & (\texttt{id},\texttt{id}) \\
\texttt{f1} \xrightarrow{\text{a}} \texttt{f2} \xrightarrow{\text{d}} \texttt{f3} \xrightarrow{\text{f}} \texttt{f1} & (\texttt{id},\texttt{C}^{-1}) \\
\texttt{f1} \xrightarrow{\text{g}} \texttt{f3} \xrightarrow{\text{e}} \texttt{f1} & (\texttt{id},\texttt{id}) \\
\texttt{f1} \xrightarrow{\text{g}} \texttt{f3} \xrightarrow{\text{f}} \texttt{f1} & (\texttt{id},\texttt{C}^{-1}) \\
\texttt{f2} \xrightarrow{\text{b}} \texttt{f2} & (\texttt{C}^{-2},\texttt{C}^{-1}) \\
\texttt{f3} \xrightarrow{\text{c}} \texttt{f3} & (\texttt{id},\texttt{C}^{-2})
\end{array}
$$

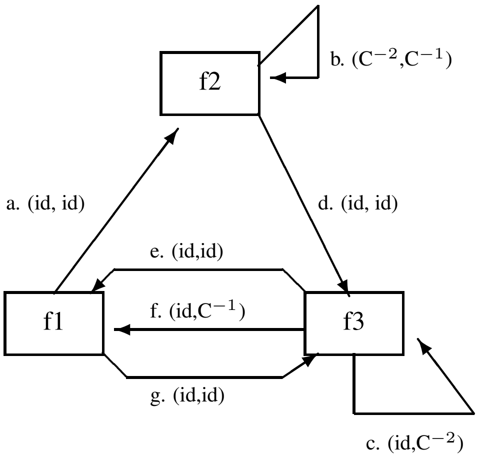

Figure 6. Simple Cycles in a Labelled Call Graph

The set of segments corresponding to the simple cycles in a labelled call graph (N, E) is denoted as $SCycle(N, E)$. This can be computed in time of complexity $\mathbf{O}(|N||E|^2)$ [19].

Theorem 4.1. (Preventing Indefinite Unfolding)
Let F be a set of mutual-recursive functions, each of which has *non-zero number of recursion parameters*. If $\forall\, s \in SCycle(N_{\texttt{F}}, E_{\texttt{F}})$, $\pi_R(s) \neq \{id\}$, then given a call to $\texttt{f} \in \texttt{F}$ with arguments of finite size, there exists a number $N > 0$ such that the call can be successively unfolded (*without instantiation*) not more than N times.

5. Semantics of Call Redundancy

The effectiveness of the MRP-tupling relies on its ability to eliminate redundant calls. As we have seen in earlier examples, although two calls may be invoked with different arguments, call redundancy can still exist during some nested invocation of calls. In this section, we provide a semantics treatment of call redundancy, with the goal of understanding the scope and safeness of any call-redundancy analysis. Furthermore, by being able to detect call duplication, the analysis also guarantees termination of the MRP-tupling process. This is because call duplication can ensure successful folding of call tuples to some existing functions.

To facilitate explanation, we assume that each call occurring in a program text is associated with a unique *call label*. Notationally, we write $c : \texttt{f} \rightarrow \texttt{g}$ or $\texttt{f} \xrightarrow{c} \texttt{g}$ just in case there exists an equation of f of form:

$$\texttt{f}(\texttt{p}_1, \ldots, \texttt{p}_\texttt{n}) = \ldots \texttt{g}(\texttt{t}_1, \ldots, \texttt{t}_\texttt{n}) \ldots$$

and the call to g has been assigned a call label c.

Thus, a *call sequence* is a finite, possibly empty, sequence $cs = c_1 c_2 \ldots c_n$ where there exist function names $\texttt{f}_\texttt{i}$ such that

$$\texttt{f}_1 \xrightarrow{c_1} \texttt{f}_2 \xrightarrow{c_2} \cdots \texttt{f}_\texttt{n} \xrightarrow{c_n} \texttt{f}_{\texttt{n}+1}$$

We also write $cs : \texttt{f}_1 \rightarrow \texttt{f}_{\texttt{n}+1}$ or $\texttt{f}_1 \xrightarrow{cs} \texttt{f}_{\texttt{n}+1}$.

Semantically, we assume the context of a single given program pgm, containing a set of function definition. We use the meta-variables u, v, w, $\cdots$, to denote values drawn from a set **Value** of constants. Thus, every constant, including structured data such as $\texttt{C}(1,2)$, has a corresponding denotational value $[\![\texttt{C(1,2)}]\!] \in$ **Value**.

The computation of a program is defined by an operational semantics that adheres to the strict semantics of the language. Notation-wise, we write the judgment

$$\Gamma \vdash \texttt{t} \Rightarrow v$$

to express the fact that the expression t is evaluated to a denotation v if all the free variables in t have their denotations captured in the environment Γ. We omit the detail operational semantics here, just pointing out that it adheres to the strict semantics of the language.

We use the notion of *states* to capture the semantics of a call, and that of a call sequence. A state σ is a member of the set of all states, or the states for one function f appearing in pgm :

$$\begin{array}{rcl} \sigma \in State^{\texttt{f}} & = & \{(\texttt{f}, \vec{v}) \mid \vec{v} \in \mathbf{Value}^{ar(\texttt{f})}\} \\ \sigma \in State & = & \bigcup\{State^{\texttt{f}} \mid \texttt{f} \text{ is a function in } \texttt{pgm}\} \end{array}$$

where $ar\ (\texttt{f})$ denotes the *arity* of function f .

Given a call labeled with $c : \texttt{f} \rightarrow \texttt{g}$, a *function call invocation* of g at c can be expressed as $c : \sigma \Rightarrow \sigma'$ if

1. $\sigma' = \sigma \cup \{(\texttt{g}, \tilde{v}) \mid \tilde{v} \in \mathbf{Value}^{ar\,(\texttt{g})}\}$,
2. there exists $(\texttt{f}, \tilde{u}) \in \sigma$ such that application of f with arguments $\tilde{u}$ leads to the application of g with arguments $\tilde{v}$.

