Towards a Modular Program Derivation via Fusion and Tupling

Wei-Ngan Chin[1,2] and Zhenjiang Hu[3,4]

[1] National University of Singapore
[2] Singapore-MIT Alliance
chinwn@comp.nus.edu.sg
[3] University of Tokyo
[4] Presto 21, Japan Science and Technology Corporation
hu@ipl.t.u-tokyo.ac.jp

Abstract. We show how programming pearls can be systematically derived via fusion, followed by tupling transformations. By focusing on the elimination of intermediate data structures (fusion) followed by the elimination of redundant calls (tupling), we systematically realise both space and time efficient algorithms from naive specifications. We illustrate our approach using a well-known maximum segment sum (MSS) problem, and a less-known maximum segment product (MSP) problem. While the two problems share similar specifications, their optimised codes are significantly different. This divergence in the transformed codes do not pose any difficulty. By relying on modular techniques, we are able to systematically reuse both code and transformation in our derivation.

1 Introduction

A major impetus for highlighting programming pearls is to better understand how elegant and efficient algorithms could be invented. While creative algorithms are interesting to exhibit, they often lose their links to the programming techniques that were employed in their discoveries.

A more motivated approach to programming pearls is to formally derive creative algorithms from naive specifications. While elegant, most program derivations require deep insights to obtain major efficiency jumps for the transformed code. This can make it particularly difficult for human to comprehend, and machine to implement. In this paper, we show that it is possible to minimise some of these insights, and provide a systematic and modular approach towards discovering programming pearls.

Consider the maximum segment product problem. Given a list $[x_1, \ldots, x_n]$, we are interested to find the maximum product of all non-empty (contiguous) segments (of the form $[x_i, x_{i+1}, \ldots, x_j]$ where $1 \leq i \leq j \leq n$) taken from the input list. An initial specification for this problem can be written, as follows:

$msp(xs) \quad = max(map(prod,segs(xs)))$

As defined below, the innermost *segs* call returns a complete list of all segments, while the *map* call applies *prod* to each segment to yield its product, before the outermost *max* call chooses the largest value.

$$
\begin{array}{ll}
segs([x]) & = [[x]] \\
segs(x{:}xs) & = inits(x{:}xs) +\!\!+ segs(xs) \\
inits([x]) & = [[x]] \\
inits(x{:}xs) & = [x]{:}map((x{:}),inits(xs)) \\
map(f,Nil) & = Nil \\
map(f,x{:}xs) & = f(x){:}map(f,xs) \\
prod([x]) & = x \\
prod(x{:}xs) & = x * prod(xs) \\
max([x]) & = x \\
max(x{:}xs) & = max2(x,max(xs)) \\
max2(x,v) & = \text{if } v>x \text{ then } v \text{ else } x
\end{array}
$$

The above specification uses modular and reusable coding. Through the use of functions, such as *segs*, *inits*, *map*, *max* and *prod*, we can specify the *msp* function via straightforward composition of simpler functions. These high level specification are easier to comprehend and more reusable. For example, the better known maximum segment sum problem [Ben86] can be specified by replacing the *prod* call with *sum*, as follows:

$$
\begin{array}{ll}
mss(xs) & = max(map(sum,segs(xs))) \\
sum([x]) & = x \\
sum(x{:}xs) & = x+sum(xs)
\end{array}
$$

Unfortunately, high-level specifications have one major drawback, namely that they may be terribly inefficient. Fortunately, it is possible to use transformation to calculate efficient algorithms, that are usually unintuitive.

Our thesis is that high-level transformation techniques can provide a systematic approach to discovering programming pearls with good time and space behaviours. To substantiate this claim, we propose to apply two key transformation techniques, namely (i) *fusion* enhanced with laws, and (ii) *tupling*. The insights needed by our derivation are mainly confined to the fusion technique, in the form of laws needed to facilitate its transformation.

To appreciate the virtues of the transformational approach, the reader may want to try invent an efficient algorithm for maximum segment product, before studying the rest of this paper. We had some difficulties, until we embark on the transformational approach.

The main contributions in this paper are:

- We propose a *modular* derivation that supports the reuse of codes and transformation techniques. Particularly, we highlight two important transformation techniques, fusion and tupling, which in combination can be surprisingly good for deriving efficient algorithms.
- Our derivation is more *systematic*, minimizing the use of complex laws with deeper insights, such as Horner's rule in [Bir89]. Instead, we use a set of smaller laws which are motivated by the need to make fusion succeed. Most of these laws are *distributive* in nature.
- Our derivation is *powerful*. To the best of our knowledge, we demonstrate the first full and systematic derivation for the maximum segment product problem, without "suitable cunning" used in a previous derivation [Bir89].

- We show how *accumulation transformation*, known to invalidate *tupling* method, can be avoided.

In the rest of this paper, we first outline an enhanced fusion technique, which depends on laws, for its transformation (Sec 2). Later, we apply our modular approach, based on fusion and tupling, to a well-known maximum segment sum problem (Sec 3). We also highlight how a related but little known problem, called maximum segment product, can be similarly derived by our approach (Sec 4). We then compare with a classical derivation via Horner's rule (Sec 5), before an advice on the use of *accumulation* technique (Sec 6).

2 Enhanced Fusion with Laws

Fusion method [Chi92,TM95,CK01] is potentially a useful and prevalent transformation technique. Given a composition *f(g(x))* where *g(x)* yields an intermediate data structure for use by *f*, fusion would attempt to merge the composition into a specialised function *p(x)* with the same semantics as *f(g(x))* but without the need for an intermediate data structure.

