Synchronisation Analysis to Stop Tupling*

Wei-Ngan Chin[1], Siau-Cheng Khoo[1], and Tat-Wee Lee[2]

[1] National University of Singapore
[2] Singapore Police Academy

Abstract. Tupling transformation strategy can be used to merge loops together by combining recursive calls and also to eliminate redundant calls for a class of programs. In the latter case, this transformation can produce super-linear speedup. Existing works in deriving a safe and automatic tupling only apply to a very limited class of programs. In this paper, we present a novel parameter analysis, called *synchronisation analysis*, to solve the termination problem for tupling. With it, we can perform tupling on functions with multiple recursion *and* accumulative arguments without the risk of non-termination. This significantly widens the scope for tupling, and potentially enhances its usefulness. The analysis is shown to be of polynomial complexity; this makes tupling suitable as a compiler optimisation.

1 Introduction

Source-to-source transformation can achieve global optimisation through specialisation for recursive functions. Two well-known techniques are *partial evaluation* [9] and *deforestation* [20]. Both techniques have been extensively investigated [18, 8] to discover automatic algorithms and supporting analyses that can ensure correct and terminating program optimisations.

Tupling is a lesser 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 simultaneously. When successfully applied, redundant calls can be eliminated, and multiple traversals of data structures combined.

As an example, consider the Tower of Hanoi function.

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

Note that $+\!\!+$ denotes list catenation. 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. 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 Sec 2.) This definition contains *redundant calls*, which can be eliminated. By gathering each set of overlapping calls, which share common recursion arguments into a tupled function, the tupling method introduces:

* This work was done while Lee was a recipient of a NUS Graduate Scholarship.

```
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)) ;
```

and transforms *hanoi* to the following :

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

Despite a significant loss in modularity and clarity, the resulting tupled function is *desirable* as their better performance can be mission critical. Sadly, manual and error-prone coding of such tupled functions are frequently practised by functional programmers. 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 knapsack definition below, where *W(i)* and *V(i)* return the weight and value of some *i*-th item.

```
knap(0,w)      = 0 ;
knap(1+n,w)    = if w < W(1+n) then knap(n,w)
                   else max(knap(n,w),knap(n,w-W(1+n))+V(1+n) ;
```

Redundant calls exist but tupling fails to stop when it is performed on the above function. Specifically, tupling encounters the following growing tuples of calls.

1. (knap(n,w),knap(n,w-W(1+n)))
2. (knap(n_1,w),knap(n_1,w-W(1+n_1)),knap(n_1,w-W(2+n_1)),
   knap(n_1,w-W(2+n_1)-W(1+n_1)))
3. (knap(n_2,w),knap(n_2,w-W(1+n_2)),knap(n_2,w-W(2+n_2)-W(1+n_2)),
   knap(n_2,w-W(2+n_2)),knap(n_2,w-W(3+n_2)),knap(n_2,w-W(3+n_2)-W(1+n_2)),
   knap(n_2,w-W(3+n_2)-W(2+n_2)),knap(n_2,w-W(3+n_2)-W(2+n_2)-W(1+n_2)))

$\vdots$

Why did tupling failed to stop in this case? It was because the calls of *knap* *overlap*, but their recursion parameter *n* did not *synchronise* with an accumulative parameter *w*.

To avoid the need for parameter synchronisation, previous proposals in [2, 7] restrict tupling to only functions with a single recursion parameter, and without any accumulative parameters. However, this blanket restriction also rules out many useful functions with multiple recursion and/or accumulative 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 $\mathbf{O}(n^2)$ where *n* is the size of the input tree. The two functions each has the first parameter being recursion and the second being accumulative. For the calls which overlap, the two parameters synchronise with each other (see Sec. 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 to *rstup* yields the following $\mathbf{O}(n)$ 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) ;

This paper proposes a novel parameter analysis, called *synchronisation analysis*, to solve the termination problem for tupling. With it, we can considerably widen the scope of tupling by selectively handling functions with multiple recursion (and/or accumulative) parameters. If our analysis shows that the multiple parameters synchronises for a given function, we guarantee that tupling will stop when it is applied to the function. However, if our analysis shows possible non-synchronisation, we must skip tupling. (This failure may be used to suggest more advanced but expensive techniques, such as *vector-based* [3] or *list-based* [14] memoisations. These other techniques are complimentary to tupling since they may be more widely applicable but yield more expensive codes.) Lastly, we provide a costing for our analysis, and show that it can be practically implemented.