A function call invocation is also considered a *1-step transition*. Consequently, a *call-sequence transitions* is associated with a call sequence $cs : \sigma \Rightarrow^n \sigma'$, or simply $\sigma \Rightarrow^n \sigma'$, where $n = |cs|$. We may also write $cs : \sigma \Rightarrow \sigma'$ with the same meaning, since the number $n \geq 0$ of steps involved is given by $|cs|$. Conventionally, the transitive closure of $\Rightarrow$ is denoted by $\sigma \Rightarrow^\star \sigma'$.

Definition 5.1. (Call Space)
The *call space* for state σ is a directed graph $\mathcal{CS}(\sigma) = (V, E)$, where (V, E) is the smallest graph such that

- $\sigma \in V$
- If $\sigma_1 \in V$ and $\sigma_1 \stackrel{c}{\Rightarrow} \sigma_2$, then $\sigma_2 \in V$ and $\sigma_1 \stackrel{c}{\Rightarrow} \sigma_2 \in E$.

Definition 5.2. (Call Redundancy)
Program pgm has *call redundancy* from state σ iff $\mathcal{CS}(\sigma)$ is not a tree.

The following definition is the direct result of Definitions 5.1 and 5.2.

Lemma 5.1. (Call Redundancy)
Program pgm has call redundancy from state σ iff there exist states $\sigma_1, \sigma_2, \sigma_3$ such that $\sigma \stackrel{c_1}{\Rightarrow} \sigma_1 \Rightarrow^{n_1} \sigma_3$ and $\sigma \stackrel{c_2}{\Rightarrow} \sigma_2 \Rightarrow^{n_2} \sigma_3$, where c_1, c_2 are distinct calls, and n_1, $n_2 \geq 0$.

A restricted version of call-redundancy problem is to limit the definition of call redundancy to *syntactic call redundancy*.

Definition 5.3. (Syntactic Transfer Function)
The syntactic transfer function $strans(cs) : \mathit{Expr}^{ar(\mathtt{f})} \rightarrow \mathit{Expr}^{ar(\mathtt{g})}$ for call sequence $cs : \mathtt{f} \rightarrow \mathtt{g}$ is a partial function defined as follows:

1. If $c : \mathtt{f} \rightarrow \mathtt{g}$ because the definition of f has the form
$$\mathtt{f}(\mathtt{p_1}, \ldots, \mathtt{p_n}) = \cdots c : \mathtt{g}(\mathtt{t_1}, \ldots, \mathtt{t_m}) \cdots$$
then
$$strans(c)(\mathtt{t}'_1, \ldots, \mathtt{t}'_\mathtt{n}) = (\theta t_1, \ldots, \theta t_m)$$
where $\theta = [p_1 \rhd_s t'_1, \ldots, p_n \rhd_s t'_n]$. Here, the notation $p \rhd_s t$ indicates that variables in p are assigned the corresponding subexpressions in t, provided that both p and t are of the same type; otherwise, the function has no result.[1]
2. $strans(\varepsilon) = id$ (on *Expr* domain).
3. $strans(c\ cs) = strans(cs) \circ strans(c)$.

That the syntactic transfer function describes a form of (semantic-based) call-sequence transition defined earlier can be shown in the following lemma:

Lemma 5.2. (Syntactic Transfer function)
Suppose $strans(cs)(\mathtt{p_1}, \ldots, \mathtt{p_n}) = (\mathtt{t_1}, \ldots, \mathtt{t_m})$ where $cs : \mathtt{f} \rightarrow \mathtt{g}$, then there exists $(u_1, \ldots, u_n) \in \mathbf{Value}^n$, $(v_1, \ldots, v_m) \in \mathbf{Value}^n$ such that if $cs : (\mathtt{f}, \tilde{u}) \Rightarrow (\mathtt{g}, \tilde{v})$, then

$$\{\mathtt{p_1} \rhd u_1, \ldots, \mathtt{p_n} \rhd u_n\} \vdash \mathtt{t_i} \Rightarrow v_i \;\; \forall i \in \{1, \ldots, m\}$$

where $p \rhd v$ indicates that variables in p are assigned the corresponding components of the denotable value v, provided that both p and t are of the same type; otherwise, the function has no result.

Using syntactic transfer function, we can define syntactic call redundancy as follows:

[1]We ignore the presence of let expression in this discussion, for ease of presentation.

Definition 5.4. Program pgm has syntactic call redundancy if

$$strans(cs)(\mathtt{p_1}, \ldots, \mathtt{p_n}) = strans(cs')(\mathtt{p_1}, \ldots, \mathtt{p_n})$$

for two distinct call sequences $cs,\ cs'\ :\ \mathtt{f}\ \rightarrow\ \mathtt{g}$.

The following theorem states that there is no syntactic call redundancy in a SRP-tupled functions.

Theorem 5.1. (Call-Redundancy-Free of SRP-Tupled Functions)
Consider an M-set of mutual-recursive SRP-functions which adhere to the descending-RP and bounded-argument restrictions, if these functions are successfully SRP-tupled, then the resulting tupled functions do not contain syntactic call redundancy.

Undecidability of the Call-Redundancy Problem

Given a program pgm, we would like to detect if call redundancy exists. The following lemma shows that the problem is undecidable, which can be proven by reduction from *Post's Correspondence Problem*.

Lemma 5.3. (Undecidability of Call Redundancy)
It is undecidable whether there exist redundant calls from a function f to a function h.

This Lemma indicates that that the call-redundancy problem is undecidable even if we restrict the problem to syntactic restrict the definition of call redundancy to two calls having *syntactically* identical argument. The task at hand is therefore to find a safe and effective call-redundancy analysis that is a safe approximation of the actual set of call redundancy. In the next section, we will demonstrate such an analysis, which is based on detecting syntactically identical call arguments.