In recent years, many attempts have been put forward to automate such fusion calculations [SF93,GLPJ93,SS97,HIT96]. Most current attempts are restricted to using equational definitions during transformation. For example, the deforestation algorithm [Wad88] relies on only define, unfold and fold rules [BD77] base on equational definitions. Unfortunately, this approach is inadequate since we often need laws (useful properties between functions, such as associativity and distributivity) to apply fusion successfully.

Consider a program which flattens a tree into a list, before finding its size.

$$
\begin{array}{ll}
\mathit{sizetree}(t) & = \mathit{length}(\mathit{flattree}(t)) \\
\mathit{flattree}(\mathit{Leaf}(a)) & = [a] \\
\mathit{flattree}(\mathit{Node}(l,r)) & = \mathit{flattree}(l) \mathbin{+\!\!+} \mathit{flattree}(r) \\
\mathit{length}(\mathit{Nil}) & = 0 \\
\mathit{length}(x{:}xs) & = 1 + \mathit{length}(xs)
\end{array}
$$

To optimise this program, we could try to fuse *length(flattree(t))*. However, this cannot be done without the distributive law of *length* over ++.

$$\mathit{length}(xr \mathbin{+\!\!+} xs) \quad = \mathit{length}(xr) + \mathit{length}(xs)$$

With this law, the fusion derivation of *sizetree* can be carried out, as follows:

$$
\begin{array}{lll}
\mathit{sizetree}(\mathit{Leaf}(a)) & = & \quad \{ \text{ instantiate } t{=}\mathit{Leaf}(a) \ \} \\
 & & \mathit{length}(\mathit{flattree}(\mathit{Leaf}(a))) \\
 & = & \quad \{ \text{ unfold } \mathit{flattree}\text{, then } \mathit{length} \ \} \\
 & & 1 \\
\mathit{sizetree}(\mathit{Node}(l,r)) & = & \quad \{ \text{ instantiate } t{=}\mathit{Node}(l,r)\text{, unfold } \mathit{flattree} \ \} \\
 & & \mathit{length}(\mathit{flattree}(l) \mathbin{+\!\!+} \mathit{flattree}(r)) \\
 & = & \quad \{ \text{ apply law (2) : } \mathit{length}(xr \mathbin{+\!\!+} xs){=}\mathit{length}(xr){+}\mathit{length}(xs) \ \} \\
 & & \mathit{length}(\mathit{flattree}(l)) + \mathit{length}(\mathit{flattree}(r)) \\
 & = & \quad \{ \text{ fold with } \mathit{sizetree} \text{ twice } \} \\
 & & \mathit{sizetree}(l) + \mathit{sizetree}(r)
\end{array}
$$

What was the rationale for using a distributive law during the above fusion? Informally, the inner *flattree* produces ++ calls during unfolding, which cannot be consumed by the pattern-matching equations of the outer *length* function. Instead, we need the distributive law of *length* over ++ to successfully consume ++ calls from the inner *flattree* function. A more detailed description of how laws may help fusion can be found in [Chi94].

These needed laws must either be supplied by programmers, or be derived via advanced synthesis techniques, such as [Smi89,CT97]. There are scope for automated help to synthesize (or check) these laws, but this issue is beyond the scope of the present paper. In the rest of this paper, we shall assume that relevant laws will be provided by users.

3 A Modular Derivation Strategy

We propose a modular derivation strategy based on two key transformation, namely fusion and tupling. To illustrate this strategy, consider:

mss(xs) = *max(map(sum,segs(xs)))*

The above specification has bad time and space complexities. If n is the size of the input list, then *mss* has a time complexity of $O(n^3)$. Note that *segs* returns $O(n^2)$ sub-lists which each requires $O(n)$ time to process by *sum*.

In general, space usage can be broken down into three parts:

- stack space for the function calls (such as *segs, map, sum*).
- heap space for input and output of main function (i.e. *mss*).
- heap space for intermediate data structures (by *segs, map* and *sum*).

We shall ignore the somewhat *fixed* space cost associated with the stack and input/output, but focus on the *variable* space cost due to intermediate data structures. In the case of *mss*, this space cost is due to *segs* generating $O(n^2)$ sub-lists of $O(n)$ length each, while *map* yields another intermediate list of size $O(n^2)$; giving a space complexity of $O(n^3)$.

Our strategy for deriving efficient algorithms via *fusion*, followed by *tupling* is outlined in Figure 1 for the MSS problem.

Fusion transformation is capable of eliminating all intermediate data structures for this example. During this fusion, we encountered another composition which was defined as the following new definition:

mis(xs) = *max(map(sum,inits(xs)))*

With the help of appropriate laws, both *mss* and *mis* functions can be transformed to a pair of new recursive functions, shown in Figure 1(b). The fused *mss* has a much improved O(1) variable space complexity. However, it still suffers from a time-complexity of O(n^2) due primarily to redundant *mis* calls. The redundant calls can be eliminated by tupling transformation. Firstly, define:

msstup(xs) = *(mss(xs),mis(xs))*

Subsequent transformation yields a new recursive tupled definition shown in Figure 1(c). Without any redundant calls, the new *msstup* definition has a time-complexity of O(n). We present the detailed derivations next.

$$\begin{array}{ll} & (a) \\ mss(xs) & = max(map(sum,segs(xs))) \end{array}$$

$\Downarrow$ **Fusion Tactic**
with $mis(xs)=max(map(sum,inits(xs)))$

$$\begin{array}{ll} & (b) \\ mss([x]) & = x \\ mss(x{:}xs) & = max2(max2(x,x+mis(xs)),mss(xs)) \\ mis([x]) & = x \\ mis(x{:}xs) & = max2(x,x+mis(xs)) \end{array}$$