In Sec. 2, we lay the foundation for the discussion of tupling transformation and synchronisation analysis. Sec. 3 gives an overview of the tupling algorithm and the two obstacles towards terminating transformation. Sec. 4 provides a formal treatment of segments, which are used to determine synchronisation. In Sec. 5, we formulate prevention of indefinite unfolding via investigation of the behaviour of segment concatenation. In Sec. 6, we formally introduce synchronisation analysis, and state the conditions that ensure termination of tupling transformation. Sec. 7 describes related work, before a short conclusion in Sec. 8. Due to space constraint, we refer the reader to [4] for more in-depth discussion of related issues.

2 Language and Notations

We consider a strict first-order functional language.

Definition 1 (A Simple Language). A program in our simple language consists of sets of mutual-recursive functions:

$$
\begin{array}{lll}
P & ::= [M_i]_{i=0}^{n} & \text{(Program)} \\
M & ::= [F_i]_{i=0}^{n} & \text{(Set of Mut. Rec. Fns)} \\
F & ::= \{f(p_{i1},\ldots,p_{in}) = t_i\}_{i=0}^{m} & \text{(Set of Equations)} \\
t & ::= v \mid C(t_1,\ldots,t_n) \mid f(t_1,\ldots,t_n) & \text{(Expression)} \\
 & \quad \mid \mathit{if}\ t_1\ \mathit{then}\ t_2\ \mathit{else}\ t_3 \mid \mathit{let}\ \{p_i = t_i\}_{i=0}^{n}\ \mathit{in}\ t & \\
p & ::= v \mid C(p_1,\ldots,p_n) & \text{(Pattern)}
\end{array}
$$

We allow use of infix binary data construction in this paper. Moreover, we shall construct any Peano integer as a data constructed from 0 and $1{+}n$. Applying the constructor, $(1{+})$, k times to a variable m will be abbreviated as $k{+}m$.

For clarity, we adopt the following multi-holes context notation :

Definition 2 (Multi-holes Context Notation). The RHS of an equation of a function f can be expressed as $E_f[t_1,\ldots,t_n]$ where $t_1, \ldots, t_n$ are sub-expressions occurring in the RHS. □

Safety of tupling transformation 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 3 (Argument Operators).

1. Each data constructor C of arity n in the language is associated with n *descend operators*, which are data destructors. Notation-wise, let $C(t_1,\ldots,t_n)$ be a data structure, then any of its corresponding data destructors is denoted by C^{-i}, and defined as $C^{-i}\ C(t_1,\ldots,t_n) = t_i$.
2. For each constant (i.e. variable-free constructor subterm), denoted by c in our program, a *constant operator* $\underline{c}$ always return that constant upon application.
3. An *identity*, id, is the unit under function composition.
4. For any n-tuple argument $(a_1,\ldots,a_n)$, the i^{th} *selector*, $\sharp_i$, is defined as $\sharp_i\ (a_1,\ldots, a_n) = a_i$.
5. An *accumulative operator* is any operator that is not defined above. For instance, a data constructor, such as C described in item 1 above, is an accumulative operator; and so is the tuple constructor $(op_1,\ldots,op_n)$. □

Composition of operators is defined by $(f \circ g)\ x = f\ (g\ x)$. 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 $g(x) = E_g[g(C(x,2))]$. $C(x,2)$ in the RHS can be constructed from parameter x via the operation path : $C \circ (id, \underline{2})$.

Changes in an n-tuple argument can be described by an n-tuple of operation paths, called a *segment*, and denoted by $(op_1,\ldots,op_n)$. 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 a function graph to show how the function arguments are transformed:

Definition 4 (Labelled Call Graph). The *labelled call graph* of a set of mutual-recursive functions F, denoted as (N_F, E_F), is a graph whereby each function name from F is a node in N_F; and each caller-callee transition is represented by an arrow in E_F, labelled with the segment information. □

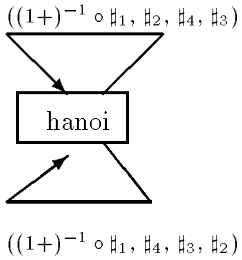

Fig. 1. Labelled Call Graph of *Hanoi* defined in Sec. 1.