6. Termination of MRP-Tupling

The study of call-redundancy problem in Section 5 lays a semantics foundation upon which we understand the scope and limitation of detecting call redundancy. From the semantics, we understand that call redundancy can be detected by examining sequences of operators applying to function parameters. Therefore, we begin our investigation with the development of an algebra of operators.

6.1. Algebra of Segments

A set of segments forms an algebra under concatenation operation.

Definition 6.1. (Concatenation of Operation Paths)
Concatenation of two operation paths $op1$ and $op2$, denoted by $op1\ ;op2$, is defined as $op2\ \circ\ op1$. □

Definition 6.2. (Concatenation of Segments)
Concatenation of two segments $s1$ and $s2$, denoted by $s1;s2$, is defined component-wise as follows:

$$s1 = (\ op_1, \ldots, op_n\ ) \wedge s2 = (\ op'_1, \ldots, op'_n\ ) \Rightarrow s1\ ;\ s2 = (\ op_1;op'_1, \ldots, op_n;op'_n\ ). \quad \square$$

A concatenated segment can be expressed more compactly by applying the following *reduction rules*:

1. For any operator O, $\texttt{id} \circ O$ reduces to O, and $O \circ \texttt{id}$ reduces to O.

2. $\forall\ O1, \ldots, On$ and O, $(O1 \circ O, \ldots, On \circ O)$ reduces to $(O1, \ldots, On) \circ O$.

Applying the above reduction rules to a segment yields a **compacted segment**. On the other hand, by viewing these reduction rules as bi-directional conversions of segments, we can define an equivalence relation among a set of segments. Specifically, two segments are said to be *equivalent* if and only if they can be converted to one another under these conversion rules. In this respect, compacted segment is a canonical representation of an equivalence class of segments. Henceforth, we deal with compacted segment, and use the term "segment" and "compacted segment" interchangeably to mean the latter, unless stated otherwise.

Lastly, concatenating a segment, s, n times is expressed as s^n. Such repetition of segment leads to the notion of factors of a segment, as described below:

Definition 6.3. (Factorisation of Segments)
Given segments s and f. f is said to be a *factor* of s if $\exists n > 0. s = f^n$. We call n the *power* of s wrt f. □

We note that if $s1 = (op_1, \ldots, op_n)$ has a factor of power k, then each op_i, for $i \in 1 \ldots n$, has a factor of power k too.

For example, $(\texttt{C}^{-2},\ \texttt{id})$ is a factor of $(\texttt{C}^{-2} \circ \texttt{C}^{-2} \circ\ \texttt{C}^{-2}, \texttt{id})$, since $(\texttt{C}^{-2}, \texttt{id})^3 = (\texttt{C}^{-2} \circ\ \texttt{C}^{-2} \circ\ \texttt{C}^{-2}, \texttt{id})$. Every segment has at least one factor – itself. On the other hand, when a segment has *exactly one* factor, it is called a *prime segment*. An example of prime segment is $(\texttt{C}_1^{-1} \circ\ \texttt{C}_2^{-1},\ \texttt{id})$.

The following lemma shows that any compacted segment has a unique prime factorisation. This results is critical to the success of detecting synchronisation among call arguments.

Lemma 6.1. (Uniqueness of Prime Factorisation)
Let s be a compacted segment, there exists a *unique* prime segment f such that $s = f^k$ for some $k > 0$.

6.2. Synchronisation Analysis

We now present an analysis that prevents the MRP-tupling from generating infinitely many new tuple functions at the application of rule $(d_{\mathcal{M}})$. Our analysis ensures the finiteness of syntactically different (modulo variable renaming) tupled calls. As a group of bounded arguments can only be obtained from itself by the application of either selectors, identity or constants operators, it can only have finitely many different structures. Consequently, bounded arguments do not cause the MRP-tupling to loop infinitely. Hence, we focus on determining the structure of recursion and accumulating arguments in this section. Specifically, *we can safely ignore bounded arguments in our treatment of segments.*

Since syntactic changes to call arguments are captured by series of segments, differences in call arguments can be characterised by the relationship between the corresponding segments. We discuss below a set of relationships between segments.

Definition 6.4. (Levels of Synchronisation)
Two segments s_1 and s_2 are said to be :

1. *level-1 synchronised*, denoted by $s_1 \simeq_1 s_2$, if

$$\exists s'_1, s'_2.(s_1;s'_1 = s_2;s'_2).$$

   Otherwise, they are said to be *level-0 synchronised*, or simply, *unsynchronised*.

2. *level-2 synchronised* ($s_1 \simeq_2 s_2$) if

$$\exists s'_1, s'_2.((s_1;s'_1 = s_2;s'_2) \wedge (s'_1 = id \vee s'_2 = id)).$$

3. *level-3 synchronised* ($s_1 \simeq_3 s_2$) if

$$\exists s.\exists n, m > 0.(s_1 = s^n \wedge s_2 = s^m).$$

4. *level-4 synchronised* ($s_1 \simeq_4 s_2$) if $s_1 = s_2$. □

Examples of different levels of synchronisation is given below:

$$\begin{array}{lcl} (\mathtt{C}_1^{-1} \circ \mathtt{C}_1^{-1} \circ \mathtt{C}_1^{-1}, \mathtt{C}_2^{-1}) & \simeq_1 & (\mathtt{C}_1^{-1}, \mathtt{C}_2^{-1} \circ \mathtt{C}_2^{-1}) \\ (\mathtt{C}_1^{-1} \circ \mathtt{C}_1^{-1}, \mathtt{C}_2^{-1} \circ \mathtt{C}_2^{-1}) & \simeq_3 & (\mathtt{C}_1^{-1}, \mathtt{C}_2^{-1}) \end{array} \qquad \begin{array}{rcl} (\mathtt{C}_1^{-1} \circ \mathtt{C}_1^{-1}, \mathtt{C}_2^{-1}) & \simeq_2 & (\mathtt{C}_1^{-1}, \mathtt{C}_2^{-1}) \\ (\mathtt{C}_1^{-1}, \mathtt{C}_2^{-1}) & \simeq_4 & (\mathtt{C}_1^{-1}, \mathtt{C}_2^{-1}) \end{array}$$

Levels 1 to 4 of synchronisation form a strict hierarchy, with synchronisation at level `i` implying synchronisation at level `j` if $\mathtt{i} > \mathtt{j}$. Together with level-0, these help identify termination property of the MRP-tupling.