$\Downarrow$ **Tupling Tactic**
with $msstup(xs)=(mss(xs),mis(xs))$

$$\begin{array}{ll} & (c) \\ mss(xs) & = let\ (u,\_)=msstup(xs)\ in\ u \\ msstup([x]) & = (x,x) \\ msstup(x{:}xs) & = let\ \{(u,v)=msstup(xs);\ b=max2(x,x+v)\} \\ & \quad in\ (max2(b,u),b) \end{array}$$

Fig. 1. Modular Derivation Strategy via Fusion and Tupling

3.1 Fusion to Remove Intermediate Data Structures

The enhanced fusion method relies on laws, in addition to the supplied equation, for its transformation. We would like to stress again that these laws do not come from thin air, but are instead motivated by the need to perform fusion. In the case of *mss*, we need the following *distributive* laws.

$$map(f,xr{+\!\!+}xs) = map(f,xr){+\!\!+}map(f,xs) \tag{1}$$

$$max(xr{+\!\!+}xs) = max2(max(xr),max(xs)) \tag{2}$$

$$map(f,map(g,xs)) = map(f \circ g,xs)\ where\ (f \circ g)(x) = f(g(x)) \tag{3}$$

$$max(map((x+),xs)) = x+max(xs) \tag{4}$$

The first two laws are distributive laws of *map* and *max* over the $+\!\!+$ operator, while law (3) distributes over an inner *map* call (or over function composition if used backwards). The last law is concerned with the distributivity of *max* over an *(x+)* call that is being applied to each element of its input list. A more general version of this last law can be constructed in conjunction with law (3), as follows:

$$max(map((x+) \circ g,xs)) = x+max(map(g,xs)) \quad (5)$$

Fusion/deforestation method makes use of normal-order transformation strategy [SGN94], to merge functional compositions. In the case of *mss*, the outermost *max* call demands an output from an inner *map* call, which in turn demands an output from *segs*. Thus, the innermost *segs(xs)* call is selected for unfolding. This can be done via two possible instantiations to its argument, *xs*. The base case instantiation results in:

mss([x])	=	{ instantiate *xs=[x]* }
		max(map(sum,segs([x])))
	=	{ unfold *segs, map, max, sum* }
		x

For the recursive case instantiation, the *segs* function actually produces ++ calls which must be consumed by *map*, as follows:

mss(x:xs)	=	{ instantiate *xs=x:xs*,unfold *segs* }
		max(map(sum,inits(x:xs)++segs(xs)))
	=	{ apply law (1) : *map(f,xr++xs) = map(f,xr)++map(f,xs)* }
		max(map(sum,inits(x:xs))++map(sum,segs(xs)))

Another ++ operator is produced by the distributive law of *map* itself. This must in turn be consumed via the distributive law of *max*, as follows:

mss(x:xs)	=	{ apply law (2) : *max(xr++xs)=max2(max(xr),max(xs))* }
		max2(max(map(sum,inits(x:xs))),max(map(sum,segs(xs))))

At this point, *max(map(sum,segs(xs)))* is a re-occurrence of the definition for *mss* which can be handled using a fold operation. Also, *max(map(sum,init(xs)))* represents a new composed expression just encountered. We could introduce a new function, say *mis*, to denote it and then obtain:

mss(x:xs)	=	{ fold with *mss* }
		max2(max(map(sum,inits(x:xs))),mss(xs))
	=	{ fold with a new *mis*, then unfold }
		max2(max2(x,s+mis(xs)),mss(xs))

The new composition encountered is captured by:

mis(xs) = *max(map(sum,inits(xs)))*

We re-apply fusion transformation, by beginning with an unfold of *inits(xs)* using the two possible instantiation to *xs*. A similar sequence of transformations via unfolding, application of laws, and folding yield the following equations.

mis([x]) = *x*
mis(x:xs) = *max2(x,x+mis(xs))*

The primary gain from fusion method is the complete elimination of intermediate data structures. This results in an improved time complexity of $O(n^2)$, and a much improved variable space complexity of $O(1)$.

3.2 Tupling to Eliminate Redundant Calls

After fusion, our program may have redundant function calls. This inefficiency can be overcome by the tupling method [Chi93,HITT97] which essentially gathers calls with overlapping arguments together. In the case of *mss*, we find two calls with identical arguments in its recursive equation. Tupling would gather these two calls, as follows.

msstup(xs) $=$ *(mss(xs),mis(xs))*

This can then be instantiated and further transformed, as follows:

msstup([x]) $=$ { instantiate *xs=[x]* }
(mss([x]),mis([x]))
$=$ { unfold *mss* & *mis* }
(x,x)

The recursive case instantiation and transformation is outlined below.

msstup(x:xs) $=$ { instantiate *xs=x:xs* }
(mss(x:xs),mis(x:xs))
$=$ { unfold *mss* & *mis* }
(max2(max2(x,x+mis(xs)),mss(xs)),max2(x,x+mis(xs)))
$=$ { gather *mss* and *mis* calls using let }
let (u,v)=(mss(xs),mis(xs)) in (max2(max2(x,x+v),u),max2(x,x+v))
$=$ { fold with *msstup*, and abstract *b* }
let {(u,v)=msstup(xs); b=max2(x,x+v)} in (max2(b,u),b)

Note the use of a gathering step to collect calls with overlapping arguments, resulting in a tuple of two calls. This can later be folded against *msstup*. The redundant occurrences of *mis* call was eventually shared by such a tuple gathering step. The end result is an efficient linear time $O(n)$ algorithm for maximum segment sum.