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

Definition 5 (Characterising Parameters/Arguments). Given an equation of the form $f(p_1, \ldots, p_n) = t$,

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 *accumulative parameter*. □

Correspondingly, an argument to a function call is called a *recursion/accumulative argument* if it is located at the position of a recursion/accumulative parameter.

We can partition a segment according to the kinds of parameters it has:

Definition 6 (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/accumulative/bounded parameters. The sub-tuple preserves the original ordering of the operation paths in the segment.

Furthermore, we write $\overline{\pi_B}(s)$ to denote the sub-tuple of s excluding $\pi_B(s)$. □

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* (defined in Sec. 1 too), both their first parameters are recursion parameters, and their second parameters are accumulative.

Our analysis of segments requires us to place certain restrictions on the parameters and its relationship with the arguments. This is described as follows:

Definition 7 (Restrictions on Parameters).

1. Each set of mutual-recursive functions (incl. recursive auxiliary functions), M, has the same number of recursion/accumulative parameters but can have an arbitrary number of bounded parameters.
2. Given an equation from M, the i^{th} recursion/accumulative 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/accumulative parameters. Though restrictive, these requirements can be selectively lifted by pre-processing transformation and/or improved analysis. Due to space constraint, the details are described in a companion technical report [4].

3 Tupling Algorithm

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 8 (Call Overlapping).

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

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

If two calls overlap, the call graphs initiated from them may overlap, and thus contain redundancy. Hence, it is useful, during tupling transformation, to gather the overlapping calls into a common function body with the hope that redundant calls (if any) will eventually be detected and eliminated (via abstraction). Once these redundant calls have been eliminated, what's left behind in the RHS will be disjoint calls.

Applying tupling on a function thus attempts to transform the function into one in which every pair of calls in the new RHS are disjoint. Fig. 2 gives an operational description of the tupling algorithm.[1]

There are two types of unfolds in the tupling algorithm: In Step 4.2, calls are *unfolded without instantiation*; ie., a call is unfolded only when all its arguments matches the LHS of an equation. On the other hand, in Step 4.4.2.2, a call is selected and forced to unfold. This henceforth requires instantiation. Among the

[1] Please refer to [10] for detail presentation.

1. Decide a set of recursive function calls to tuple, C.
2. Let E contains a set of equations (with C calls in their RHS) to transform.
3. Let D be empty. (D will be the set of tuple definitions.)
4. While E is not empty,
4.1. Remove an equation, say $f(p_1, \ldots, p_n) = t$, from E.
4.2. Modify $t \Rightarrow t_1$ by repeatedly *unfolding without instantiation* each function call to C in t.
4.3. Gather each subset of overlapping C calls, s_i, and perform the following *tuples abstraction steps*:
4.3.1. For each tuple s_i, abstract the common substructure, $t_{i,j}$, occurring in the accumulative arguments of all calls in s_i, and modify $s_i \Rightarrow let\ \{v_{i,j} = t_{i,j}\}_{j=1}^{k_i}\ in\ s_i'$.
4.3.2. Modify $t_1 \Rightarrow let\ \bigcup_{i=1}^{m} \{v_{i,j} = t_{i,j}\}_{j=1}^{k_i}\ in\ let\ \{(v_{i,1}, \ldots, v_{i,n_i}) = s_i'\}_{i=1}^{m}\ in\ t_2$.
4.4. For each tuple s_i' from $\{s_i'\}_{i=1}^{m}$, we transform to s_i'' as follows : Let $\{v_1, \ldots, v_n\} = Vars(s_i')$ where $Vars(e)$ returns free variables of e
4.4.1. If s_i' has ≤ 1 function call, then $s_i'' = s_i'$. (No action taken.)
4.4.2. If there is no tuple definition for s_i' in D, then
4.4.2.1. Add a new definition, $g(v_1, \ldots, v_n) = s_i'$ to D.
4.4.2.2. Instantiate g by *unfolding (with instantiation)* a call in s_i' that has maximal recursion argument. This introduces several new equations for g, which are then added to E.
4.4.2.3. Set $s_i'' = g(v_1, \ldots, v_n)$.
4.4.3. Otherwise, a tuple definition, of name say h, matching s_i' is found in D.
4.4.3.1. *Fold* s_i' against definition of h to yield a call to h, by setting $s_i'' = h(v_1, \ldots, v_n)$.
4.5. Output a transformed equation $f(p_1, \ldots, p_n) = let\ \bigcup_{i=1}^{m} \{v_{i,j} = t_{i,j}\}_{j=1}^{k_i}\ in\ let\ \{(v_{i,1}, \ldots, v_{i,n_i}) = s_i''\}_{i=1}^{m}\ in\ t_2$.
5. Halt.

