Effective Optimisation of Multiple Traversals in Lazy Languages

Wei-Ngan CHIN   Aik-Hui GOH   Siau-Cheng KHOO

School of Computing

National University of Singapore

Abstract

Tupling transformation strategy can be applied to eliminate *redundant calls* in a program and also to eliminate *multiple traversals* of data structures. While the former application can produce super-linear speedup in the transformed program, the effectiveness of the latter has yet to be appreciated. In this paper, we investigate the pragmatic issues behind elimination of multiple data traversal in the context of lazy languages, and propose a framework of tupling tactic called *strictness-guided tupling*. This tactic is capable of exploiting specialised strictness contexts where possible to effect tupling optimisation. Two further enhancements of the tupling tactic are also proposed. One achieves *circular tupling* when multiple traversals from nested recursive calls are eliminated. The other exploits *speculative strictness* to further improve the performance of tupling. Benchmarks are given throughout the paper to illustrate the performance gains due to these tactics.

Keywords: *Tupling, Multiple Traversals, Strictness, Circular Programs, Speculation.*

1 Introduction

Tupling transformation strategy can be applied to eliminate redundant calls in a program and to eliminate multiple traversals of data structures. While the speed up gained from redundant calls elimination is undisputed, the effectiveness of eliminating multiple data-structure traversals has been largely ignored.

Consider the following function, `av` , which traverses a list twice:

```
av xs  = let {u = sum xs; v = len xs} in  u / v
sum xs = case xs of
         []     -> 0
         (y:ys) -> let {u = sum ys} in y + u
len xs = case xs of
         []     -> 0
         (z:zs) -> let {v = len ys} in 1 + v
```

Application of tupling to `av` returns a function that eliminates double traversal of the input list by `sum` and `len` , as follows:

```
av xs = let {(u,v) = avtup xs} in u / v
avtup xs = case xs of
           []     -> (0,0)
           (y:ys) -> let {(u,v)  = avtup ys
                          u' = y + u
                          v' = 1 + v}
                     in (u', v')
```

Even though multiple traversals are eliminated, the tupled program performs significantly *worse* than the original program, particularly under *lazy evaluation*. This is shown in the first two rows of Tab. 1.

	Heap (bytes)	Time(s)			
		INIT	MUT	GC	Total
Before Tup.	24,207,372	0.02	8.58	0.21	8.81
After Tup.	60,209,100	0.04	15.31	2.14	17.49
New Tup.	16,208,668	0.05	5.63	0.07	5.75

Table 1: Measurement for executing 100 times of `av [1..10000]` under Glasgow GHC 0.29

In this paper, we propose an enhanced tupling tactic that *yields practical effectiveness.* The new tactic uses strictness information of the subject program to guide the transformation. We call it *strictness-guided tupling.* It transforms `av` function to the following `avtup'` program:

```
av xs = let! {(u,v) = avtup' xs} in u / v
avtup' xs = case xs of
            []     -> (0,0)
            (y:ys) -> let! {(u,v)  = avtup' ys
                            u' = y + u
                            v' = 1 + v}}
                      in    (u', v')
```

Notice that `let!` has been used in place of `let`. This forces the local declarations of `let!` to be evaluated strictly. It can help in two ways. Firstly, tuple components, such as `u'` and `v'`, can be evaluated strictly; alleviating the need for their closures to be built. Secondly, the tuple result itself, e.g. `(u',v')`, can be strictly evaluated and be returned via the stack (or registers), instead of being constructed in the heap. (Some strict languages, such as Moscow ML, do not presently have this capability of returning tupled results via the stack. As a result, multiple traverals optimisation does not work properly for them.) With these optimisations, `avtup'` runs about 35% faster as shown in the last row of Tab. 1. Note that this improvement is due solely to elimination of multiple traversals. To isolate the effect of strictness optimisation, the original code used in the first row of Tab. 1 was also subjected to strictness analysis before its performance was measured.

Based on this strictness-guided tupling tactic, we investigate the use of strictness information to further improve the transformed programs, as well as to systematically generate circular programs. The key contributions of our paper are as follows:

- We provide the first pragmatic evidence on the usefulness of *tupling tactic for eliminating multiple traversals*. This task is particularly hard for lazy languages, as the extra cost incurred by tupled programs easily negate the gain from the elimination of multiple traversals. We show how meaningful gain can be achieved via a strictness-guided tupling algorithm.
- We advocate the use of an advance but yet practical strictness analysis (with support for strictness of recursive types) for our tupling tactic. While many advanced strictness analysis have been proposed in the literature[Bur91, Wad87], there have been few practical justifications for their adoption. This has led a number of researchers to believe that simple strictness is sufficient for most programs[PJP93]. We show how our *tupling tactic guided by an advanced strictness analysis can give new impetus to both techniques* - through more opportunities for optimisation.
- We highlight a novel use of tupling due to [Bir84] that results in tupled circular programs. A systematic way to incorporate *circular tupling* into our framework is proposed, providing an opportunity for Bird's technique to be automated.
- We propose a new analysis framework for *speculative strictness* and show how it could be utilised to enhance tupling.

The rest of the paper is organised as follows: Section 2 describes the language syntax; Section 3 gives an overview of basic tupling transformation. We discuss the use of strictness information during tupling, and present strictness-guided tupling in Section 4. In Section 6, we derive circular programs through tupling. This is followed by a proposal for speculative strictness to be used with conventional strictness to support more aggressive tupling (Section 7). Finally, we raise some important issues for discussion in Section 8, before concluding the paper.

2 Language

Our subject programs are written in a first-order, typed, lazy functional language (Fig. 1), similar to the intermediate languages of *practical functional compilers*. `let` statements are non-recursive. `let!` statements are used to introduce strict evaluation of abstracted expressions. `letrec` statements enable mutual-recursive definitions.

We also represent an expression `e` as $\mathcal{C}\langle e_1,\ldots,e_n\rangle$ to convey the idea that 'somewhere inside expression `e` lies the sub-expressions e_1, …, e_n'. This allows us to look into nested sub-expressions without being too bothered by unnecessary contextual details.

The class of functions which will be subject to tupling transformation are known as *SRP functions* [Chi93]. These are functions with *single recursion parameter*. A recursion parameter has type of a recursive data structure. Argument bound to such recursion parameter are guaranteed to reduce in size when the associated function calls are unfolded. Examples of SRP functions include `len`, `sum` , `av` , as well as `rev` and `sumseg` , which will be defined in due course. For convenience, we place the recursion parameter at the first position of an SRP function, and write it as `xs` whenever possible.

3 Basic Tupling Algorithm, an Overview

The basic tupling algorithm proposed in [Chi93] can be used to eliminate both multiple traversals and redundant calls for strict languages. The algorithm is constructed based on the fold/unfold rules of [BD77]. Prior to transformation, the system determines a class of SRP functions, *SRPSet*, whose calls are to be gathered. In Fig. 2, we highlight the main components of the algorithm by listing five syntax-directed rules, B1, …, B5.

Rules B1 and B2 merely skip over outer `let` and `case` expressions to search inside their sub-expressions for set of calls which could be tupled. The main rules of tupling are:

- B3 - To float out inner `let` abstraction, so that calls located in the `let` body with locally-defined variables can be collected for tupling. We forbid floating of `let` abstraction which contains calls to functions in *SRPSet*, so that the transformer does not miss the opportunity for collecting those calls. Implicitly, an expression can only be floated within the the binding scope of all of its free variables, and variables are renamed whenever necessary to avoid name clash.