Why does synchronisation play an important role in termination of the MRP-tupling? Intuitively, if two sequences of segments synchronise, then calls following these two sequences will have finite variants of argument structures. This thus enables folding (in the $(d_{\mathcal{M}})$ rule) to take effect, and eventually terminates the transformation.

6.3. MRP-Tupling Termination at Different Synchronisation Levels

In this section, we provide an informal account of some of the interesting findings pertaining to the MRP-tupling, as implied by the different levels of synchronisation.

Finding 1. *Transforming two calls with identical arguments but following level-0 synchronised segments will end up with disjoint arguments.*[2]

Example 4. Consider the following two equations for functions `g1` and `g2` respectively:

$$\begin{array}{lcl} \mathtt{g1}(\mathtt{C}_1(\mathtt{x}_1, \mathtt{x}_2), \mathtt{C}_2(\mathtt{y}_1, \mathtt{y}_2)) & = & \zeta_{\mathtt{g1}}\langle \mathtt{g1}(\mathtt{x}_1, \mathtt{y}_1)\rangle \\ \mathtt{g2}(\mathtt{C}_1(\mathtt{x}_1, \mathtt{x}_2), \mathtt{C}_2(\mathtt{y}_1, \mathtt{y}_2)) & = & \zeta_{\mathtt{g2}}\langle \mathtt{g2}(\mathtt{x}_1, \mathtt{y}_2)\rangle \end{array}$$

[2]Sometimes, two apparently level-0 synchronised may turn into synchronisation of other levels when they are prefixed with some initial segment. Such initial segments may be introduced by the argument structures of the two initially overlapping calls. Such hidden synchronisation can be detected by extending the current technique to handle "rotate/shift synchronisation" [11].

The segment leading to the call $\mathtt{g1}(\mathtt{x}_1,\mathtt{y}_1)$ is $(\mathtt{C}_1^{-1},\mathtt{C}_2^{-1})$, whereas that leading to the call $\mathtt{g2}(\mathtt{x}_1,\mathtt{y}_2)$ is $(\mathtt{C}_1^{-1},\mathtt{C}_2^{-2})$. These two segments are level-0 synchronised. Suppose that we have an expression containing two calls, to `g1(u,v)` and `g2(u,v)` respectively, with identical arguments. The MRP-tupling transforming these two calls will create a tuple function:

```
g_tup(u,v) = (g1(u,v),g2(u,v))
```

This then transforms (through instantiation) to the following:

$$\mathtt{g\_tup}(\mathtt{C}_1(\mathtt{u}_1,\mathtt{u}_2),\mathtt{C}_2(\mathtt{v}_1,\mathtt{v}_2)) \;=\; (\zeta_{\mathtt{g1}}\langle \mathtt{g1}(\mathtt{u}_1,\mathtt{v}_1)\rangle,\zeta_{\mathtt{g2}}\langle \mathtt{g2}(\mathtt{u}_1,\mathtt{v}_2)\rangle)$$

As the arguments of the two calls in the RHS above are now disjoint, the MRP-tupling terminates. However, the effect of MRP-tupling is simply an unfolding of the calls. Thus, it is safe (with respect to termination of the MRP-tupling transformation) but not productive to transform two calls with identical arguments if these calls follow segments that are level-0 synchronised. □

Finding 2. *Tuple of calls with identical arguments that follow level-4 synchronised segments will lead to a tuple of identical structures.*
This has already manifested in the example of `repl` and `sdrop` in Section 1; each of these function has one recursion parameter and one accumulating parameter. There, initial calls gathered are of identical arguments (`repl(l,xs)` and `sdrop(l,xs)`).

Finding 3. *Level-4 synchronisation is a strong assurance for termination of the MRP-tupling, even when the calls gathered are overlapping but not identical.* (This fact is a consequence of Theorem 6.2.)

Example 5. Consider two overlapping calls

$$\mathtt{h2}(\mathtt{C}_1(\mathtt{u}_1,\mathtt{u}_2),\mathtt{v}_1) \text{ and } \mathtt{h2}(\mathtt{u}_1,\mathtt{C}_2(\mathtt{v}_1,\mathtt{v}_2))$$

appearing in the definition of `h1`, as follows:

$$\begin{array}{lll}
\mathtt{h1}(\mathtt{C}_1(\mathtt{u}_1,\mathtt{u}_2),\mathtt{C}_2(\mathtt{v}_1,\mathtt{v}_2)) & = & \zeta_{\mathtt{h1}}\langle \mathtt{h2}(\mathtt{C}_1(\mathtt{u}_1,\mathtt{u}_2),\mathtt{v}_1),\mathtt{h2}(\mathtt{u}_1,\mathtt{C}_2(\mathtt{v}_1,\mathtt{v}_2))\rangle \\
\mathtt{h2}(\mathtt{C}_1(\mathtt{x}_1,\mathtt{x}_2),\mathtt{C}_2(\mathtt{y}_1,\mathtt{y}_2)) & = & \zeta_{\mathtt{h2}}\langle \mathtt{h2}(\mathtt{x}_1,\mathtt{y}_1)\rangle
\end{array}$$