Fig. 2. Tupling Algorithm, $\mathcal{T}$

calls available, we choose to unfold a call having *maximal recursion* arguments; that is, the recursion arguments, treated as a tree-like data structure, is deepest in depth among the calls.

Although effective in eliminating redundant calls, execution of algorithm $\mathcal{T}$ may not terminate in general due to one of the following reasons: (1) Step 4.2 may unfold calls indefinitely; (2) Step 4.4.2 may introduce infinitely many new tuple definitions as tupling encounters infinitely many different tuples.

We address these two termination issues in Sec. 5 and Sec. 6 respectively. But first, we give a formal treatment of algebra of segments.

4 Algebra of Segments

A set of segments forms an algebra under concatenation operation.

Definition 9 (Concatenation of Operation Paths). Concatenation of two operation paths *op1* and *op2*, denoted by *op1* ; *op2*, is defined as *op2* $\circ$ *op1*. □

Definition 10 (Concatenation of Segments). Concatenation of two segments *s1* and *s2*, denoted by *s1*;*s2*, is defined componentwise as follows:

$$s1 = (\ op_1,\ldots,op_n\ )\ \&\ s2 = (\ op'_1,\ldots,op'_n\ ) \Rightarrow s1\ ;\ s2 = (\ op_1;op'_1,\ldots,op_n;op'_n\ ).$$ □

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

1. For any operator O, $id \circ O$ reduces to O, and $O \circ id$ 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*. Henceforth, we deal with compacted segments, unless we state 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 11 (Factorisation of Segments). Given segments s and f. f is said to be a *factor* of s if (1) f is a substring of s, and (2) $\exists n > 0.\ s = f^{\,n}$.
We call n the *power* of s wrt f. □

For example, (C^{-2}, id) is a factor of $(C^{-2}\circ C^{-2}\circ C^{-2}, id\ )$, since $(C^{-2}, id\ )^3 = (C^{-2}\circ C^{-2}\circ C^{-2}, id\ )$. It is trivial to see that every segment has at least one factor – itself. However, when a segment has *exactly one* factor, it is called a *prime segment*. An example of prime segment is $(C_1^{-1}\circ C_2^{-1}, id\ )$.

Lemma 12 (Uniqueness of Prime Factorisation[10]). *Let s be a compacted segment, there exists a* unique *prime segment f such that* $s = f^{\,k}$ *for some* $k>0$. □

5 Preventing Indefinite Unfolding

We now provide a simple condition that prevents tupling transformation from admitting indefinite unfolding at Step 4.2. This condition has been presented in [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. 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 $\mathbf{O}(|N||E|^2)$ [10].

Theorem 13 (Preventing Indefinite Unfolding). *Let F be a set of mutual-recursive functions, each of which has* non-zero number of recursion parameters*. If* $\forall\ s \in SCycle(N_F,E_F)$, $\pi_R(s) \neq (id,\ldots,id)$, *then given a call to* $f \in F$ *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.* □

6 Synchronisation Analysis

We now present analysis that prevents tupling from generating infinitely many new tuple functions at Step 4.4.2. The idea is to ensure the finiteness of syntactically different (modulo variable renaming) tupled calls. As a group of bounded arguments is 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 tupling transformation to loop infinitely. Hence, we focus on determining the structure of recursion and accumulative arguments in this section. Specifically, *we assume non-existence of 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 14 (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 $s_1 = s_2$. □

Levels 1 to 4 of synchronisation form a strict hierarchy, with synchronisation at level i implying synchronisation at level j if $i > j$. Together with level-0, these can help identify the termination property of tupling transformation.