  We allow liberal application of floating. Arbitrary floating of `let` abstraction may lead to unnecessary closures being built. Although we do not do it for the code presented in this paper, we can apply the `let` "float-in" technique of [PJPS96] to post-process the code after tupling.
- B4 - To gather multiple calls with common recursion arguments together to form a new tuple function, followed by unfolding of the calls. Gathering of calls provides an opportunity for redundant calls to be shared, and also facilitates rule B5.

$$
\begin{array}{lll}
e & ::= & k \mid v \mid f\ (v_1,\ldots,v_n) \mid op\ (v_1,\ldots,v_n) \mid c\ (v_1,\ldots,v_n) \mid \texttt{case}\ e_0\ \texttt{of}\ \{c_i\ v_{i1}\ldots v_{in}\rightarrow e_i\}_i \\
 & & \mid\ \texttt{let}\ \{(v_{1i},\ldots,v_{ni}) = e_i\}_{i\in M}\ \texttt{in}\ e \mid\ \texttt{let!}\ \{(v_{1i},\ldots,v_{ni}) = e_i\}_{i\in M}\ \texttt{in}\ e \\
 & & \mid\ \texttt{letrec}\{(v_{1i},\ldots,v_{ni}) = e_i\}_{i\in M}\ \texttt{in}\ e \\
p & \in & \mathit{Program} \\
 & & p\ ::= \{\ f_i\ (v_{i1},\ldots,v_{ik})\ =\ e_i\}_i
\end{array}
$$

Context Notation :

$$
\begin{array}{lll}
\mathcal{C}\langle\rangle & = & \#m \mid k \mid v \mid \texttt{f}\ (\mathcal{C}\langle\rangle_1,\ldots,\mathcal{C}\langle\rangle_n) \mid \texttt{op}\ (\mathcal{C}\langle\rangle_1,\ldots,\mathcal{C}\langle\rangle_n) \\
 & \mid & \texttt{c}\ (\mathcal{C}\langle\rangle_1,\ldots,\mathcal{C}\langle\rangle_n) \mid\ \texttt{let}\ \{v_i = \mathcal{C}\langle\rangle_i\}_{i\in M}\ \texttt{in}\ \mathcal{C}\langle\rangle \\
 & \mid & \texttt{case}\ \mathcal{C}\langle\rangle_0\ \texttt{of}\ \{c_i\ v_{i1}\ldots v_{in}\rightarrow \mathcal{C}\langle\rangle_i\}_i
\end{array}
$$

where # is a special variable known as a hole, labelled with a number m.

Figure 1: First-order, typed, lazy functional language

B1 (Skip outer `let`)

$$\mathcal{T}_B\ ds\ [\texttt{let}\ \{(v_1,\ldots,v_n) = (t_1,\ldots,t_n)\}\ \texttt{in}\ t] \ \Rightarrow\ \texttt{let}\ \{(v_1,\ldots,v_n) = (t'_1,\ldots,t'_n)\}\ \texttt{in}\ \mathcal{T}_B\ ds\ [t]$$
$$\textit{where } t'_i = \mathcal{T}_B\ ds\ [t_i] \quad \forall i \in 1..n$$

B2 (Skip outer `case`)

$$\mathcal{T}_B\ ds\ [\texttt{case}\ t\ \texttt{of}\ \{\ p_i\ \texttt{->}\ t_i\ \}_{i\in N}\ ] \ \Rightarrow \texttt{case}\ \mathcal{T}_B\ ds\ [\ t\ ]\ \texttt{of}\ \{\ p_i\ \texttt{->}\ \mathcal{T}_B\ ds\ [\ t_i\ ]\ \}_{i\in N}$$

B3 (Float out inner `let`)

$$\mathcal{T}_B\ ds\ [\ \mathcal{C}\langle\texttt{let}\ \{\ v = t\ \}\ \texttt{in}\ t_1\rangle\ ] \quad \Rightarrow \texttt{let}\ \{\ v = t\ \}\ \texttt{in}\ \mathcal{T}_B\ ds\ [\ \mathcal{C}\langle t_1\rangle\ ]$$
$$\textit{where } t \textit{ contains no calls to functions in SRPSet}$$

B4 (Gather calls)

$$\mathcal{T}_B\ ds\ [\ \mathcal{C}\langle f_1(xs,\vec{t_1}),\ \ldots,\ f_n(xs,\vec{t_n})\rangle\ ] \Rightarrow$$
$$\textit{if } n > 1 \textit{ then } \texttt{let}\ \{(u_1,\ldots,u_n) = \texttt{ftup}(\texttt{xs},\vec{v})\}\ \texttt{in}\ \mathcal{T}_B\ ds'\ [\ \mathcal{C}\langle\ u_1,\ \ldots,\ u_n\ \rangle\ ]$$
$$\textit{where } ds' = ds \cup \{\ (\texttt{ftup}\ (\texttt{xs},\vec{v}),\ (f_1(xs,\vec{t_1}),\ \ldots,\ f_n(xs,\vec{t_n}))) \mid \vec{v} = \mathit{freevar}[\vec{t_1},\ldots,\ \vec{t_n}]\ \}$$
$$\texttt{ftup}\ (\texttt{xs},\vec{v}) = \mathcal{T}'_B\ ds'\ [(tf_1[\vec{t_1}/\vec{v_1}],\ldots,tf_n[\vec{t_n}/\vec{v_n}])\ ]$$
$$\forall i \in 1\cdots n\ .\ tf_i \textit{ is the RHS of function } f_i.$$

B5 (Tuple `case`)

$$\mathcal{T}'_B\ ds\ [(\texttt{case xs of}\ \{p_i\ \texttt{->}\ t1_i\}_{i\in N},\ \ldots,\ \texttt{case xs of}\ \{p_i\ \texttt{->}\ tn_i\}_{i\in N})] \Rightarrow$$
$$\texttt{case xs of}\ \{\ p_i\ \texttt{->}\ \mathcal{T}_B\ ds\ [\ (t1_i,\ \ldots,\ tn_i)\ ]\ \}_{i\in N}$$

Figure 2: Basic Tupling Algorithm

Calls gathered are recorded in the set *ds*, which acts as a pool of memoised points for terminating tuple transformation.

- B5 - To combine multiple case constructs (testing on the same recursion argument) into a single case construct. This eliminates multiple traversals over the common recursion argument.

Termination of the tupling transformation is facilitated by the well-known folding operation. Thus, instead of defining a new tuple function at B4, previously-defined tuple function is used whenever each tuple of calls gathered is identical (modulo variable renaming and ordering of the calls) to an earlier tuple.

Let us consider the effect of basic tupling algorithm on a typical function with multiple traversals shown in Fig. 3a. Inside the definition of `f`, there are two functions, `g` and `h` that traverse some common recursive structure. Generally speaking, their recursive calls occur in nested function applications and so we have to abstract out these calls by let-expressions. The basic tupling algorithm will transform the above functions to the form in Fig. 3b, where closures are made explicit through the `let` construct.

Closures built in the programs are identified in Fig. 3. Those that exist before *and* after tupling are linked by a solid line. Notice that more closures are built after tupling.

With this view of cost allocation, the extra cost incurred from tupling are: building one extra closure for the `n`-tuple and `n` extra closures for the *tuple components*. We shall refer to the first kind of closure as *tuple-closure*, and the second kind as *component-closure*.

4 Strictness-Guided Tupling

Naive elimination of multiple traversals may cause performance to degrade for lazy programs. This is due to the extra closures created by our transformation in an attempt to adhere to the lazy semantics of the subject language. Fortunately, our programs are often inherently stricter. Exploiting such strictness properties may help avoid some of these extra closures. For example, consider the `avtup(xs)` call in the RHS of `av` in Sec. 1. Both components of `avtup(xs)` are strictly needed, and so are every recursive call to `avtup`. The former requirement can help eliminate closures for the tuple-components, while the latter suggests that the tu-

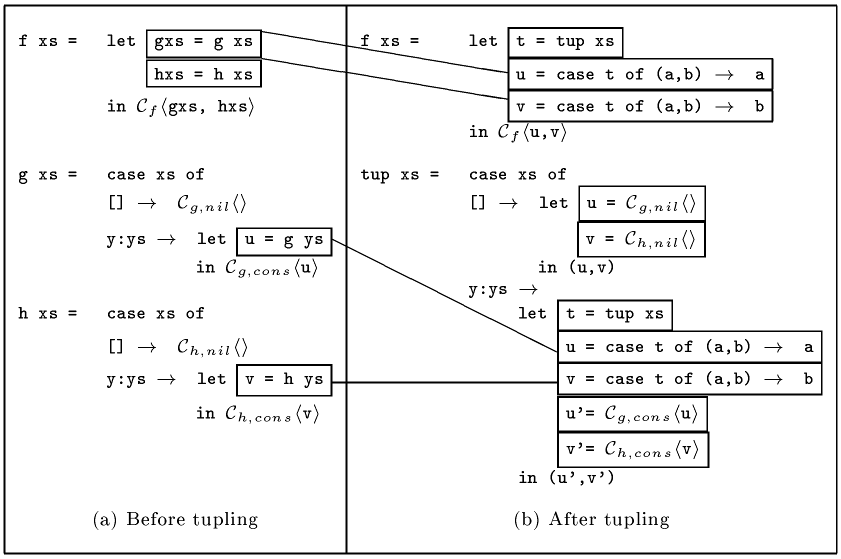

Figure 3: Closure Allocation ($\boxed{\text{A box indicates a closure}}$)

ple itself can be strictly evaluated with its result returned via the stack (or registers).