The two calls in the first equation follow the same segment ($\mathtt{C}_1^{-1}$, $\mathtt{C}_2^{-1}$) when there are unfolded; their segments are therefore level-4 synchronised. The MRP-tupling first groups them in the tuple definition `h_tup1` :

$$\mathtt{h\_tup1}(\mathtt{u}_1,\mathtt{u}_2,\mathtt{v}_1,\mathtt{v}_2) \;=\; (\mathtt{h2}(\mathtt{C}_1(\mathtt{u}_1,\mathtt{u}_2),\mathtt{v}_1),\mathtt{h2}(\mathtt{u}_1,\mathtt{C}_2(\mathtt{v}_1,\mathtt{v}_2)))$$

The transformation terminates and yields the following new equations (and functions) :

$$\begin{array}{lll}
\mathtt{h_tup1}(\mathtt{u}_1,\mathtt{u}_2,\mathtt{C}_2(\mathtt{v}_{11},\mathtt{v}_{12}),\mathtt{v}_2) & = & \mathtt{let\ (u,v)\ =\ h_tup2}(\mathtt{u}_1,\mathtt{v}_{11},\mathtt{v}_{12},\mathtt{v}_2) \\
 & & \mathtt{in}\ (\zeta_{\mathtt{h2}}\langle \mathtt{u}\rangle,\mathtt{v}) \\
\mathtt{h_tup2}(\mathtt{C}_1(\mathtt{u}_{11},\mathtt{u}_{12}),\mathtt{v}_{11},\mathtt{v}_{12},\mathtt{v}_2) & = & \mathtt{let\ (u,v)\ =\ h_tup1}(\mathtt{u}_{11},\mathtt{u}_{12},\mathtt{v}_{11},\mathtt{v}_{12}) \\
 & & \mathtt{in}(\zeta_{\mathtt{h2}}\langle \mathtt{u}\rangle,\zeta_{\mathtt{h2}}\langle \mathtt{v}\rangle)
\end{array}$$

Effectively, multiple traversal of data structures arisen from the two initial `h2`-call invocation has been eliminated via the MRP-tupling. □

Finding 4. *Applying the MRP-tupling on calls that follow level-2 synchronised segments may not terminate.*

Example 6. Consider the binomial function defined below:

```
bin(0,k)       = 1
bin(n+1,0)     = 1
bin(n+1,k+1)   = if k≥n then 1 else bin(n,k)+bin(n,k+1)
```

The segments leading to the two calls `bin(n,k)` and `bin( n,k+1)` are (`(+1)`$^{-1}$, `(+1)`$^{-1}$) and (`(+1)`$^{-1}$, `id`) respectively. They are level-2 synchronised. Performing the MRP-tupling on `(bin(n,k), bin(n, k+1))` will keep generating new set of overlapping calls in rule $(d_{\mathcal{M}})$, as shown below:

1. $(\mathtt{bin(n,k)}, \mathtt{bin(n,k+1)})$
2. $(\mathtt{bin(n_1+1,k)}, \mathtt{bin(n_1,k)}, \mathtt{bin(n_1,k+1)})$
3. $(\mathtt{bin(n_1,k_1)}, \mathtt{bin(n_1,k_1+1)}, \mathtt{bin(n_1,k_1+2)})$
4. $(\mathtt{bin(n_2+1,k_1)}, \mathtt{bin(n_2,k_1)}, \mathtt{bin(n_2,k_1+1)}, \mathtt{bin(n_2,k_1+2)})$

$\vdots$

Hence, the MRP-tupling fails to terminate. □

The non-termination behaviour of transforming functions such as `bin` can be predicted from the synchronisability of its two segments — Given two sequences of segments, s_1 = (`(+1)`$^{-1}$, `(+1)`$^{-1}$) and s_2 = (`(+1)`$^{-1}$, `id`). If these two sequences are constructed using only s_1 and s_2 respectively, then it is impossible for the two sequences to be identical (though the calls they represent remain overlapping).

This tupling failure may be used to suggest more advanced but expensive techniques, such as vector-based [9] or *list-based* [25] memoisations.

Since level-2 synchronisation implies level-1 synchronisation, Finding *4* above applies to calls that follow level-1 synchronised segments as well.

However, if two segments are level-3 synchronised, then it is always possible to build from these segments two sequences that are identical; thanks to the following Prop. 6.1 about level-3 synchronisation.

Property 6.1. (Properties of Level-3 Synchronisation)

1. Let f_1 and f_2 be the prime factors of s_1 and s_2 respectively, then $s_1 \simeq_3 s_2 \Leftrightarrow f_1 = f_2$.
2. Level-3 synchronisation is an *equivalence* relation over segments (*ie.,* it is reflexive, symmetric, and transitive).

Informally, for level-3 synchronised segments, their common prime factor acts as a *common generator*, from which synchronised segments are generated. This idea of *common generator* was first mentioned by Cohen in [12]. Different from [12], we allow co-existence of multiple common generators to classify different classes of synchronisable segments. Common generator, and thus level 3 synchronisation, provides an opportunity for the termination of the MRP-tupling. Indeed, the following theorem highlights such an opportunity.

Theorem 6.2. (Termination Induced by Level-3 Synchronisation)
Let F be a set of mutual-recursive functions with S being the set of segments corresponding to the edges in $(N_{\mathtt{F}}, E_{\mathtt{F}})$. Let C be an initial set of overlapping F-calls to be tupled. If

1. $\forall s \in SCycle(N_{\mathtt{F}}, E_{\mathtt{F}}).\pi_R(s) \neq \{\mathtt{id}\}$,
2. $\forall s_1, s_2 \in S.\overline{\pi_B}(s_1) \simeq_3 \overline{\pi_B}(s_2)$.

then performing the MRP-tupling on C terminates. □

The notion $\overline{\pi_B}(s)$ was defined in Definition 4.4. The first condition in Theorem 6.2 prevents infinite number of unfolding, whereas the level-3 synchronisation condition ensures that the number of different tuples generated during transformation is finite.