Why does synchronisation play an important role in the termination of tupling transformation? 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 Step 4.4.3) to take effect, and eventually terminates the transformation.

In this section, we provide an informal account of some of the interesting findings pertaining to tupling termination, 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 15. Consider the following two equations for functions *g1* and *g2* respectively:

[2] Sometimes, two apparently level-0 synchronised may turn into synchronisation of other levels when they are prefixed with some initial segment. 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" [5].

$g1(C_1(x_1,x_2),C_2(y_1,y_2)) \quad = E_{g1}[g1(x_1,y_1)]$;
$g2(C_1(x_1,x_2),C_2(y_1,y_2)) \quad = E_{g2}[g2(x_1,y_2)]$;

The segment leading to call $g1(x_1,y_1)$ is (C_1^{-1},C_2^{-1}), whereas that leading to call $g2(x_1,y_2)$ is (C_1^{-1},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. Tupling-transform these two calls causes the definition of a tuple function:

$g\_tup(u,v) = (\ g1(u,v),\ g2(u,v)\ )$;

which will then transform (through instantiation) to the following:

$g\_tup(C_1(u_1,u_2),C_2(v_1,v_2)) = (E_{g1}[g1(u_1,v_1)],E_{g2}[g2(u_1,v_2)])$;

As the arguments of the two calls in the RHS above are now disjoint, tupling terminates. However, the effect of tupling is simply an unfolding of the calls. Thus, it is safe[3] but not productive to transform two calls with identical arguments if these calls follow segments that are level-0 synchronised. □

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

Example 16. Consider the binomial function, as defined below:

$bin(0,k) \quad = 1$;
$bin(1{+}n,0) \quad = 1$;
$bin(1{+}n,1{+}k) = \textit{if } k \geq n \textit{ then } 1 \textit{ else } bin(n,k)+bin(n,1{+}k)$;

Redundant call exists in executing the two overlapping calls $bin(n,k)$ and $bin(n,1{+}k)$. Notice that the segments leading to these calls $(((1+)^{-1},(1+)^{-1})$ and $((1+)^{-1},id))$ are level-2 synchronised. Performing tupling transformation on $(bin(n,k),bin(n,1{+}k))$ will keep generating new set of overlapping calls at Step 4.4.2.2, as shown below:

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

$\vdots$

Hence, tupling transformation fails to terminate. □

Non-termination of transforming functions such as *bin* can be predicted from the (non-)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 they overlap).

However, if two segments are level-3 synchronised, then it is always possible to build from these two segments, two sequences that are identical; thanks to the following Prop. 17(a) about level-3 synchronisation.

Property 17 (Properties of Level-3 Synchronisation).

[3] with respect to termination of tupling transformation.

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

This, thus, provides an opportunity for termination of tupling transformation. Indeed, the following theorem highlights such an opportunity.

Theorem 18 (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_F,E_F). Let C be an initial set of overlapping F-calls to be tupled. If*

1. $\forall\ s \in SCycle(N_F,E_F)\ .\ \pi_R(s) \neq (id,\ldots,id)$,
2. $\forall\ s_1, s_2 \in S\ .\ \overline{\pi_B}(s_1) \simeq_3 \overline{\pi_B}(s_2)$.

then performing tupling transformation on C terminates. □

The notion $\overline{\pi_B}(s)$ was defined in Defn. 6. The first condition in Theorem 18 prevents infinite number of unfolding, whereas the level-3 synchronisation condition ensures that the number of different tuples generated during transformation is finite. The proof is available in [4].