4.1 A Suitable Strictness Analysis

To implement the suggested enhancement, we require an appropriate strictness analyser. Here, we propose to use a strictness domain based on the 4-point strictness analysis first introduced by Wadler [Wad87] for recursive list-type objects. The four strictness values required are:

- `?` *Don't Know* - Expression may not be evaluated;
- `!` *Simple strictness* - Expression is evaluated to its head normal form;
- `$` *Tail-strictness* - Recursively evaluates all recursive components of the expression;
- $\star$ *Total-strictness* - All recursive components are evaluated to tail-strict form, while the non-recursive component are evaluated to head normal form.

Note the partial ordering of strictness values based on information containment: $\star \sqsubseteq \$ \sqsubseteq ! \sqsubseteq ?$.

Given an expression, e, and a strictness value s, we write $e ::_{\mathcal{S}} s \vdash \theta$ to mean e be used in the strictness context expressed by s to infer strictness environment θ which captures the strictness of all free/bound variables in e.

The 4-point strictness domain was originally introduced for the list-type but it is straightforward to extend it to arbitrary recursive types - with `$` to denote that recursive spines of the expression will be evaluated, and $\star$ implies evaluating the non-recursive components to head normal form, in addition to evaluating the recursive spines. Based on this domain, functions `sum` and `len` of Sec. 1 could be annotated with the following strictness rules:

$$\begin{array}{l}\texttt{sum} ::_{\mathcal{S}} \star \to \texttt{!} \\ \texttt{len} ::_{\mathcal{S}} \texttt{\$} \to \texttt{!}\end{array}$$

These strictness rules state that, when the result of evaluation is required in head normal form, `len` is tail-strict (`$`) on its list-input, while `sum` is totally-strict ($\star$) on its input. Furthermore, for functions which may be invoked at different strictness contexts, we associate a set of strictness rules to it, such as the function `rev` defined below:

```
rev ::S {$ × ? → !, $ × $→ $, ⋆ × ⋆→ ⋆}
rev(xs,ws) = case xs of {[] -> ws ;
                          x:xs' -> rev(xs',x:ws)}
```

For example, the first rule of `rev` is applicable under simple strictness context, while the second and third rules are applicable under tail-strict and totally-strict contexts, respectively.[1]

For accuracy, our chosen strictness domain also incorporates *disjunctive* strictness information, similar to that proposed by Jensen in [Jen97]. Consider:

```
f(xs,y,z) = case xs of {[] -> y ; x:xs' -> z}
```

To capture the strictness of this function `f` more accurately, we require:

[1] By default, no argument will get evaluated when the function result is not needed. That is, $f ::_{\mathcal{S}} ? \to ?$ for all function f.

$$\texttt{f} :: _{\mathcal{S}} \; (\texttt{!},\texttt{!},\texttt{?}) \vee (\texttt{!},\texttt{?},\texttt{!}) \rightarrow \texttt{!}$$

This indicates that the first argument will be evaluated to head normal form, and so will either the second or the third argument, but not both.

Though rarely used in practice, such strictness analyses are not new. Many papers have been written about similarly advanced strictness analyses, often more sophisticated than the version presented here. Rules for analysing their strictness properties are also quite standard, and can be found in [Wad87] and [Jen97].

4.2 Strictness-Guided Tupling

We now look at how the proposed strictness annotations can help guide our tupling algorithm. Fig. 4 gives the main rules for our enhanced tupling algorithm.

We first show the conditions under which creation of extra closures may be avoided during tupling. Next, we look at the propagation of strictness information by our strictness-guided tupling algorithm.

Consider a tupled-function call below with three components which may have been obtained from a call-gathering step (S5).

$$\texttt{let (u1,u2,u3)} = (\texttt{ftup(xs,t)} :: _{\mathcal{S}} s) \texttt{ in } \mathcal{C}\langle\texttt{u1,u2,u3}\rangle$$

Without strictness information, we can only assume that all three tuple variables `u1, u2, u3` may not be needed during execution. Hence, the call `ftup(xs,t)` must be created as a tuple-closure. However, if strictness information s is available we may be able to decide if the `ftup(xs,t)` could be evaluated strictly. In fact, we can evaluate this tuple-call strictly if at least one of `u1,u2,u3` is known to be evaluated to head normal form.

In order words, if $s \sqsubseteq (\texttt{!},\texttt{?},\texttt{?}) \vee (\texttt{?},\texttt{!},\texttt{?}) \vee (\texttt{?},\texttt{?},\texttt{!})$, we can convert the above lazy `let` to a strict `let!`. For convenience, we write $\mathcal{M}_n$ as an abbreviation of the strictness context $\vee_{i=1}^{n}\{(s_1,\ldots,s_n) \mid s_i = \texttt{!} \wedge s_j = \texttt{?}, \forall j \neq i\}$. The latter implies that at least one component of its associated tuple will be needed.

Our rule to *avoid constructing tuple-closure* can now be expressed as follows, which is an intuitive interpretation of rule S5 in Fig. 4.

if $s \sqsubseteq \mathcal{M}_n \wedge \texttt{ftup(xs,t)} :: _{\mathcal{S}} s$ then
$\texttt{let } \{\texttt{(u1,..,un) = ftup(xs,t)}\} \texttt{ in } \mathcal{C}\langle\texttt{u1,u2,u3}\rangle \;\Rightarrow$
$\quad \texttt{let! } \{\texttt{(u1,..,un) = ftup(xs,t)}\} \texttt{ in } \mathcal{C}\langle\texttt{u1,u2,u3}\rangle.$

Furthermore, when a tuple component is expected to be strictly evaluated, we can avoid building closure for that component. Consider the tuple $(e_1 :: _{\mathcal{S}} s_1, \ldots, e_n :: _{\mathcal{S}} s_n)$. If it is determined that $s_i \sqsubseteq \texttt{!}$, then we can evaluate e_i strictly, provided it is neither a variable nor a constant. (Variables/constants do not result in new closures and hence need not be strictly evaluated). A general rule to *avoid building component-closure* is as follows, which corresponds to rule S6 in Fig. 4:

if $s_i \sqsubseteq \texttt{!} \wedge e_i :: _{\mathcal{S}} s_i$ then
$\quad (e_1, \ldots, e_i, \ldots, e_n) \Rightarrow$
$\quad\quad \texttt{let! } \{v_i = e_i\} \texttt{ in } (e_1, \ldots, v_i, \ldots, e_n).$

These two strictness optimisations are keys to effective tupling for lazy languages. To enable these two optimisations, our tupling algorithm must aggressively propagate strictness information during tupling. Where possible, it should also use the best strictness context available for a given tuple of abstracted calls; and aggressively propagate this strictness context into subexpressions, where feasible. This is expressed in the other rules of the algorithm, where attempts are made to propagate the strictness context to the subexpressions during tupling. Rule S1 is similar to rule B1 of the basic tupling algorithm. Rules S2 reduces `case` expression when possible. Rule S3 allows each branch of the lifted `case` to be exposed to the propagated strictness context. Lastly, rules S4 and S7 are similar to rule B3 and B5 in the basic algorithm.