Example 7. Consider the equation of f defined in Example 2:

```
f(n+2,m+4,y) = ζf⟨f(n+1,m+2,C(y)),f(n,m,C(C(y)))⟩
```

Although the recursion arguments in `f(n+1,m+2,C(y))` and `f(n,m,C(C(y)))` are consumed at different rates, the argument consumption (and accumulating) patterns for both calls are level-3 synchronised. Subjecting the calls to the MRP-tupling yields the following result:

```
f(n+2,m+4,y)     = let y1 = C(y) ; (u,v) = f_tup(n,m,y1) in ζf⟨u,v⟩
f_tup(n+1,m+2,y) = let y2 = C(y) ; (u,v) = f_tup(n,m,y2)
                   in (ζf⟨u,v⟩,u)
```
□

Finally, since level-4 synchronisation implies level-3 synchronisation, *Theorem 6.2 applies to segments of level-4 synchronisation as well.*

In situation where segments are not all level-3 synchronised with one another, we describe here a sufficient condition which guarantees termination of the MRP-tupling. To begin with, we observe from Prop. 6.1(b) above that we can partition the set of segments S into disjoint *level-3 sets* of segments. Let $\Pi_S = \{[s_1], \ldots, [s_k]\}$ be such a partition. By Prop. 6.1(a), all segments in a level-3 set $[s_i]$ share a unique prime factor, f_i say, such that all segments in $[s_i]$ can be expressed as $\{f_i^{p_1}, \ldots, f_i^{p_{ni}}\}$. We then define $HCF([s_i]) = f_i^d$ where $d = gcd(p_1, \ldots, p_{ni})$ is the greatest common divisor of $p_1, \ldots,\ p_{ni}$. $HCF([s_i])$ is thus the *highest common factor* of the level-3 set $[s_i]$.

Definition 6.5. (Set of Highest Common Factors)
Let S be a set of segment. The *set of highest common factors* of S, $HCFSet(S)$, is defined as

$$HCFSet(S) = \{\ HCF([s_i]) \mid [s_i] \in \Pi_S\ \}.$$

The following theorem states a sufficient condition for preventing infinite definition (*ie.*, infinite application of $(d_{\mathcal{M}})$-rule) during the MRP-tupling[3]

[3]It is possible to extend Theorem 6.3 further by relaxing its premises. In particular, we can show the prevention of infinite definition in the presence of segments that are level-2 synchronisation, provided such segment can be broken down into two sub-segment, of which one is level-3 synchronised with some of the existing segments, and the other is level-0 synchronised [19].

Theorem 6.3. (Preventing Infinite Definition)
Let F be a set of mutual-recursive functions. Let S be the set of segments correspond to the edges in $(N_{\mathtt{F}}, E_{\mathtt{F}})$. Let C be a set of overlapping calls occurring in the RHS of an equation in F. If $\forall s_1, s_2 \in HCFSet(S).\overline{\pi_B}(s_1) \simeq_0 \overline{\pi_B}(s_2)$, then performing the MRP-tupling on C will generate a finite number of different tuples. □

A note on the complexity of this analysis: We notice from Theorem 6.3 that the main task of synchronisation analysis is to determine that all segments in `HCFSet(S)` are level-0 synchronised. This involves expressing each segment in `S` as its prime factorisation, partitioning `S` under level-3 synchronisation, computing the highest common factors for each partition, and lastly, determining if `HCFSet(S)` is level-0 synchronised. Conservatively, the complexity of synchronisation analysis is *polynomial* wrt the number of segments in `S` and the maximum length of these segments [19].

Theorem 6.4 summarises the results of Theorem 4.1 and Theorem 6.3.

Theorem 6.4. (Termination of The MRP-Tupling)
Let F be a set of mutual-recursive functions, each of which has non-zero number of recursion parameters. Let S be the set of segments corresponding to the edges in $(N_{\mathtt{F}}, E_{\mathtt{F}})$. Let C be a set of overlapping calls occurring in the RHS of an equation in F. If

1. $\forall s \in SCycle(N_{\mathtt{F}}, E_{\mathtt{F}}).\pi_R(s) \neq \{id\}$, and
2. $\forall s_1, s_2 \in HCFSet(S).\overline{\pi_B}(s_1) \simeq_0 \overline{\pi_B}(s_2)$,

then performing the MRP-tupling on C will terminate. □

An equation is called an *MRP-equation* if all sets of overlapping calls occurring in its RHS satisfies the conditions stated in Theorem 6.4. An F-set of mutual-recursive functions is called an F-set of *MRP-functions* if all its equations are MRP-equations.

Finally, we state the effectiveness of the MRP-tupling in eliminating redundant calls.

Theorem 6.5. (Call-Redundancy-Free of MRP-Tupled Functions)
Consider a set of mutual-recursive MRP-functions. Performing the MRP-tupling on this set will result in a new set of tupled functions which do not contain syntactic call redundancy. □

6.4. Enhanced Treatments for Accumulating Arguments

We have taken a pro-active approach in handling accumulating arguments. Earlier works in tupling either assume the absence of accumulating arguments, fix its appearance, or always generalise it during transformation [7, 16]. We have taken into consideration the construction of accumulating arguments, and involve their segments in determining synchronisation.

At present, the accumulating arguments are restricted to depend on only its (in-situ) parameter, and be specified by `acc o` $\sharp_i$ for the i`-th` parameter. However, it may also be possible for an accumulating argument to also depend on other parameter values. For instance, the accumulators of `repl/sdrop` (in Section 1), which have the form `(sdrop o (Node`$^{-1}$ `o` $\sharp_1$`,` $\sharp_2$`))`, depend on two parameters. Under this scenario, it is meaningless to perform level-3 factorisation of the accumulators. Instead, we rely on a restricted version of MRP-tupling (Theorem 6.2) which simply checks for level-4 synchronisation. In the case of `repl/sdrop`, the segments which overlap do indeed synchronise at level-4. Hence, successful MRP-tupling can be guaranteed.