4 Maximum Segment Product

Let us now turn our attention to a related but less-known problem for finding maximum segment product (MSP). This MSP problem was proposed by Richard Bird in the 1989 STOP Summer School [Bir89]. It is of interests because its specification is closely related to the MSS problem, but yet its efficient implementation is considerably more complex.

For its specification and transformation, we can reuse all functions and laws used by *mss*, with the exception of those related to *sum* and $+$. Specifically, the distributive law of *max* over *map* needs to be replaced by corresponding laws over *(x∗)*. This property can be specified by a pair of laws, namely:

$$max(map((x*),xs)) = \text{if } x\geq 0 \text{ then } x*max(xs) \text{ else } x*min(xs) \tag{6}$$

$$min(map((x*),xs)) = \text{if } x\geq 0 \text{ then } x*min(xs) \text{ else } x*max(xs) \tag{7}$$

Note the need for *min*, as a dual of *max*, with definition:

min([x]) $=$ *x*
min(x:xs) $=$ *min2(x,min(xs))*
min2(x,v) $=$ *if v<x then v else x*

Why is *min* needed? Consider the expression $x{*}b$ where b is taken from a list. If x is negative, then the value of $x{*}b$ would be maximal if the selected element b is of smallest value. More practical versions of the above pair of laws are obtained by combining them with law (3), as shown below.

$$max(map((x*) \circ f,xs)) = if\ x{\geq}0\ then\ x{*}max(map(f,xs))\ else\ x{*}min(map(f,xs)) \quad (8)$$
$$min(map((x*) \circ f,xs)) = if\ x{\geq}0\ then\ x{*}min(map(f,xs))\ else\ x{*}max(map(f,xs)) \quad (9)$$

With the help of these two extra laws, we can perform a similar fusion transformation on the naive specification for *msp*. Recall:

msp(xs) = *max(map(prod,segs(xs)))*

The base case equation is easily derived, as follows.

msp([x]) = { instantiate *xs=[x]* }
max(map(prod,segs([x])))
= { unfold *segs, map, max, prod* }
x

The recursive case equation can be derived, as outlined below.

msp(x:xs) = { instantiate *xs=x:xs* }
max(map(prod,inits(x:xs)++segs(xs)))
= { apply law (1) & law (2) }
max2(max(map(prod,inits(x:xs))),max(map(prod,segs(xs))))
= { fold with *msp* }
max2(max(map(prod,inits(x:xs))),msp(xs))
= { fold with a new defn for *mip* }
max2(mip(x:xs),msp(xs))

A new composed expression, defined as *mip*, was encountered.

mip(xs) = *max(map(prod,inits(xs)))*

Similar fusion derivation results in:

mip([x]) = *x*
mip(x:xs) = { instantiate *xs=x:xs* & fuse }
$max2(x,if\ x{\geq}0\ then\ x{*}mip(xs)\ else\ x{*}mipm(xs))$
= { apply law (10) to float *if* outwards }
$if\ x{\geq}0\ then\ max2(x,x{*}mip(xs))\ else\ max2(x,x{*}mipm(xs))$

The last step floats an inner *if* out of the outermost *max2* call. This transformation can be effected by the following generic law where *E[]* denotes an arbitrary expression context with a hole. (Its floatation can facilitate the elimination of common *if* test during tupling transformation, as shown later.)

$$E[if\ e_1\ then\ e_2\ else\ e_3] = if\ e_1\ then\ E[e_2]\ else\ E[e_3] \quad (10)$$

Another composition encountered, $x{*}min(map(prod,inits(xs)))$, was defined as:

mipm(xs) = *min(map(prod,inits(xs)))*

Its fusion derivation for *mipm* is very similar to *mip*, and results in:

```
mipm([x])       = x
mipm(x:xs)      = if x≥0 then min2(x,x*mipm(xs)) else min2(x,x*mip(xs))
```

Tupling analysis of [Chi93,HITT97] would reveal that there are redundant calls to *mip* and *mipm*, which can be eliminated by gathering the following tuple of calls.

```
msptup(xs)  = (msp(xs),mip(xs),mipm(xs))
```

Subsequently, tupling transformation can be applied as follows:

```
msptup([x])  =    { instantiate xs=[x] }
                (msp([x]),mip([x]),mipm([x]))
             =    { unfold msp, mip & mipm }
                (x,x,x)
msptup(x:xs) =    { instantiate xs=x:xs }
                (msp(x:xs),mip(x:xs),mipm(x:xs))
             =    { unfold msp, mip, mipm and floats if over tuple structure }
                if x≥0 then (max2(max2(x,x*mip(xs)),msp(xs))
                            ,max2(x,x*mip(xs)),min2(x,x*mipm(xs)))
                else (max2(max2(x,x*mipm(xs)),msp(xs)),max2(x,x*mipm(xs))
                            ,,min2(x,x*mip(xs)))
             =    { gather msp, mip and mipm calls using let }
                let (u,v,w)=(msp(xs),mip(xs),mipm(xs)) in
                if x≥0 then (max2(max2(x,x*v),u),max2(x,x*v),min2(x,x*w))
                else (max2(max2(x,x*w),u),max2(x,x*w),min2(x,x*v))
             =    { fold with msptup }
                let (u,v,w)= msptup(xs) in
                if x≥0 then (max2(max2(x,x*v),u),max2(x,x*v),min2(x,x*w))
                else (max2(max2(x,x*w),u),max2(x,x*w),min2(x,x*v))
             =    { abstract & share common sub-expressions }
                let { (u,v,w)= msptup(xs); r=x*v; s=x*w; b=max2(x,r);
                      d=max2(x,s)} in if x≥0 then (max2(b,u),b,min2(x,s))
                                      else (max2(d,u),d,min2(x,r))
```