5 An Example

In this section, we provide a detailed example of tupling application. Consider the function `sumseg` :

```
sumseg(xs,n) =
  case xs of
  []     -> 0
  (y:ys) -> case (n==0) of
            True  -> 0
            False -> let { n' = n - 1 }
                     in  y + sumseg(ys,n')
```

If we apply tupling to `sumseg(xs,n)+sumseg(xs,m)` , we will gather these two calls under strictness context $(\texttt{!}\times\texttt{!})$. Transformation guided by this strictness context will be as follows (several arguments of the transformation are omitted for clarity):

$\mathcal{T}\ (\texttt{!}\times\texttt{!})\ [\ \texttt{sumseg(xs,n)+sumseg(xs,m)}\ ]$
$\quad (S5)\Rightarrow \texttt{let! (u,v) = ftup(xs,n,m)} :: _{\mathcal{S}} (\texttt{!}\times\texttt{!}) \texttt{ in u+v}$
$\quad$ Define $\texttt{ftup(xs,n,m)} :: _{\mathcal{S}} (\texttt{!}\times\texttt{!}) =$
$\mathcal{T}'\ (\texttt{!}\times\texttt{!})\ [\ (\texttt{case xs of } \{\texttt{[] -> 0; (y:ys) -> } \mathcal{C}\langle n\rangle\},$
$\quad\quad \texttt{case xs of } \{\texttt{[] -> 0; (y:ys) -> } \mathcal{C}\langle m\rangle\})\ ]$
$\quad$ *where* $\mathcal{C}\langle a\rangle = \texttt{case } (a\texttt{== 0}) \texttt{ of}$
$\quad\quad\quad \texttt{True -> 0 ;}$
$\quad\quad\quad \texttt{False -> let \{ n' = } a\texttt{-1\}}$
$\quad\quad\quad\quad \texttt{in y + sumseg(ys,n')}$
$\quad (S7)\Rightarrow \texttt{case xs of}$
$\quad\quad \texttt{[] -> (0,0)}$
$\quad\quad \texttt{y:ys -> } \mathcal{T}\ (\texttt{!}\times\texttt{!})[(\mathcal{C}\langle n\rangle,\ \mathcal{C}\langle m\rangle)]$

Though the case-test has been lifted from the components, there are still two inner case constructs, denoted by $\mathcal{C}\langle n\rangle$ and $\mathcal{C}\langle m\rangle$, which actually make the recursive calls to `sumseg` occur in lazy context (wrt the RHS of `sumseg`). However, as the strictness information $(\texttt{!}\times\texttt{!})$ is propagated to the subexpressions during transformation, both $\mathcal{C}\langle n\rangle$ and $\mathcal{C}\langle m\rangle$ lie in strict context, and this enable us to transform $(\mathcal{C}\langle n\rangle,\mathcal{C}\langle m\rangle)$ further. We apply S3 twice to lift two `case` expressions, and then apply S4 several times on each of the branches of the `case` expressions:

S1 (Skip outer `let`)

$\mathcal{T}\ s_0\ ds\ [\texttt{let}\ \{(v_1,\ldots,v_n) = (t_1,\ldots,t_n)\}\ \texttt{in}\ t] \ \Rightarrow\ \texttt{let}\ \{(v_1,\ldots,v_n) = (t'_1,\ldots,t'_n)\}\ \texttt{in}\ \mathcal{T}\ s\ ds\ [t]$

where $t ::_{\mathcal{S}} s_0 \vdash \theta$

$t'_i = \mathcal{T}\ (\theta[v_i])\ ds\ [t_i]\quad \forall i \in M$

S2 (Reduce `case`)

$\mathcal{T}\ s_0\ ds\ [\ \mathcal{C}\langle\ \texttt{case}\ c_j(\vec{t})\ \texttt{of}\ \{\ c_i(\vec{v})\ \texttt{->}\ t_i\ \}_{i\in M}\ \rangle\ ] \ \Rightarrow\ \mathcal{T}\ s_0\ ds\ [\ \mathcal{C}\langle\ t_j[\vec{t}/\vec{v}]\ \rangle\ ]$

S3 (Lift `case`)

$\mathcal{T}\ s_0\ ds\ [\ \mathcal{C}\langle\texttt{case}\ t\ \texttt{of}\ \{\ p_i\ \texttt{->}\ t_i\ \}_{i\in M}\rangle]\qquad \Rightarrow$

if $(\theta[u]) \sqsubseteq\ !$ *then* $\texttt{case}\ (\mathcal{T}\ (\theta[u])\ ds\ [t])\ \texttt{of}\ \{\ p_i\ \texttt{->}\ \mathcal{T}\ s_0\ ds\ [\ \mathcal{C}\langle t'_i[p_i/t]\rangle\ ]\ \}_{i\in M}$

where $\mathcal{C}\langle\texttt{case}\ u\ \texttt{of}\ \{\ p_i\ \texttt{->}\ t_i\ \}_{i\in M}\rangle ::_{\mathcal{S}} s_0 \vdash \theta$ *for new variable* u

S4 (Float out inner `let`)

$\mathcal{T}\ s_0\ ds\ [\ \mathcal{C}\langle\texttt{let}\ \{\ v = t\ \}\ \texttt{in}\ t_1\rangle\ ]\quad \Rightarrow \texttt{let}\ \{\ v = t\ \}\ \texttt{in}\ \mathcal{T}\ s_0\ ds\ [\ \mathcal{C}\langle t_1\rangle\ ]$

where t contains no calls to functions in SRPSet

S5 (Gather calls)

$\mathcal{T}\ s_0\ ds\ [\ \mathcal{C}\langle f_1(xs,\vec{t_1}),\ \ldots,\ f_n(xs,\vec{t_n})\rangle]\quad \Rightarrow$

if $s \sqsubseteq \mathcal{M}_n \wedge n > 1$ *then* $\texttt{let!}\ \{(u_1,\ldots,u_n) = ftup(xs,\vec{v})\}\ \texttt{in}\ \mathcal{T}\ s_0\ ds'\ [\mathcal{C}\langle u_1,\ldots,u_n\rangle\ ]$

where $\mathcal{C}\langle u_1,\ldots,u_n\rangle ::_{\mathcal{S}} s_0 \vdash \theta$ *for new variables* $u_1,\ldots,u_n$

$s = \theta[(u_1,\ldots,u_n)]$

$ds' = ds \cup \{\ (ftup(xs,\vec{v}),\ s,(f_1(xs,\vec{t_1}),\ \ldots,\ f_n(xs,\vec{t_n})))\ |\ \vec{v} = freevar[\vec{t_1},\ldots,\ \vec{t_n}]\ \}$

$ftup(xs,\vec{v}) = \mathcal{T}'\ (s \sqcap \mathcal{M}_n)\ ds'\ [\ (tf_1[\vec{t_1}/\vec{v_1}],\ldots,tf_n[\vec{t_n}/\vec{v_n}])\ ]$

$\forall i \in 1\cdots n\ .\ tf_i$ *is the RHS of function* f_i.

S6 (Make tuple components strict)

$\mathcal{T}\ s_0\ ds\ [\ (t_1\ ::_{\mathcal{S}}\ s_1,\ \ldots,\ t_n\ ::_{\mathcal{S}}\ s_n)::_{\mathcal{S}}\ s_0]\ \Rightarrow\ \texttt{let!}\ \{\ v_i = t_i\ \}_{i\in N}\ \texttt{in}\ \mathcal{T}\ s_0\ ds\ (t'_1,\ldots,t'_n)$

where $t'_i =$ *if* $i \in N$ *then* v_i *else* t_i

$N \equiv \{j\ |\ j \in 1\cdots n, s_j \sqsubseteq !, not(isVariable?(t_j)\}, not(isConstant?(t_j)\}.$

S7 (Tuple `case`)

$\mathcal{T}'\ s_0\ ds\ [(\texttt{case xs of}\ \{p_i\ \texttt{->}\ t1_i\}_{i\in M},\ \ldots,\ \texttt{case xs of}\ \{p_i\ \texttt{->}\ tn_i\}_{i\in M})\ ]\ \Rightarrow$

$\texttt{case xs of}\ \{p_i\ \texttt{->}\ \mathcal{T}\ s_0\ ds\ [\ (t1_i,\ldots,tn_i)]\ \}_{i\in M}$

