# A Modular Derivation Strategy via Fusion and Tupling

Wei-Ngan Chin
National University of Singapore

Zhenjiang Hu
University of Tokyo

Masato Takeichi
University of Tokyo

**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 can 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 lesser-known maximum segment product (MSP) problem. While the two problems share similar specifications, their optimised codes are significantly different. This divergence in their transformed codes do not pose any difficulty for our approach. In fact, by relying on modular transformation techniques, we are able to systematically reuse both code and transformation in our derivation.

**Keywords:** *Fusion, Tupling, Program Derivation, Programming Pearls.*

## 1 Introduction

A major impetus for highlighting programming pearls is to provide a better understanding of how elegant and efficient algorithms could be build. While creative algorithms are interesting to exhibit, they often lose their links to the programming techniques that were employed in their discoveries. A more motivating approach to programming pearls would be to formally relate creative algorithms with naive specifications via program derivations.

While elegant, many examples of program derivations often require deep insights which make major changes/jumps to the transformed code. This can make things very difficult for human to comprehend, and machine to implement. In this paper, we shall 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 an input 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 in a modular fashion, as follows:

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

Here, 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. The functions used in the above specification are given below.

```
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)        = if v>x then v else x
```

The above specification uses a modular approach to coding. Through the reuse of abstract functions, such as *segs*, *inits*, *map*, *max* and *prod*, we are able to specify the *msp* function via a relatively straightforward composition of simpler functions. There are two main advantages for such high-level specifications. Firstly, they are clearer for human to comprehend and more obviously correct. Secondly, their more modular style encourages software reusability. For example, the better known maximum segment sum problem [Ben86] can be specified by replacing only the *prod* function with *sum*, as follows:

```
mss(xs)      = max(map(prod,segs(xs)))
sum([x])     = x
sum(x:xs)    = x+sum(xs)
```

Unfortunately, high-level specifications have one major drawback, namely that they can be terribly inefficient. Fortunately for us, it is possible to use the transformational approach to calculate efficient algorithms. This is potentially very useful since efficient algorithms can be very unintuitive.

Our thesis is that high-level transformation techniques can help provide a systematic approach to discover programming pearls. To substantiate this claim, we propose to apply two key transformation techniques, namely (i) *fusion* enhanced with laws, and (ii) *tupling*, to help derive algorithms with good time and space behaviours. The mild 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 initiated 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.

Our main contributions in this paper are summarized as follows.

- We propose a *modular* derivation that supports the reuse of codes and transformation techniques. The basis of our approach is the identification of a small set of commonly used transformation techniques. Particularly, we highlight two important transformation techniques, fusion and tupling, which in combination can be surprisingly good for deriving very efficient algorithms.
- Our derivation is more *systematic*, minimizing the use of complex laws with deeper insights, such as Horner's rule in [Bir89], which tend to make derivation harder to carry out. Instead, our approach relies on a set of smaller laws which are motivated by the need to perform fusion, i.e., to make fusion transformation successful. So most of our laws are *distributive* in nature.
- Our derivation is also *powerful*. To the best of our knowledge, we demonstrate the first full and systematic derivation for the maximum segment product problem, which makes unnecessary the "suitable cunning" in the previous derivation [Bir89].

For 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). A comparison is then made with the classical derivation via Horner's rule (Sec 5), before an ++++++++++ advice on the use of *accumulation* technique (Sec 6). Lastly, a short conclusion is given (Sec 7).

## 2 Enhanced Fusion with Laws

Fusion method [Chi92, TM95] is potentially a very 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 to automate fusion are restricted to the use of equational definitions of functions to perform transformation. For example, the deforestation algorithm [Wad88] relies on only define, unfold and fold rules [BD77] in its transformation which can be carried out using equational definitions of subject programs. Unfortunately, this approach is inadequate since many programs also rely on laws (useful properties between functions, such as associativity and distributivity) to apply fusion successfully.