Example 19. Consider the equation of f defined below:

$f(2{+}n,4{+}m,y) = E_f[f(1{+}n,2{+}m,C(y)),f(n,m,C(C(y)))]$;

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

$f(2{+}n,4{+}m,y) \quad = let\ \{y1 = C(y)\}\ in\ let\ \{(u,v) = f\_tup(n,m,y1)\}\ in\ E_f[u,v]$;
$f\_tup(1{+}n,2{+}m,y) = let\ \{y2 = C(y)\}\ in\ let\ \{(u,v) = f\_tup(n,m,y2)\}\ in\ (E_f[u,v],u)$;

□

Finally, since level-4 synchronisation implies level-3 synchronisation, *Theorem 18 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 tupling transformation. To begin with, we observe from Prop. 17(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. 17(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 $HCF([s_i]) = f_i^{gcd(p_1,\ldots,p_{ni})}$, where gcd computes the greatest common divisor. $HCF([s_i])$ is thus the *highest common factor* of the level-3 set $[s_i]$.

Definition 20 (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 during tupling transformation[4]

Theorem 21 (Preventing Infinite Definition). *Let F be a set of mutual-recursive functions. Let S be the set of segments corresponding to the edges in (N_F, E_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 tupling transformation on C will generate a finite number of different tuples.* □

A proof of the theorem is available in [10, 4].

A note on the complexity of this analysis: We notice from Theorem 21 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.

Theorem 22 summarises the results of Theorem 13 and Theorem 21.

Theorem 22 (Termination of Tupling Transformation). *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 correspond to the edges in (N_F, E_F). Let C a set of overlapping calls occurring in the RHS of an equation in F. If*

1. $\forall \ s \in SCycle(N_F, E_F) \ . \ \pi_R(s) \neq (id, \ldots, id)$, *and*
2. $\forall \ s_1, s_2 \in HCFSet(S) \ . \ \overline{\pi_B}(s_1) \simeq_0 \overline{\pi_B}(s_2)$,

then performing tupling transformation on C will terminate. □

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), memo-functions are less practical since they rely on expensive run-time mechanisms.

[4] It is possible to extend Theorem 21 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 [10].

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 [6] 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. Pettorossi [16] gave an informal heuristic to search the DG (dependency graph of calls) for eureka tuples. Later, Proietti & Pettorossi [17] 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 accumulative parameters are generalised whenever possible. No attempt is made to analyse the synchronizability of multiple recursion/accumulating parameters.

With the aim of deriving incremental programs, Liu and Teitelbaum [12, 11] presented a three-stage method to cache, incrementalize and prune user programs. The *caching* stage gathers all intermediate and auxiliary results which might be needed to *incrementalize*, while *pruning* removes unneeded results. While their method may be quite general, its power depends largely on the incrementalize stage, which often requires heuristics and deep intuitions. No guarantee on *termination*, and the *scope* of this difficult stage is presently offered.

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 [20], to a sophisticated constraint-based analysis [18][5] to stop the transformation. Likewise, earlier tupling work [2, 7] were based simply on restricted functions. In [2], the transformable functions can only have a single recursion parameter each, while accumulative parameters are forbidden. Similarly, in the calculational approach of [7], the allowed functions can only have a single recursion parameter each, while the other parameters are lambda abstracted, as per [15]. When lambda abstractions are being tupled, they yield effective elimination of multiple traversals, but *not* effective elimination of redundant function-type calls. For example, if functions *sdrop* and *repl* are lambda-abstracted prior to tupling, then redundant calls will not be properly eliminated.

By engaging a more powerful synchronisation analysis for multiple parameters, we have managed to extend considerably the class of functions which could be tupled safely. Some initial ideas of our synchronisation analysis can be found in [5]. The earlier work is informal and incomplete since it did not cover *accumu-*

[5] from which the title of this paper was inspired

lative parameters. The present proposal is believed to be comprehensive enough to be practical.

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* that are due to unsynchronised multiple traversals of large data structures, via a compilation technique described in [19].
– It can facilitate deforestation (and other transformations) that impose a *linearity* restriction [20], often for efficiency and/or termination reasons.
– It can improve opportunity for *uniqueness typing* [1], 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 have shown the effectiveness and safeness of tupling transformation, when coupled with synchronisation analyses. Furthermore, not only do we generalise the analyses from handling of single recursion argument to that of multiple recursion arguments, we also bring together both recursion and accumulative arguments under one framework for analysis. These have considerably widened the scope of functions admissible for safe tupling. Consequently, the tupling algorithm $\mathcal{T}$ and associated synchronisation analysis could now be used to improve run-time performance, whilst preserving the clarity/modularity of programs.