Figure 4: Strictness-Guided Tupling Algorithm

```
T(!×!) [ (C⟨n⟩,C⟨m⟩) ]    (S3,S4)⇒
   case (n==0) of
   True -> case (m==0) of
           True -> T(!×!) [ (0,0) ]
           False -> let {m' = m - 1}
                    in  T(!×!)[(0,y+sumseg(ys,m'))]
   False -> case (m==0) of
            True -> let {n' = n - 1}
                    in  T(!×!)[(y+sumseg(ys,n'),0)]
            False -> let {n' = n - 1
                          m' = m - 1}
                     in T(!×!)[(y+sumseg(ys,n'),
                                y+sumseg(ys,m'))].
```

Consider the four branches in the code. The first branch does not have any recursive call, while the second and third branches have only one recursive call each. Hence, call-gathering need not be invoked. The last branch contains two recursive calls which now lie in strict context for our tupling transformer to gather. As this specialised context is identical to the previous tuple definition of `ftup`, the algorithm performs a fold operation to end the recursive transformation. Finally, as both the tuple-call and their components lie in their respective strict contexts, we can apply rules S5 and S6 to yield the following:

```
T(!×!)[(y+sumseg(ys,n'),y+sumseg(ys,m'))]
(S5,S6)⇒ let! {(u,v) = ftup(ys,n',m')::S (!×!)
               u' = y + u
               v' = y + v }
         in    (u', v')
```

The final transformed code is as follows:

```
ftup(xs,n,m)::S (!×!) =
   case xs of
   [] -> (0,0)
   (y:ys') ->
     case (n==0) of
     True -> case (m==0) of
             True -> (0,0)
             False -> let m' = m - 1 in
                      let! v = y+sumseg(ys,m')
                      in   (0, v)
     False -> case (m==0) of
              True -> let n' = n - 1 in
                      let! u = y+sumseg(ys,n')
                      in   (u, 0)
              False ->
                let {n' = n - 1
                     m' = m - 1}
                in   let! {(u,v)=ftup(ys,n',m')::S (!×!)
                           u' = y + u
                           v' = y + v}
                     in    (u', v')
```

Tab. 2 shows the run-time improvement of the transformed program.

At this point, we would like to justify our decision to allow strictness annotations to guide tupling.

One may wonder if it might be simpler to just apply strictness optimisation after basic tupling? The following is the result of transforming the same expression with basic tupling tactic:

```
ftup1(xs,n,m) =
   case xs of
   [] -> (0,0)
   (y:ys) ->
```

1000 times of sumseg with xs = [0..10000]					
	Heap (bytes)	Time(s)			
		INIT	MUT	GC	Total
sumseg(xs,900)+sumseg(xs,900)					
No Tupling	72,071,280	0.04	54.66	0.13	54.83
Basic Tupling	140,608,348	0.01	74.09	1.47	75.57
New Tupling	43,283,450	0.01	41.80	0.04	41.85
sumseg(xs,250)+sumseg(xs,750)					
No Tupling	40,068,280	0.03	30.29	0.03	30.35
Basic Tupling	117,204,988	0.02	50.23	0.65	50.90
New Tupling	32,088,520	0.01	27.04	0.05	27.10

Table 2: Execution times of `sumseg`

```
let {n' = n - 1
     m' = m - 1
     (u,v) = ftup1(ys,n',m') } in
case (n==0) of
True -> case (m==0) of
        True -> (0,0)
        False -> (0, y + v)
False -> case (m==0) of
         True -> (y + u, 0)
         False -> (y + u, y + v)
```

The above code is less efficient than the code produced by strictness-guided tupling: it will invoke calls to `ftup1` throughout, resulting in the creation of unnecessary tuple-closures as well as component-closures. Because of the lack of strictness information, the basic tupling algorithm was not able to delve into various branches of the `case` expressions, and can only gather similar calls at a global level. At that point, strictness analysis alone is unable to recover our earlier level of optimisation. Several other steps are also needed, including the inverse of tupling. A more sophisticated strictness analysis over the tupled program would have to be applied, followed by float-in of tupled-calls into `case` branches to exploit better strictness. This may not be sufficient, as some of the tupled calls (e.g. in second and third branches of `sumseg`) might have to be "untupled" to achieve better performance. Our proposal to use strictness-guided tupling is therefore simpler, as it introduces tupled-functions only when tuple-closures can be eliminated. This provides some guarantee on the performance of each such tupled-functions.

One may also wonder if it is better to apply strictness optimisation prior to basic tupling? This approach may be helpful to the extent that it could help compile away the `$`- and $\star$-strictness annotations into our code. However, strictness propagation for each tuple of calls gathered is still required during tupling. Without it, some opportunities for eliminating closures for tuple-results and their components would be lost. For example, when gathering two calls c_1 and c_2 under strictness contexts s_1, s_2, we should propagate the strictness contexts s_1, s_2 during tupling. Failure to do so may result in less closures being eliminated.

6 Circular Tupling

A particularly elegant use of tupling was proposed by Richard Bird [Bir84] where he showed how circular tupled programs could be used to eliminate multiple traversals of nested function calls. A classic example is the palindrome function:

```
pal(xs)    = eq(xs,rev(xs,[]))
eq(xs,ys)  = case xs of { [] -> (ys==[]);
   x:xs' -> case ys of { [] -> False;
            y:ys' -> (x==y) and eq(xs',ys')} }
rev(xs,ws) = case xs of { [] -> ws;
                          x:xs' -> rev(xs',x:ws) }
```

Here, we have two recursive calls, namely `eq(xs,_)` and `rev(xs,[])`. However, `rev` call is nested within `eq` call. These two calls separately traverse the same data structure `xs`. Bird showed how such nested expressions can be manually transformed to tupled circular functions. In this section, we shall examine how circular tupling could be systematically handled by our enhanced tupling algorithm. A key step is to introduce a recursive `letrec` construct with a circular variable to unnest the inner call. For the palindrome example, this step results in:

```
pal(xs) = letrec (u0,u1)=re_tup(xs,[],u0 ::F?) ::S(!×!)
          in u1
Define:
re_tup(xs,ws,u0)
        = (rev(xs,ws),eq(xs,u0 ::F?)) ::S(!×!)
```

The two recursive calls were found in strict contexts - motivating our use of strictness annotation `(!×!)` for the gathered calls. Also, as the variable `u0` is circular, it must not be evaluated strictly; otherwise our program will chase after a result that has not been created yet. Hence, during the introduction of `letrec`, circular variables must be placed in lazy context. To achieve this, we introduce a new annotation $e ::_{\mathcal{F}} s$ whose purpose is to *f*orce subterm e to a stated strictness s.

To incorporate this step into our tupling method, we require a special rule, *S9*, shown in Fig. 5. Note that a nested inner call $f_0(xs,\vec{t_0})$ is abstracted via a circular variable. Applying our enhanced tupling algorithm to the above example yields:

```
re_tup(xs,ws,u0) ::S(!×!)
  = case xs of { [] -> (ws,u0 ::F?==[]);
    x:xs' -> let!  (u,v)=re_tup(xs',x:ws,tl(u0)) ::S(!×?)
    in (u, case u0 ::F? of { [] -> False;
                             y:ys' -> (x==y) and v} }
re_tup(xs,ws,u0) ::S(!×?)
  = case xs of { [] -> (ws,u0 ::F?==[]);
    x:xs' -> let!  (u,v)=re_tup(xs',x:ws,tl(u0)) ::S(!×?)
    in (u, case u0 ::F? of { [] -> False;
                             y:ys' -> (x==y) and v} }
tl(xs) = case xs of {x:xs' -> xs'}
hd(xs) = case xs of {x:xs' -> x}
```

S8 (Lift `case` - Speculative)

$$\mathcal{T}\ s_0\ ds\ [\ \mathcal{C}\langle \texttt{case}\ t\ \texttt{of}\ \{\ p_i\ \texttt{->}\ t_i\ \}_{i\in M}\rangle]\quad\Rightarrow$$
$$\textit{if}\ \ s \sqsubseteq \texttt{!}\ \textit{then}\ \ \texttt{case}\ (\mathcal{T}\ s\ ds\ [t])\ \texttt{of}\ \{\ p_i\ \texttt{->}\ \mathcal{T}\ s_0\ ds\ [\ \mathcal{C}\langle t'_i[p_i/t]\rangle\ ]\ \}_{i\in M}$$
$$\textit{where}\ \theta_0 \vdash t ::_{\mathcal{O}} s\ \textit{(NB: } \theta_0 \textit{ is the strictness environment of its current context.)}$$