The final optimised program is:

```
msp(xs)      = let (u,_,_)=msptup(xs) in u
msptup([x])  = (x,x,x)
msptup(x:xs) = let { (u,v,w)= msptup(xs); r=x*v; s=x*w; b=max2(x,r);
                     d=max2(x,s)} in if x≥0 then (max2(b,u),b,min2(x,s))
                                     else (max2(d,u),d,min2(x,r))
```

The derived algorithm for *msptup* is more complex than that for *msstup*, even though their initial specifications are similar. However, we used essentially the same transformation techniques, namely fusion followed by tupling. We reiterate that fusion helps to eliminate intermediate data structures (improving on space), while tupling helps to eliminate redundant calls (improving on time).

Compared to *mss* derivation, only two extra laws, that allow distribution of *max* (and *min*) over products, are needed to systematically derive a more intricate, but yet efficient algorithm for the MSP problem. An alternative derivation proposed by Bird, requires a somewhat deeper insight based on Horner's rule. This approach is considerably more complex since the corresponding Horner's rule have to be invented over tupled functions (for the MSP problem). Let us review the Horner's rule approach.

5 Classical Derivation via Horner's rule

The MSS problem originated from Bentley [Ben86]. Formal derivation to obtain efficient linear-time algorithm was developed by Bird [Bir88], amongst others.

The traditional derivation for the MSS problem has been based on function-level reasoning via the Bird-Meerstens Formalism (BMF). A major theme of the BMF approach is to capture common patterns of computations via higher-order functions, and to make heavy use of laws/theorems concerning these operations. Often, algebraic properties on the components of higher-order operations are required as side-conditions.

An important example is the Horner's rule to reduce the number of operations used for polynomial-like evaluation. A special case of this rule/law (instantiated to three terms) can be stated as:

$$(a_1 \otimes (a_2 \otimes a_3)) \oplus ((a_2 \otimes a_3) \oplus a_3) = ((a_1 \oplus 1_\otimes) \otimes a_2 \oplus 1_\otimes) \otimes a_3$$

The algebraic side-conditions required are that both $\oplus$ and $\otimes$ be associative, $1_\otimes$ be the left identity of $\otimes$, and $\otimes$ distributes through $\oplus$. To generalise to n terms, we could express this rule as:

$$(\oplus\ /)\ map(\otimes\ /,\ tails([a_1, \ldots, a_n])) = \circledast \not\rightarrow_{1_\otimes} [a_1, \ldots, a_n] \quad (11)$$

where the operators $\circledast$, $/$, $\not\rightarrow$ and *tails* are defined by:

$a \circledast b$	$= (a \otimes b) \oplus 1_\otimes$
$\odot\ /\ [x]$	$= x$
$\odot\ /\ (xs{+\!\!+}ys)$	$= (\odot\ /\ xs) \odot (\odot\ /\ ys)$
$\odot \not\rightarrow_e Nil$	$= e$
$\odot \not\rightarrow_e (xs{+\!\!+}[y])$	$= (\odot \not\rightarrow_e xs) \odot y$
$tails(Nil)$	$= Nil$
$tails(x{:}xs)$	$= (x{:}xs){:}tails(xs)$

Horner's rule is a key insight used in the calculational derivation of *mss* in [Bir88]. We re-produce this classical derivation below.

$$\begin{array}{rl}
mss(xs) = & (max2\ /)(map((+\ /),segs'(xs))) \\
= & \quad \{ \text{ unfold } segs'(xs)=flatten(map(tails,inits(xs))) \ \} \\
 & (max2\ /)(map((+\ /),flatten(map(tails,inits(xs))))) \\
= & \quad \{ \text{ apply law : } map(f,flatten(xss))=flatten(map(\backslash xs.map(f,xs),xss)) \ \} \\
 & (max2\ /)(flatten(map(\backslash z.map((+\ /),z),map(tails,inits(xs))))) \\
= & \quad \{ \text{ apply law : } max(flatten(xss))=max(map(max,xss)) \ \} \\
 & (max2\ /)(map((max2\ /),map(\backslash z.map((+\ /),z),map(tails,inits(xs))))) \\
= & \quad \{ \text{ apply law : } map(f,map(g,xs)) = map(f \circ g,xs) \text{ twice } \} \\
 & (max2\ /)(map(\backslash y.\ (max2\ /)(map(\backslash z.map((+\ /),z),tails(y))),\ inits(xs))) \\
= & \quad \{ \text{ apply Horner's rule : } (\oplus\ /)map(\otimes\ /,tails\ xs) = \circledast \not\rightarrow_{1_\otimes} xs \ \} \\
 & (max2\ /)(map(\circledast \not\rightarrow_0,\ inits(xs)))\ where\ a \circledast b = max2(a{+}b,0) \\
= & \quad \{ \text{ apply scan law : } map(\odot \not\rightarrow_0,inits(xs))=\odot \not\!\!\!\!\rightarrow\!\!\!\!\!\!\rightarrow_0 xs \ \} \\
 & (max2\ /)(\circledast \not\!\!\!\!\rightarrow\!\!\!\!\!\!\rightarrow_0 xs)
\end{array}$$