There are other possible extensions to enhance the power and applicability of MRP-tupling. We refer the reader to our technical paper [10] for detail.

7. Related Work

One of the earliest mechanisms for avoiding redundant calls is memo-functions [13]. Memo-functions are special functions which remember/store some or all of their previously computed function calls in a memo-table, so that re-occurring calls can have their results retrieved from the memo-table rather than re-computed. Though general (with no analysis required), automatic memoisation can have major overheads since they rely on complex run-time machinery that may not pay-off. A recent approach to overcoming this overhead is to empower the programmer with control over the specifics of memoisation [1], including the cost of equality checking and the caching and replacement policy for memo-tables. While selective memoisation can potentially give maximal performance, there is an onus on to the programmers to achieve that.

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. By focusing on a restricted bi-linear self-recursive functions, Cohen [12] identified some algebraic properties, such as *periodic commutative*, *common generator*, and *explicit* descent relationships, to help predict redundancy patterns and corresponding tabulation schemes. Unfortunately, this approach is rather limited since the functions considered are restricted. In addition, the algebraic properties are difficult to detect, and yet limited in scope. (For example, the Tower-of-Hanoi function does not satisfy any of Cohen's algebraic properties, but can still be tupled by our method.)

Another approach is to perform *direct* search of the DG. An early work of Pettorossi [24] gave an informal heuristic to search the DG (dependency graph of calls) for eureka tuples. Later, Proietti & Pettorossi [27] proposed an Elimination Procedure, which combines fusion and tupling, to eliminate unnecessary intermediate variables from Logic programs. To ensure termination, they only handled functions with a single recursion parameter, while the accumulating parameters are generalised whenever possible. No attempt is made to analyse the synchronisability of multiple recursion/accumulating parameters.

With the aim of deriving incremental programs, Liu and her co-workers [20, 22, 21] presented a three-stage method to cache, incrementalise and prune user programs. The *caching* stage gathers all intermediate and auxiliary results which might be needed to *incrementalise*, while *pruning* removes unneeded results. In [21], they show how redundant calls can be eliminated by transforming programs into ones that employ dynamic programming technique manipulating on dynamic data structures. Their technique complements the technique described in this paper in that the latter uses more efficient static structures but is applicable to more restricted class of programs. Liu *et al.*'s method has been shown to be applicable to a wide range of computational-intensive programs. On the other hand, it requires deeper intuition. The embedding of dynamic tree-like data structures employed by their technique has an effect closer to memoisation, and this usually requires extra mechanism for space recovery. Moreover, there is not yet a characterisation of the class of programs to which the method applies.

The link between supercompilation[32] and tupling was recently explored by Secher [28], when he proposed *collapsed jungle evaluation* as the underlying evaluation semantics that has the potential for super-linear speed-up. This evaluation strategy is based on graph-based reductions where repeated nodes are re-used rather than re-computed. Secher proposed a variant of positive supercompilation [15] that works on jungles rather than terms, that has the potential to achieve tupling-like effect. To ensure termination, he proposed the use of homeomorphic-embedding [14] to ensure well-quasi-ordering. It would be of interests to see how this termination criteria compares with our more intricate synchronisation analysis, as well as the exact relationship between tupling and graph-based supercompilation.

Ensuring termination of transformers has been a central concern for many automatic transformation techniques. Though the problem of determining termination is in general undecidable, a variety of analyses can be applied to give meaningful results. In the case of deforestation, the proposals range from simple *pure treeless* syntactic form [33], to a sophisticated constraint-based analysis [29] to stop the transformation. Likewise, earlier tupling work [7, 16] were based simply on restricted functions. In [7], the transformable functions can only have a single recursion parameter each, while accumulating parameters are forbidden. Similarly, in the calculational approach of [16], functions can only have a single recursion parameter each, while the other parameters are lambda abstracted, as per [26]. Multiple recursion arguments are not considered. Furthermore, when lambda abstractions are being tupled, they yield effective elimination of multiple traversals, but *not* the effective reuse of redundant function-type calls. Thus, if functions `sdrop` and `repl` in Section 1 is lambda-abstracted prior to tupling, its redundant calls will not be properly eliminated.

8. Conclusion

There is little doubt that tupled functions are extremely useful. Apart from the elimination of redundant calls and multiple traversals, tupled function are often *linear* with respect to the common arguments (i.e. each now occurs only once in the RHS of the equation). This linearity property has a number of advantages, including:

- It can help avoid *space leaks* arising from unsynchronised multiple traversals of large data structures [31].
- It can facilitate deforestation (and other transformations) that impose a *linearity* restriction [33], often for efficiency and/or termination reasons.
- It can improve opportunity for *uniqueness typing* [3], which is good for storage overwriting and other optimisations.

Because of these nice performance attributes, functional programmers often go out of their way to write such tupled functions, despite them being more awkward, error-prone and harder to write and read.

In this paper, we show the effectiveness and safeness of an automatic tupling method, when argument change, between pairs of caller and callee, are found to be arising from a common generator, which we called level-3 synchronisation, our MRP-tupling algorithm guarantees an effective and safe transformation. The presence of common generator enables lock-step unfolding of calls to take place, which guarantees fold operation to take place after some recursive steps.

Our syncrhonisation analyses operate on sets of mutual-recursive functions, and require clear characterisation of function parameters into recursion, bound and accumulating parameters. By bringing multiple recursion arguments and accumulating arguments into one framework, we have considerably widened the scope of functions admissible for safe tupling. Consequently, the tupling algorithm and the associated synchronisation analysis could now be used to meet the run-time performance need, whilst preserving the clarity/modularity of programs.

9. Acknowledgment

We would like to thank the referees and the editors for their insightful comments. We would like to thank Tat-Wee Lee for contributing to the complexity results related to synchronisation analysis.

References

[1] Acar, U., Blelloch, G., Harper, R.: Selective Memoization, *ACM Conference on Principles of Programming Languages*, ACM Press, 2003.