S9 (Circular Tupling)

$$\mathcal{T}\ s_0\ ds\ [\ \mathcal{C}\langle f_1(\mathrm{xs},f_0(\mathrm{xs},\vec{t_0}),\vec{t_1}),\ \ldots,\ f_n(\mathrm{xs},\vec{t_n})\rangle]\quad\Rightarrow$$
$$\texttt{letrec}\ \{(u_0,\ldots,u_n) = \textit{ftup}(\mathrm{xs},u_0 ::_{\mathcal{F}} \texttt{?},\vec{v})\}\ \texttt{in}\ \mathcal{T}\ s_0\ ds'\ [\mathcal{C}\langle u_1,\ldots,u_n\rangle\ ]$$
$$\textit{where}\ (\texttt{let}\ \ u_1{=}f_1(\mathrm{xs},u_0,\vec{t_1})\ \texttt{in}\ \mathcal{C}\langle u_1,\ldots,u_n\rangle) ::_{\mathcal{S}} s_0 \vdash \theta \qquad \textit{for new variables } u_0,\ldots,u_n$$
$$s = \theta[(u_0,\ldots,u_n)]$$
$$ds' = ds \cup \{\ (\textit{ftup}(\mathrm{xs},u_0,\vec{v}),\ s,(f_0(\mathrm{xs},\vec{t_0}),\ f_1(\mathrm{xs},u_0,\vec{t_1}),\ \ldots,\ f_n(\mathrm{xs},\vec{t_n})))\mid \vec{v} = \textit{freevar}[\vec{t_0},\ldots,\ \vec{t_n}]\ \}$$
$$\textit{ftup}(\mathrm{xs},\vec{v}) = \mathcal{T}'\ \ (s\sqcap\mathcal{M}_n)\ ds'\ [\ (\textit{tf}_0[\vec{t_0}/\vec{v_0}],\textit{tf}_1[u_0/v_0,\vec{t_1}/\vec{v_1}],\ldots,\textit{tf}_n[\vec{t_n}/\vec{v_n}])\ ]$$
$$\forall i \in 0\cdots n.\ \textit{tf}_i \textit{ is the RHS of function } f_i.$$

S10 (Make tuple components strict - Speculative)

$$\mathcal{T}\ s_0\ ds\ [\ (t_1\ ::_{\mathcal{O}} s_1,\ \ldots,\ t_n\ ::_{\mathcal{O}} s_n)]\quad\Rightarrow\quad \texttt{let!}\ \{\ v_i = t_i\ \}_{i\in N}\ \texttt{in}\ \mathcal{T}\ s_0\ ds\ [\ (t'_1,\ldots,t'_n)\ ]$$
$$\textit{where}\ t'_i = \textit{if}\ i \in N\ \textit{then}\ v_i\ \textit{else}\ t_i$$
$$N \equiv \{j \mid j \in 1\cdots n, s_j \sqsubseteq \texttt{!}, \textit{not}(\textit{isVariable?}(t_j)), \textit{not}(\textit{isConstant?}(t_j)\}.$$

Figure 5: Extra Rules for Enhanced Tupling Algorithm

Two `re_tup` definitions were introduced under strictness contexts (`!`×`!`) and (`!`×`?`), respectively. However, as both definitions are syntactically identical we could combine them into a single definition to reduce code duplication. Also, as our tupling algorithm is strictness-guided, it is able to detect that tuple-closures were unnecessary for the recursive `re_tup` calls. In addition, the first component of `re_tup` calls can be strictly evaluated, but not the second component. The lazy annotation on `u0` forces closures to be built for the second component, despite the fact that the original `eq` call was lying in a strict context. This is in contrast to [Bir84] which requires a manual intervention (before tupling) to redefine `eq` to make its second parameter 'lazy'.

Another interesting example is the program to replace all tips of a given tree by its minimum value.

```
data Tree(a) = Leaf(a)  | Node(Tree(a),Tree(a))
mintip(t) = repl(t,mint(t))
repl(t,m) = case t of { Leaf(a) -> Leaf(m);
            Node(l,r) -> Node(repl(l,m),repl(r,m)) }
mint(t)   = case t of { Leaf(a) -> a;
            Node(l,r) -> min2(mint(l),mint(r)) }
min2(x,y) = case (x < y) of { True -> x; False -> y }
```

As `mintip` returns a tree structure, it may be evaluated under different strictness contexts. If we use a head-strict `!` context for `mintip` and apply basic tupling algorithm, we obtain:

```
mintip(t) ::S != letrec (u,v)=rm_tup(t,v) ::S (!×?) in u
rm_tup(t,m) ::S (!×?)
   = case t of { Leaf(a) -> (Leaf(m),a);
     Node(l,r) -> let (u,v)=rm_tup(l,m) ::S (?×?) in
                  let (y,z)=rm_tup(r,m) ::S (?×?) in
                  (Node(u,y),min2(v,z)) } }
rm_tup(t,m) ::S (?×?)
   = case t of { Leaf(a) -> (Leaf(m),a);
     Node(l,r) -> let (u,v)=rm_tup(l,m) ::S (?×?) in
                  let (y,z)=rm_tup(r,m) ::S (?×?) in
                  (Node(u,y),min2(v,z)) }
```

Such functions are less efficient than their untupled equivalent. Fortunately, our enhanced tupling algorithm avoids such functions, since Step *S5* only introduces each tupled function when its corresponding tuple-closure can be eliminated.

If `mintip` is used under a tail-strict `$` context, we can obtain a more efficient tupled program:

```
mintip(t) ::S $= letrec (u,v)=rm_tup(t,v) ::S ($×?) in u
rm_tup(t,m) ::S ($×?)
   = case t of { Leaf(a) -> let!  r=Leaf(m) in (r,a);
     Node(l,r) -> let!  (u,v)=rm_tup(l,m) ::S ($×?) in
                  let!  (y,z)=rm_tup(r,m) ::S ($×?) in
                  let!  r=Node(u,y) in (r,min2(v,z)) }
```

No tuple-closures will be built on the heap. However, the second component, involving `min2` calls, will still be built as thunks.

An even better result can be obtained if `mintip` is transformed under a totally-strict $\star$ context. Under this scenario, our tupling algorithm obtains:

```
mintip(t) ::S ⋆ = letrec (u,v)=rm_tup(t,v) ::S (⋆×!) in u
rm_tup(t,m) ::S (⋆×!)
   = case t of { Leaf(a) -> let!  r=Leaf(m) in (r,a);
     Node(l,r) -> let!  (u,v)=rm_tup(l,m) ::S (⋆×!) in
                  let!  (y,z)=rm_tup(r,m) ::S (⋆×!) in
                  let!  r=Node(u,y) in
                  let!  n=min2(v,z) in (r,n) }
```

Performance for all three tupled programs, under their respective strictness contexts, are shown in Table 3. Naive-tupling under `!`-context ends up being worse than no tupling. However, both `$`-strict and $\star$-strict tuplings are better by about 16% and 18%, respectively, when compared to the corresponding untupled programs with the same level of strictness. This gain is due solely to the elimination of multiple traversals. The tupled function of *mintip* under $\star$-strictness is itself about 18% better than the corresponding tupled function under `$`-strictness. This gain is due to the elimination of closures for the second component of `rm_tup` calls.

100 times of `mintip` on a tree of depth 12					
	Heap (bytes)	Time(s)			
		INIT	MUT	GC	Total
No Tupling (!)	29,679,824	0.02	19.49	0.57	20.08
Tupling with (!)	52,620,228	0.01	27.81	4.14	31.96
No Tupling ($)	28,039,728	0.03	17.28	0.85	18.16
Tupling with ($)	23,126,952	0.02	14.51	0.80	15.33
No Tupling ($\star$)	24,762,928	0.03	14.51	0.57	15.11
Tupling with ($\star$)	18,211,728	0.02	11.95	0.34	12.31

Table 3: Execution times of `mintip`

7 Speculative Strictness

The optimisation achieved by our tupling algorithm is to a large extent determined by the level of strictness detected. Better strictness can often be found if more specialised strictness contexts are considered. However, a drawback is that more specialised contexts often meant more code duplication. For example, if we decide to keep all three strictness contexts for `mintip`, we will need to keep three variants of its tupled functions.