Note that we used a non-recursive definition of *segs'* which returns segments in a different order (from *segs* given in Sec. 1). We used:

$\odot \not\!\!\!\!\!\!\#_e\ Nil \quad = [e]$
$\odot \not\!\!\!\!\!\!\#_e\ (xs{+\!\!+}[y]) \quad = (\odot \not\!\!\!\!\!\!\#_e\ xs) {+\!\!+} [last(\odot \not\!\!\!\!\!\!\#_e\ xs) \odot y]$
$last(xs{+\!\!+}[y]) \quad = y$
$flatten(Nil) \quad = Nil$
$flatten(xs{:}xss) \quad = xs{+\!\!+}flatten(xss)$

The final algorithm obtained for *mss* has a linear time complexity, and also a linear (variable) space complexity due to an intermediate list from $(\circledast \not\!\!\!\!\!\!\#_0\ xs)$. This slightly worse space behaviour may be improved by fusion transformation. Although this classical derivation looks more concise that our proposed derivation, it requires more insightful steps with non-trivial side conditions.

For example, the Horner's rule for *MSS* problem requires that $+$ distributes through *max2*, and that the identity of $+$, namely *0*, be present. (The use of *0* as the identity of $+$ actually results in a less defined *mss* algorithm since it becomes ill-defined for lists with only negative numbers. But this shortcoming is often tolerated.) Worse still is the possibility that distributive property required may not be immediately detected, but support for such property may come from generalised/tupled functions instead. Consider the MSP problem. The $*$ operator does not distribute over *max2* for negative numbers, but we do have:

$$max2(a,b)*c = if\ x{\geq}0\ then\ max2(a*c,b*c)\ else\ min2(a*c,b*c)$$
$$min2(a,b)*c = if\ x{\geq}0\ then\ min2(a*c,b*c)\ else\ max2(a*c,b*c)$$

As Bird reported : *"These facts are enough to ensure that, with suitable cunning, Horner's rule can be made to work"*[Bir89]. Instead of *max2* and * as instantiations of $\oplus$ and $\otimes$ operators for its Horner's rule, a more insightful tupled functions was used instead.

$(a_1, b_1) \oplus (a_2, b_2) \quad = (min2(a_1, a_2), max2(b_1, b_2))$
$(a,b) \otimes c \quad = if\ c \geq 0\ then\ (a*c,b*c)\ else\ (b*c,a*c)$

With the above, we can now prove that $\otimes$ distributes through $\oplus$:

$$((a_1, b_1) \oplus (a_2, b_2)) \otimes c = ((a_1, b_1) \otimes c) \oplus ((a_2, b_2) \otimes c))$$

Inventive insights are needed to come up with such tupled functions for MSP-like problems. In addition, the original definition of *msp* has to be rewritten to use such tupled functions before its calculational derivation can be applied. The main difficulty stems from the highly abstract nature of Horner's rule and its algebraic side-conditions. Fortunately, our proposal avoids this problem by decomposing the derivation into fusion (which requires the distributive conditions), followed by tupling (to eliminate redundant calls). Such separation allows difficult theorems/insights to be dispensed by simpler transformation techniques.

6 Avoiding Accumulation to Save Tupling

The perceptive reader may noticed that our specification of *mss* differs slightly from [Bir89]. Specifically, the classical definition of *mss* generates segments via:

$segs'(xs) \quad = flatten(map(tails,inits(xs)))$

In contrast, we started with the following definition before it was fused to the recursive definition given in Sec 1.

$segs(xs) \quad = flatten(map(inits,tails(xs)))$

Both *segs'* and *segs* yield the same set of segments, except for their order. Unfortunately, this innocous change seems to have an effect on the kind of derivations which can be performed.

For example, if *segs* were used by the classical derivation, we will need a different type of Horner' rules, that are oriented for right-to-left reductions. Correspondingly, if *segs'* were used by our modular approach to derivation, we require equations based on right-to-left evaluation. These equations are typically referred to as *snoc*-based equations (which deconstruct a given list backwards), instead of the usual *cons*-based equations.

At this point, two questions may puzzle the reader : How do we obtain such *snoc*-based equations? When should we use them?

The *snoc*-based equations can be obtained as a by-product of parallelization. Given a *cons*-based equation, the inductive parallelization method presented in [HTC98] is capable of (automatically) deriving a $+\!\!+$-based parallel equation. This can subsequently be instantiated to the *snoc*-based equation. As an example, consider the *cons*-based version of *inits* function given in Sec 1. Using the method of [HTC98], it is possible to derive the following $+\!\!+$-based parallel equation.

$inits(xs+\!\!+ys) \quad = inits(xs)+\!\!+map((xs+\!\!+),inits(xs))$

By instantiating *ys* to *[y]*, we can obtain the following *snoc*-based equation.