Acknowledgments We thank the annonymous referees for their valuable comments and Peter Thiemann for his contribution to earlier work.

References

1. E. Barendsen and J.E.W. Smetsers. Conventional and uniqueness typing in graph rewrite systems. In *13th Conference on the Foundations of Software Technology & Theoretical Computer Science*, pages 45–51, Bombay, India, December 1993.
2. Wei-Ngan Chin. Towards an automated tupling strategy. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 119–132, Copenhagen, Denmark, June 1993. ACM Press.
3. Wei-Ngan Chin and Masami Hagiya. A bounds inference method for vector-based memoisation. In *2nd ACM SIGPLAN Intl. Conference on Functional Programming*, pages 176–187, Amsterdam, Holland, June 1997. ACM Press.

4. W.N. Chin, S.C. Khoo, and T.W. Lee. Synchronisation analysis to stop tupling – extended abstract. Technical report, Dept of IS/CS, NUS, Dec 1997. `http://www.iscs.nus.sg/~khoosc/paper/synTech.ps.gz`.
5. W.N. Chin, S.C. Khoo, and P. Thiemann. Synchronisation analyses for multiple recursion parameters. In *Intl Dagstuhl Seminar on Partial Evaluation (LNCS 1110)*, pages 33–53, Germany, February 1996.
6. Norman H. Cohen. Eliminating redundant recursive calls. *ACM Trans. on Programming Languages and Systems*, 5(3):265–299, July 1983.
7. Z. Hu, H. Iwasaki, M. Takeichi, and A. Takano. Tupling calculation eliminates multiple traversals. In *2nd ACM SIGPLAN International Conference on Functional Programming*, pages 164–175, Amsterdam, Netherlands, June 1997. ACM Press.
8. N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.
9. N.D. Jones, P. Sestoft, and H. Sondergaard. An experiment in partial evaluation: the generation of a compiler generator. *Journal of LISP and Symbolic Computation*, 2(1):9–50, 1989.
10. Tat Wee Lee. Synchronisation analysis for tupling. Master's thesis, DISCS, National University of Singapore, 1997. `http://www.iscs.nus.sg/~khoosc/paper/ltw_thesis.ps.gz`.
11. Y A. Liu, S D. Stoller, and T. Teitelbaum. Discovering auxiliary information for incremental computation. In *23rd ACM Symposium Principles of Programming Languages*, pages 157–170, St. Petersburg, Florida, January 1996. ACM Press.
12. Y A. Liu and T. Teitelbaum. Caching intermediate results for program improvement. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 190–201, La Jolla, California, June 1995. ACM Press.
13. Donald Michie. Memo functions and machine learning. *Nature*, 218:19–22, April 1968.
14. A. Pettorossi and M. Proietti. Program derivation via list introduction. In *IFIP TC 2 Working Conf. on Algorithmic Languages and Calculi*, Le Bischenberg, France, February 1997. Chapman & Hall.
15. A. Pettorossi and A. Skowron. Higher order generalization in program derivation. In *TAPSOFT 87*, Pisa, Italy, (LNCS, vol 250, pp. 306–325), March 1987.
16. Alberto Pettorossi. A powerful strategy for deriving programs by transformation. In *3rd ACM LISP and Functional Programming Conference*, pages 273–281. ACM Press, 1984.
17. M. Proietti and A. Pettorossi. Unfolding - definition - folding, in this order for avoiding unnecessary variables in logic programs. In *Proceedings of PLILP*, Passau, Germany, (LNCS, vol 528, pp. 347–258) Berlin Heidelberg New York: Springer, 1991.
18. H. Seidl and M.H. Sørensen. Constraints to stop higher-order deforestation. In *24th ACM Symposium on Principles of Programming Languages*, Paris, France, January 1997. ACM Press.
19. Jan Sparud. How to avoid space-leak without a garbage collector. In *ACM Conference on Functional Programming and Computer Architecture*, pages 117–122, Copenhagen, Denmark, June 1993. ACM Press.
20. Phil Wadler. Deforestation: Transforming programs to eliminate trees. In *European Symposium on Programming*, Nancy, France, (LNCS, vol 300, pp. 344–358), March 1988.