An alternative strategy is to make use of *speculative strictness*. Conventional strictness analysis is used to detect expressions which are definitely needed, and could therefore be evaluated safely in advance. Speculative strictness, on the other hand, is used to detect expressions which may not be needed but are nevertheless safe to evaluate because their (advance) evaluations do not contribute to new sources of non-termination.

Definition 1: *Speculative Strictness*
Given an expression e, under non-strict context `?`, and strictness environment θ, we could safely evaluate e under speculative strictness s if its evaluation does not result in non-termination, i.e. $\theta \vdash eval_s\, e \neq \perp$. Note that $eval_s\, e$ denotes the evaluation of e to the strictness extent s.

To support speculative strictness, we propose a new set of analysis rules in Fig. 6 which could determine the extent an expression e can be safely over-evaluated, under strictness environment θ. This forward analysis is written as $\theta \vdash e ::_{\mathcal{O}} s$, where s is the speculative strictness of e under θ. Note that data construction is written in the figure as $c(\mathtt{e}_1,\ldots,\mathtt{e}_m \mid \mathtt{e}_{m+1},\ldots,\mathtt{e}_n)$, where $\mathtt{e}_1,\ldots,\mathtt{e}_m$ are the non-recursive components of the constructor, and $\mathtt{e}_{m+1},\ldots,\mathtt{e}_n$ are the recursive components. For example, the list constructor can be written as `cons(x|xs)` .

Some simple examples of speculative strictness are given below:

$$
\begin{array}{lcll}
\{\mathtt{x} ::_{\mathcal{S}} !,\mathtt{p} ::_{\mathcal{S}} !\} & \vdash & (\mathtt{x>p}) & ::_{\mathcal{O}} !\\
\{\mathtt{x} ::_{\mathcal{S}} !\} & \vdash & (\mathtt{x<p}) & ::_{\mathcal{O}} ?\\
\{\mathtt{x} ::_{\mathcal{S}} !,\mathtt{p} ::_{\mathcal{S}} !\} & \vdash & (\mathtt{x/p}) & ::_{\mathcal{O}} !\\
\{\mathtt{x} ::_{\mathcal{S}} !,\mathtt{p} ::_{\mathcal{S}} !\} & \vdash & (\mathtt{x+p}) & ::_{\mathcal{O}} !
\end{array}
$$

The strictness environment $\{\mathtt{x} ::_{\mathcal{S}} !,\mathtt{p} ::_{\mathcal{S}} !\}$ indicates that both `x` and `p` are assumed to have been evaluated. Under this assumption, the expression `(x>p)` can be speculatively evaluated with strictness `!`, even though it may not be required. This is possible since such an evaluation will not cause non-termination, under the assumption that `x` and `p` are already evaluated. In the second example, `(x<p)` cannot be speculatively evaluated since its argument `p` (which may be $\perp$) has not been evaluated yet.

Some operators such as `/` and `+` may cause runtime errors, e.g. divide-by-zero or overflow exceptions, even when their inputs have already been evaluated. Such exceptions may alter their programs' semantics, making them unsuitable for speculation. Fortunately, modern architectures provide a solution to this problem [MCH+92] by suppressing these exceptions initially, and re-asserting them later when it is determined that they occur in the original program. Our formulation of speculative strictness aggressively assumes this capability.

To illustrate the usefulness of speculative strictness, consider the quicksort function.

```
qsort(xs)    = case xs of { [] -> [];
   x:xs' -> qsort(lower(xs',x))++[x]
            ++qsort(higher(xs',x)) }
lower(xs,p)  = case xs of { [] -> [];
   x:xs' -> case (x < p) of {True -> x:lower(xs',p)
                             False -> lower(xs',p)} }
higher(xs,p) = case xs of { [] -> [];
   x:xs' -> case (x ≥ p) of {True -> x:higher(xs',p)
                             False -> higher(xs',p)} }
```

The `qsort` function has strictness signature
`qsort` $::_{\mathcal{S}}$ `{$ -> !, $ -> $, ⋆ -> ⋆}`

If we apply our tupling algorithm under strictness context `!` (but without speculative strictness), we could gather its two calls `(lower(xs,x),higher(xs,x))` under strictness `($×?)`, followed by transformation to:

```
qsort(xs) ::S !
   = case xs of { [] -> [];
     x:xs' -> let!  (u,v)=lh_tup(xs',x) ::S ($×?)
              in qsort(u)++[x]++qsort(v) }
lh_tup(xs,p) ::S ($×?)
   = case xs of { [] -> ([],[]);
   x:xs' -> let!  (u,v)=lh_tup(xs',p) ::S ($×?) in
    case (x < p) of {
    True -> let!  a=x:u in
            (a,case (x ≥ p) of { True -> x:v;
                                 False -> v});
    False -> (u,case (x ≥ p) of { True -> x:v;
                                  False -> v})}
```

This tupled program is already quite efficient[2], having avoided the need for tuple-closures, and also closures for the first components. However, closures for the second components, involving `higher` calls, are still created. To avoid these closures, there is no need to resort to a

[2]It is possible to improve the tupled version of `qsort` further by exploiting the fact that `(x≥p)=not(x<p)`. This would then allow two common tests to be combined, with two dead branches of inner `case` eliminated. Though we do not show it here, it is a side-benefit make possible by tupling tactic.

$$\frac{}{\theta \vdash \mathtt{v} ::_{\mathcal{O}} \theta[\mathtt{v}]} \qquad \frac{\theta \vdash \mathtt{e} ::_{\mathcal{O}} s}{\theta \vdash \mathtt{e} ::_{\mathcal{O}} s'} \ (s \sqsubseteq s')$$

$$\frac{\mathtt{f} ::_{\mathcal{O}} s_1 \rightarrow s_2 \ \wedge \ \theta \vdash a ::_{\mathcal{O}} s_1}{\theta \vdash (\mathtt{f}\ a) ::_{\mathcal{O}} s_2} \qquad \frac{\forall i \in 1..n,\ \theta \vdash t_i ::_{\mathcal{O}} s_i}{\theta \vdash (t_1, \ldots, t_n) ::_{\mathcal{O}} (s_1 \times \cdots \times s_n)}$$

$$\frac{\forall i \in M.\, \theta \vdash \mathtt{e}_i ::_{\mathcal{O}} s_i \ \wedge \ \{(\mathtt{v}_i, s_i)\}_{i \in M} \cup \theta \vdash \mathtt{e} ::_{\mathcal{O}} s}{\theta \vdash (\mathtt{let}\ \{\mathtt{v}_i = \mathtt{e}_i\}_{i \in M}\ \mathtt{in}\ \mathtt{e}) ::_{\mathcal{O}} s} \qquad \frac{\theta \vdash \mathtt{e}_0 ::_{\mathcal{O}} s \ \wedge \ \forall i \in N.\, \theta \cup \{(p_i, s)\} \vdash \mathtt{e}_i ::_{\mathcal{O}} s_i}{\theta \vdash (\mathtt{case}\ e_0\ \mathtt{of}\ \{p_i \rightarrow \mathtt{e}_i\}_{i \in N}) ::_{\mathcal{O}} \bigvee_{i \in N} s_i} \ (s \sqsubseteq !)$$

$$\frac{(\forall i \in 1..m.\, \theta \vdash \mathtt{e}_i ::_{\mathcal{O}} \{?, ?, !\}) \ \wedge \ (\forall j \in \mathrm{m}{+}1..n.\, \theta \vdash \mathtt{e}_j ::_{\mathcal{O}} \{?, \$, \star\})}{\theta \vdash c(\mathtt{e}_1, \ldots, \mathtt{e}_m \mid \mathtt{e}_{m+1}, \ldots, \mathtt{e}_n) ::_{\mathcal{O}} \{!, \$, \star\}}$$