Consider a function *sizetree* to compute the number of leaves in a tree, by flattening the leaves of the tree into a list, and then finding its size.

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

To optimise this program, we could try to fuse the composition $length(flattree(t))$. However, this cannot be done using just the above equations via unfold/fold rules. In particular, we also require the following distributive law of *length* over $\mathbin{+\!\!+}$.

$$length(xr \mathbin{+\!\!+} xs) \quad = \quad length(xr)+length(xs) \tag{1}$$

Using this law, the fusion derivation of *sizetree* can be carried out, as outlined below.

$$
\begin{array}{lll}
sizetree(Leaf(a)) & = & \{ \text{ instantiate } t{=}Leaf(a) \text{ } \} \\
 & & length(flattree(Leaf(a))) \\
 & = & \{ \text{ unfold } flattree \text{ } \} \\
 & & length([a]) \\
 & = & \{ \text{ unfold } length \text{ } \} \\
 & & 1
\end{array}
$$

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

What was the rationale for using a distributive law during the above fusion of $length(flattree(t))$? Informally, the inner function *flattree* produces $\mathbin{+\!\!+}$ 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 $\mathbin{+\!\!+}$, to consume the $\mathbin{+\!\!+}$ calls from the inner *flattree* function for successful fusion. A more detailed description of how laws help fusion can be found in [Chi94].

The laws needed by such enhanced fusion technique must either be supplied by programmers with their programs, or be derived via advanced synthesis techniques, such as [Smi89, CT97]. Even if the laws are supplied by users, they should still be verified - perhaps by a theorem-prover. There is some potential 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 and checked by users.

# 3 A Modular Derivation Strategy

We propose a modular derivation strategy based on two key transformation techniques, namely fusion and tupling, which are applied in sequence. To illustrate this strategy, consider the MSS problem:

$$mss(xs) \quad = max(map(sum,segs(xs)))$$

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

The 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 generated (by *segs*, *map* and *sum*).

Stack space is usually pre-allocated and cheap to recover. It is principally related to the depth of recursive calls (ignoring the effect of tail-call optimisation). The space occupied by input/output of the main function is fixed, and not changed by program transformation. 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*, the variable space cost are due to *segs* generating $O(n^2)$ sub-lists of $O(n)$ length each, while *map* yields another intermediate list of size $n^2$. These intermediate data structures result in a variable space complexity of $O(n^3)$.

Our strategy aims to derive efficient algorithms via two key techniques : *fusion*, followed by *tupling*. The effect of these two transformations are illustrated in Figure 1 for the MSS problem.