$inits(xs+\!\!+[y]) \quad = inits(xs)+\!\!+[xs+\!\!+[y]]$

The second question is *when should we use such snoc-based equations*? We should consider them when our fusion technique is about to fail through the application of an *accumulation* tactic – which is known be unhelpful to tupling! For example, consider the fusion of *segs'* below.

$$
\begin{array}{lll}
segs'(x{:}xs) & = & \{ \text{ instantiate } xs{=}x{:}xs \ \} \\
 & & flatten(map(tails,inits(x{:}xs))) \\
 & = & \{ \text{ unfold } inits,\ map,\ flatten \ \} \\
 & & tails([x])+\!\!+flatten(map(tails,map((x{:}),inits(xs)))) \\
 & = & \{ \text{ apply law (3) : } map(f,map(g,xs)) = map(f \circ g,xs) \ \} \\
 & & tails([x])+\!\!+flatten(map(tails \circ (x{:}),inits(xs)))
\end{array}
$$

After several steps, we are still unable to fold as we encountered a slightly enlarged expression of the form $flatten(map(tails \circ (x{:}),inits(xs))$. As reported elsewhere [Bir84] and [HIT99], this calls for the use of an *accumulation tactic* which generalizes $(x{:})$ to $(w+\!\!+)$:

$asegs'(w,xs) \quad = flatten(map(tails \circ (w+\!\!+),inits(xs)))$

A subsequent fusion transformation obtains:

$$
\begin{array}{ll}
asegs'(w,[x]) & = tails(w+\!\!+[x]) \\
asegs'(w,x{:}xs) & = tails(w+\!\!+[x])+\!\!+asegs'(w+\!\!+[x],xs)
\end{array}
$$

In general, this accumulation tactic is bad for two reasons. Firstly, the presence of an accumulating (list) parameter indicates that fusion is not totally successful (having failed for the accumulating parameter). Secondly, the resulting function (with an accumulating parameter) is **bad** for tupling since its redundant calls may have infinitely many variants of the accumulative arguments during transformation. This reduces the chances of successful folding. As a result, we are unable to apply tupling to *asegs'* (or its *mss* counterpart) to eliminate the redundant *tails* calls (or its *mis*-like counterparts). A key lesson is – *avoid (or delay) the application of accumulating tactic, where possible.* One way to avoid accumulation is to turn to *snoc*-based equations, whenever the use of accumulation is inevitable. In the case of *segs'*, the corresponding fusion transformation using *snoc*-based equations can proceed (without accumulation), as shown:

$$
\begin{array}{lll}
segs'(xs{+\!\!+}[y]) & = & \{\text{ instantiate } xs{=}xs{+\!\!+}[y] \;\} \\
 & & flatten(map(tails,inits(xs{+\!\!+}[y]))) \\
 & = & \{\text{ unfold } inits\text{,apply law (1) } \} \\
 & & flatten(map(tails,inits(xs)){+\!\!+}map(tails,[xs{+\!\!+}[y]])) \\
 & = & \{\text{ apply law : } flatten(xr{+\!\!+}xs) = flatten(xr){+\!\!+}flatten(xs) \;\} \\
 & & flatten(map(tails,inits(xs))){+\!\!+}flatten(map(tails,[xs{+\!\!+}[y]])) \\
 & = & \{\text{ fold with } segs'\text{, unfold } map,\ flatten \;\} \\
 & & segs'(xs){+\!\!+}\ [tails(xs{+\!\!+}[y])]
\end{array}
$$

With this version of *segs'*, the main *mss* function can be transformed to:

$$
\begin{array}{ll}
mss([x]) & = x \\
mss(xs{+\!\!+}[y]) & = max2(mss(xs),max2(mis(xs)+y,y)) \\
mis([x]) & = x \\
mis(xs{+\!\!+}[y]) & = max2(mis(xs)+y,y)
\end{array}
$$

The redundant calls in the above fused program can now be eliminated via tupling without being hindered by accumulating parameters.

7 Discussion and Concluding Remarks

Fusion transformation is considered to be one of the most important derivation technique in the constructive algorithmics [Bir89,Fok92], with many useful fusion theorems being developed for deriving various classes of efficient programs (A good summary of these theorems can be found in [Jeu93]). In contrast, the importance of tupling transformation technique [Fok89] for program derivation was hardly addressed, let alone a good combination of fusion and tupling.

In this paper, we have proposed a new strategy for algorithm derivation through two key transformation techniques. Our strategy provides a more modular derivation with two key phases, for the eliminations of intermediate data and redundant calls, respectively. While the steps taken are smaller than the traditional BMF approach, the opportunities for mechanisation are high since we rely on less insightful laws/theorems. In particular, only simple distributive laws are needed in the enhanced fusion process, while tupling depends on only equational definitions for its transformation. This combination of fusion (with laws) and tupling is particularly powerful. Finding a good collection of modular transformation techniques could provide an improved methodology for developing programming pearls.

References

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

[Ben86] Jon Bentley. *Programming Pearls*. Addison-Wesley, 1986.

[Bir84] Richard S. Bird. The promotion and accumulation strategies in transformational programming. *ACM Trans. on Programming Languages and Systems*, 6(4):487–504, October 1984.

[Bir88] Richard S. Bird. *Lectures on Constructive Functional Programming*. Springer-Verlag, 1988.

[Bir89] Richard S. Bird. Lecture notes on theory of lists. In *STOP Summer School on Constructive Algorithmics, Abeland*, pages 1–25, 9 1989.

[Chi92] Wei-Ngan Chin. Safe fusion of functional expressions. In *7th ACM LISP and Functional Programming Conference*, pages 11–20, San Francisco, California, June 1992. ACM Press.

[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.

[Chi94] Wei-Ngan Chin. Safe fusion of functional expressions II: Further improvements. *Journal of Functional Programming*, 4(4):515–555, October 1994.

[CK01] Manuel MT Chakravarty and Gabriele Keller. Functional array fusion. In *ACM Intl. Conference on Functional Programming*, pages 205–216, Florence, Italy, September 2001. ACM Press.

[CT97] W.N. Chin and A. Takano. Deriving laws by program specialization. Technical report, Hitachi Advanced Research Laboratory, July 1997.