Figure 6: Rules for Speculative Strictness

$-strict context for `qsort`. Instead, we could use speculative strictness on the second component, under its strictness environment θ={x $::_{\mathcal{S}}$!,xs $::_{\mathcal{S}}$ $,p $::_{\mathcal{S}}$!}. The two inner `case` constructs has a common test, (x $\geq$ p) $::_{\mathcal{O}}$!, that can be speculatively strict, and so are the cons-cell of (x:v) $::_{\mathcal{O}}$!. Applying rules *S8* (to lift `case` speculatively) and *S10* (to make tuple-component speculatively strict), we obtain the following optimised code where closures for the second components have now been eliminated.

```
lh_tup(xs,p) ::S ($×?)
   = case xs of { [] -> ([],[]);
      x:xs' -> let!  (u,v)=lh_tup(xs',p) ::S ($×?) in
      case (x < p) of {
       True -> let!  {a=x:u; b=x:v} in
               case (x ≥ p) of { True -> (a,b);
                                 False -> (a,v)});
       False -> let!  b=x:v in
               case (x ≥ p) of {True -> (u,b);
                                 False -> (u,v)}}
```

Speculative strictness can help reduce the amount of thunks created. However, there is still a trade-off involved, namely that if a speculatively strict expression e is not used, its over-evaluation becomes an overhead.

To highlight the usefulness of speculation, we measured the average time taken for `take(sort(xs),r)` with `r` ranging from `1` to |`xs`|. The performance of three versions of `qsort` are shown in Table 4. The tupled version of `qsort` under !-strict context has about the same speed as the untupled version. The gain from multiple traversals has been eroded by the need to build tuples for the second components of tuple-call. With the aid of speculative strictness, our tupled program now runs about 16% faster, despite the fact that some of the speculative evaluations may be redundant. This gain comes from the elimination of closures via safe over-evaluation.

1000 times of `qsort` on 1000 integers The n-th time taking the first n integers					
	Heap (bytes)	Time(s)			
		INIT	MUT	GC	Total
No Tupling	146,336,536	0.02	146.44	0.44	146.90
Tupling with (!)	186,535,224	0.03	148.89	0.63	149.55
+ Speculation	134,184,556	0.00	125.68	0.23	125.91

Table 4: Execution times of `qsort`

8 Discussion

It is relatively easy to show that our tupling transformation terminates. First, we observe that the application of $\mathcal{T}$ either advances towards the subexpressions, or reduces the complexity of the expression at hand. Second, it is likely that there be an increase in the number of possible tuple functions created during transformation, compared to the conventional (basic) tupling tactic. Such increment is due to the specialisation of calls with respect to strictness information, in addition to the usual symbolic arguments. Since there are only finitely many distinct strictness information associated with a function, the increment in the number of tuple functions created will be bounded. Thus, the termination proof of our new tupling tactic can be reduced to that of the conventional tupling [Chi93].

Due to space constraint, we do not provide any correctness proof to our tupling tactics. Nevertheless, we note that such correctness proof depends on the correctness of each transformation rules as well as the soundness of the strictness analysis. The former can be mirrored from the relevant work in program transformation, such as the work by Runciman for ensuring adherence to lazy semantics [RFJ89] and that by Sands to ensure total correctness of transformed programs [San95]. Our strictness analysis can be considered as a simplified variant of disjunctive program analysis [Jen97], and its soundness can be proven in a similar spirit.

We have so far described the mechanism for tupling functions with single recursion arguments. More work needs to be done for devising effective tupling algorithm for functions with multiple recursion arguments in lazy programs. Here, it is necessary to consider synchronisation of changes of multiple arguments between two calls, in addition to the strictness information of the individual arguments. Handling of functions with multiple recursion arguments in strict languages has been described in [CKL98].

9 Conclusion

Most past work on tupling/tabulation techniques, such as those in [Coh83, Pet84, Chi93, HITT97], have focused mainly on the mechanics for realising big gains from the elimination of redundant calls. While impressive, there are still some doubts over how often redundant calls occur in practice. Functions which perform multiple traversals are likely to be more common. However, to the best of our knowledge, there have been no systematic investigation into the use of tupling for the elimination of multiple traversals. This work is useful as it could meaningfully complement the pool of existing optimising techniques for functional programs.

Applying this optimisation to lazy functional languages poses a particularly difficult challenge since naive elimination of multiple traversals could cause performance to degrade, rather than improve. We have demonstrated that due to laziness, such tupling results in the building of unnecessary closures. This is the bane of tupling and manifests itself in the penalty of both heap usage and execution time. To rectify this condition, we have proposed a range of solutions to minimise the building of these closures by forcing their evaluations when it is safe to do so, guided by strictness analysis. By using specialised strictness contexts where possible, we could guide our transformer to uncover more opportunites for effective tupling. Our tupling transformer could also exploit a more aggressive form of strictness, known as speculative strictness. As demonstrated, significant savings are possible, despite the potential for redundant over-evaluation.

It is important to note that the effect of strictness-guided tupling cannot be achieved by naive tupling, followed by (advanced) strictness analysis. Without the guidance of strictness contexts, blind introduction of tuple-functions could cause inefficient tupling to occur, from which strictness analysis alone could not recover.

There is little doubt that tupled functions are very 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 [Spa93].
- It can facilitate deforestation (and other transformations) that impose a *linearity* restriction [Wad88], used for efficiency and/or termination reasons.
- It can improve opportunity for *uniqueness typing* [BS93], 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. It is hoped that this burden on programmers could be relieved in the near future. We have a prototype implementation of our strictness-guided tupling algorithm for a restricted first-order language. At the current moment, it requires strictness signatures of user-defined functions to be given. For future work, we plan to extend our proposal to the full higher-order language, and provide for a type-based strictness inference analyser.

References

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

[Bir84] Richard S. Bird. Using circular programs to eliminate multiple traversals of data. *Acta Informatica*, 21:239–250, 1984.

[BS93] 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.

[Bur91] G.L. Burn. The evaluation transformer model of reduction and its correctness. In *TAPSOFT (LNCS)*, Brighton, 1991.

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

[CKL98] W.N. Chin, S.C. Khoo, and T.W. Lee. Synchronisation analyses to stop tupling. In *European Symposium on Programming (LNCS 1381)*, pages 75–89, April 1998.

[Coh83] Norman H. Cohen. Eliminating redundant recursive calls. *ACM Trans. on Programming Languages and Systems*, 5(3):265–299, July 1983.

[HITT97] Z. Hu, H. Iwasaki, M. Takeichi, and A. Takano. Tupling calculation eliminates multiple traversals. In *2nd ACM SIGPLAN International Conference on Functional Programming*, pages 164–175, Amsterdam, Netherlands, June 1997. ACM Press.

[Jen97] Thomas. Jensen. Disjunctive program analysis for algebraic data types. *ACM Trans. on Programming Languages and Systems*, 19(5):751–803, September 1997.

[MCH+92] S.A. Mahlke, W. Chen, W. Hwu, B. Ramakrishna Rau, and S. Schlansker. Sentinel scheduling for VLIW and superscalar processors. In *5th ASPLOS*, pages 238–247, Boston, Massachusetts, October 1992. ACM Press.

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

[PJP93] Simon Peyton-Jones and Will Partain. Measuring the effectiveness of a simple strictness analyser. In *Glasgow Functional Programming Workshop*, pages 201–220, Springer-Verlag Workshop Series, 1993.

[PJPS96] S. Peyton-Jones, W. Partain, and A. Santos. Let-floating: moving bindings to give faster programs. In *ACM SIGPLAN International Conference on Functional Programming*, Philadelphia, Pennsylvannia, May 1996. ACM Press.

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

[San95] David Sands. Total correctness by local improvement in program transformation. In *22nd ACM Principles of Programming Languages Conference*, pages 221–232. ACM Press, January 1995.

[Spa93] 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.

[Wad87] Phil Wadler. Strictness analysis in non-flat domains (by abstract interpretation over finite domains). In A. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*. Ellis Horwood, Chichester, UK, 1987.

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