[2] Augustsson, L.: Compiling Pattern-Matching, *2nd ACM Functional Programming Languages and Computer Architecture Conference*, Nancy, France, (LNCS, vol 201, pp. 368–381) Berlin Heidelberg New York: Springer, 1985.

[3] Barendsen, E., Smetsers, J.: Conventional and Uniqueness Typing in Graph Rewrite Systems, *13th Conference on the Foundations of Software Technology & Theoretical Computer Science*, Bombay, India, December 1993.

[4] Bird, R. S.: Tabulation Techniques for Recursive Programs, *ACM Computing Surveys*, **12**(4), December 1980, 403–417.

[5] Burstall, R., Darlington, J.: A Transformation System for Developing Recursive Programs, *Journal of ACM*, **24**(1), January 1977, 44–67.

[6] Chin, W.: *Automatic Methods for Program Transformation*, Ph.D. Thesis, Imperial College, University of London, March 1990.

[7] Chin, W.: Towards an Automated Tupling Strategy, *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, ACM Press, Copenhagen, Denmark, June 1993.

[8] Chin, W., Darlington, J.: A Higher-Order Removal Method, *LISP and Symbolic Computation*, **9**(4), 1996, 287–322.

[9] Chin, W., Hagiya, M.: A Bounds Inference Method for Vector-Based Memoisation, *2nd ACM SIGPLAN Intl. Conference on Functional Programming*, ACM Press, Amsterdam, Holland, June 1997.

[10] Chin, W., Khoo, S., Jones, N.: *Redundant Call Elimination via Tupling*, Technical report, Dept of Computer Science, NUS, March 2005.

[11] Chin, W., Khoo, S., Thiemann, P.: Synchronisation Analyses for Multiple Recursion Parameters, *Intl Dagstuhl Seminar on Partial Evaluation (LNCS 1110)*, Germany, February 1996.

[12] Cohen, N. H.: Eliminating Redundant Recursive Calls, *ACM Trans. on Programming Languages and Systems*, **5**(3), July 1983, 265–299.

[13] Donald, M.: Memo Functions and Machine Learning, *Nature*, **218**, April 1968, 19–22.

[14] Glück, R., Sørensen, M.: An algorithm of generalization in positive supercompilation, *Intl. Symp. on Logic Programming*, 1995.

[15] Glück, R., Sørensen, M., Jones, N.: A Positive Supercompiler, *Journal of Functional Programming*, **6**(6), 1996, 811–838.

[16] Hu, Z., Iwasaki, H., Takeichi, M., Takano, A.: Tupling Calculation Eliminates Multiple Traversals, *2nd ACM SIGPLAN International Conference on Functional Programming*, ACM Press, Amsterdam, Netherlands, June 1997.

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

[18] Jones, N., Sestoft, P., Sondergaard, H.: An Experiment in Partial Evaluation: the generation of a compiler generator, *Journal of LISP and Symbolic Computation*, **2**(1), 1989, 9–50.

[19] Lee, T.: *Synchronisation Analysis for Tupling*, Master Thesis, DISCS, National University of Singapore, 1997.

[20] Liu, Y., Stoller, S., Teitelbaum, T.: Static caching for incremental computation, *ACM Trans. Program. Lang. Syst.*, **20**(3), 1998, 546–585, ISSN 0164-0925.

[21] Liu, Y. A., Stoller, S. D.: Dynamic programming via static incrementalization, *Higher-Order and Symbolic Computation (HOSC)*, **16 (1-2)**, March 2003, 37–62.

[22] Liu, Y. A., Stoller, S. D., Teitelbaum, T.: Strengthening invariants for efficient computation, *Science of Computer Programming*, **41**(2), 2001, 139–172.

[23] Okasaki, C.: Amortization, Lazy Evaluation, and Persistence: Lists with Catenation via Lazy Linking, *IEEE Symposium on Foundations of Computer Science*, 1995.

[24] Pettorossi, A.: A Powerful Strategy for deriving Programs by Transformation, *3rd ACM LISP and Functional Programming Conference*, ACM Press, 1984.

[25] Pettorossi, A., Proietti, M.: Program Derivation via List Introduction, *IFIP TC 2 Working Conf. on Algorithmic Languages and Calculi*, Chapman & Hall, Le Bischenberg, France, February 1997.

[26] Pettorossi, A., Skowron, A.: Higher Order Generalization in Program Derivation, *TAPSOFT 87*, Pisa, Italy, (LNCS, vol 250, pp. 306–325), March 1987.

[27] Proietti, M., Pettorossi, A.: Unfolding - Definition - Folding, in this order for Avoiding Unnecessary Variables in Logic Programs, *Proceedings of PLILP*, Passau, Germany, (LNCS, vol 528, pp. 347–258) Berlin Heidelberg New York: Springer, 1991.

[28] Secher, J.: Driving in the Jungle, *Programs as Data Objects (PADO II)*, Aarhus, Denmark, May 2001.

[29] Seidl, H., Sørensen, M.: Constraints to Stop Higher-Order Deforestation, *24th ACM Symposium on Principles of Programming Languages*, ACM Press, Paris, France, January 1997.

[30] Sørensen, M., Glück, R., Jones, N.: Towards Unifying Deforestation, Supercompilation, Partial Evaluation and Generalised Partial Computation, *European Symposium on Programming (LNCS 788)*, Edinburgh, April 1994.

[31] Sparud, J.: How to Avoid Space-Leak Without a Garbage Collector, *ACM Conference on Functional Programming and Computer Architecture*, ACM Press, Copenhagen, Denmark, June 1993.

[32] Turchin, V. F.: The Concept of a Supercompiler, *ACM Trans. on Programming Languages and Systems*, **8**(3), July 1986, 90–121.

[33] Wadler, P.: Deforestation: Transforming Programs to Eliminate Trees, *European Symposium on Programming*, Nancy, France, (LNCS, vol 300, pp. 344–358), March 1988.