[Fok89] M. Fokkinga. Tupling and mutumorphisms. *Squiggolist*, 1(4), 1989.

[Fok92] M. Fokkinga. *Law and Order in Algorithmics*. Ph.D thesis, Dept. INF, University of Twente, The Netherlands, 1992.

[GLPJ93] A. Gill, J. Launchbury, and S. Peyton-Jones. A short-cut to deforestation. In *6th ACM Conference on Functional Programming Languages and Computer Architecture*, Copenhagen, Denmark, June 1993. ACM Press.

[HIT96] Z. Hu, H. Iwasaki, and M. Takeichi. Deriving structural hylomorphisms from recursive definitions. In *ACM SIGPLAN International Conference on Functional Programming*, pages 73–82, Philadelphia, Pennsylvannia, May 1996. ACM Press.

[HIT99] Z. Hu, H. Iwasaki, and M. Takeichi. Calculating accumulations. *New Generation Computing*, 17(2):153–173, 1999.

[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.

[HTC98] Z. Hu, M. Takeichi, and W.N. Chin. Parallelization in calculational forms. In *25th Annual ACM Symposium on Principles of Programming Languages*, pages 316–328, San Diego, California, January 1998. ACM Press.

[Jeu93] J. Jeuring. *Theories for Algorithm Calculation.* Ph.D thesis, Faculty of Science, Utrecht University, 1993.

[JH99] S. Peyton Jones and J. Hughes, editors. *Haskell 98: A Non-strict, Purely Functional Language.* Available online: `http://www.haskell.org`, February 1999.

[SF93] T. Sheard and L. Fegaras. A fold for all seasons. In *6th ACM Conference on Functional programming Languages and Computer Architecture*, Copenhagen, Denmark, June 1993. ACM Press.

[SGN94] M.H. Sørensen, R. Glück, and Jones N.D. Towards unifying deforestation, supercompilation, partial evaluation and generalised partial computation. In *European Symposium on Programming (LNCS 788)*, Edinburgh, April 1994.

[Smi89] Douglas R. Smith. KIDS - a semi-automatic program development system. Technical report, Kestrel Institute, October 1989.

[SS97] 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.

[TH00] M. Takeichi and Z. Hu. Calculation carrying programs: How to code program transformations (invited paper). In *International Sumposium on Principles of Software Evolution (ISPSE 2000)*, Kanazawa, Japan, November 2000. IEEE Press.

[TM95] A. Takano and E. Meijer. Shortcut deforestation in calculational form. In *ACM Conference on Functional Programming and Computer Architecture*, pages 306–313, San Diego, California, June 1995. ACM Press.

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

[YHT02] T. Yokoyama, Z. Hu, and M. Takeichi. Yicho: A system for programming program calculations. Technical Report METR 2002–07, Department of Mathematical Engineering, University of Tokyo, June 2002. submitted for publication.

Appendix : Implementing Modular Derivation using Yicho

The modular derivation approach via fusion and tupling can be easily implemented using the Yicho system [YHT02], a system supporting the coding of calculation carrying programs [TH00] that can relax the tension between efficiency and clarity in programming. The main idea is to accompany clear programs with appropriate calculations describing intention of how to manipulate programs to be efficient. Calculation specification can be executed automatically by the Yicho system to derive efficient programs.

To illustrate, we give the complete code for solving the maximum segment sum problem. The initial program is first coded in Haskell [JH99], using curried syntax. The laws used for fusion (Section 3.1) are coded as follows. The names of meta variables are prefixed with `%`, and `# e1 -> e2` denotes meta lambda abstraction (over object expression).

```
%mss_law = %map_append <+
           %max_append <+
```

```
            %map_distribution <+
            %max_add
%map_append = # (map %f (%xs ++ %ys)) -> map %f %xs
                                          ++ map %f %ys;
%max_append = # (max (%xs ++ %ys)) -> max2 (max %xs) (max %ys);
%map_distribution = # (map %f (map %g %xs)) ->
                                      map (%f . %g) %xs;
%max_add = # (max (map ((+) %x) %xs)) -> %x + max %xs;
```

The modular derivation of tupling transformation after fusing function `f` with another auxiliary function `aux` using `law` can be described by

```
%fusing_tupling2 = # %law %f %aux ->
    letm
         -- fusion --
         {%e1 = %rec %law (%f []);
          \a x -> %h1 a (%f x,%aux x) =
                \a x -> %rec %law (%f (a:x));
          %e2 = %rec %law (%aux []);
          \a x -> %h2 a (%f x,%aux x) =
                \a x -> %rec %law (%aux (a:x))}
    in
         -- tupling --
         let %h = \a (x,y) -> (%h1 a (x,y), %h2 a (x,y))
         in fst (foldr %h (%e1,%e2));
```

We will not explain this calculation program in detail, where higher-order pattern matching plays an important role [YHT02,TH00] in the implementation of fusion transformation. The code fragment:

```
\a x -> %h1 a (%f x,%g x) = \a x -> %rec %law (%f (a:x));
```

is intended to apply law `%law` recursively to transform the expression `%f (a:x)`, and then match the result with a higher-order pattern `%h1 a (%f x, %aux x)` to get a definition for `%h1`. With these definitions, we can execute the expression

```
%fusing_tupling2 %mss_law mss mis
      where mis = max . map sum . inits;
```

on Yicho to obtain the efficient program for `mss` (as in Figure 1) in a fully automatic way. One can follow the above to implement the derivation for `msp`. The calculation code is almost the same, except for the laws and auxiliary functions.