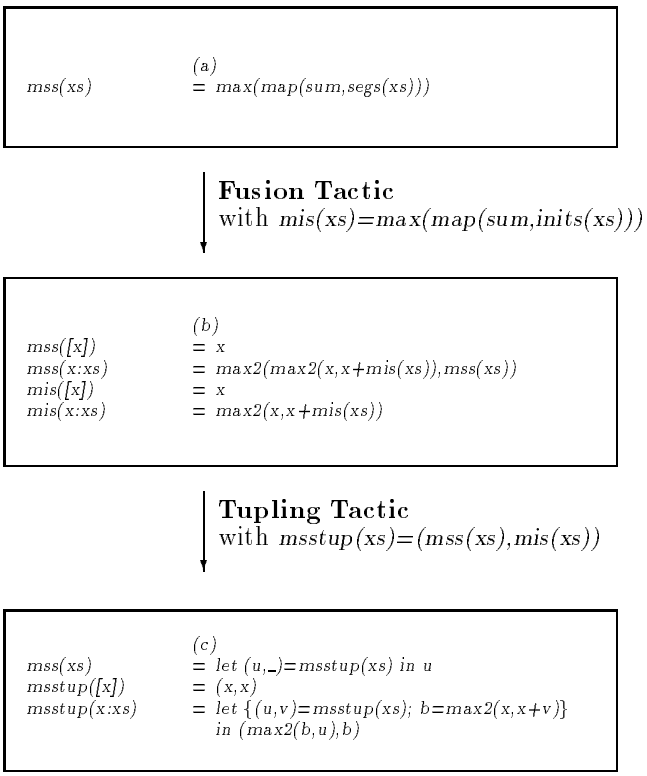


Figure 1: Modular Derivation Strategy via Fusion & Tupling

Fusion transformation is capable of eliminating all intermediate data structures for this example. Apart from the composition in the original definition of *mss*, we encountered another composition which was defined as the following new definition:

$$mis(xs) \quad = max(map(sum,inits(xs)))$$

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* function does not generate any intermediate data structures. Hence, it 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 which would introduce the following tuple definition:

$$msstup(xs) \quad = (mss(xs),mis(xs))$$

Subsequent transformation yields a new recursive tupled definition shown in Figure 1(c). Without any redundant calls, the new *msstup* definition has a much improved time-complexity of $O(n)$. In the next two sections, we present the actual derivations for obtaining these optimised programs.

### 3.1 Fusion to Remove Intermediate Data Structures

The enhanced fusion technique 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 additional *distributive* laws.

$$map(f,xr{+\!\!+}xs) = map(f,xr){+\!\!+}map(f,xs) \quad (2)$$
$$max(xr{+\!\!+}xs) = max2(max(xr),max(xs)) \quad (3)$$
$$map(f,map(g,xs)) = map(f \circ g,xs) \text{ where } (f \circ g)(x) = f(g(x)) \quad (4)$$
$$max(map((x+),xs)) = x+max(xs) \quad (5)$$

The first two laws are distributive laws of *map* and *max* over the ++ operator, while law (4) 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 (4), as follows:

$$max(map((x+) \circ g,xs)) = x+max(map(g,xs)) \quad (6)$$

Fusion/deforestation method makes use of normal-order symbolic evaluation/unfolding [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 and transformation can be achieved, as follows:

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

For the recursive case instantiation, the *segs* function actually produces ++ calls which must be consumed by *map* through its distributive law.

```
mss(x:xs)     =   { instantiate xs=x:xs }
                  max(map(sum,segs(x:xs)))
              =   { unfold segs }
                  max(map(sum,inits(x:xs)++segs(xs)))
              =   { apply law (2) : 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 (3) : 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. In addition, *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 function }
                  max2(mis(x:xs),mss(xs))
```

The new composition encountered is captured by the following definition.

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

We can again 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 can 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 from the composed expressions. This results in an improved time complexity of $O(n^2)$, and a much improved variable space complexity of $O(1)$. The final program, after an unfolding of *mis(x:xs)*, is shown below.

```
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))
```

### 3.2 Tupling to Eliminate Redundant Calls

After fusion, the transformed program may still contain redundant function calls. This inefficiency can be overcome by the tupling method [Chi93, HITT97]. The primary mechanism used in tupling is to gather calls with identical arguments together. In the case of *mss*, we can find two calls with identical arguments in its recursive equation. Tupling would gather these two calls into a tuple definition, as follows.

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

This can then be further transformed by using instantiations to facilitate the unfolding of one (or more) calls in the tuple. The base case instantiation and transformation can proceed, as follows:

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

The recursive case instantiation and transformation can be carried out, as 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 }
                  let (u,v)=msstup(xs) in (max2(max2(x,x+v),u),max2(x,x+v))
                 =    { share a common sub-expression  }
                  let {(u,v)=msstup(xs); b=max2(x,x+v)} in (max2(b,u),b)
```

Note how the use of a gathering step for calls with identical arguments, results in a tuple of two calls, which 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, shown below.

```
mss(xs)          = let (u,_)=msstup(xs) in u
msstup([x])      = (x,x)
msstup(x:xs)     = let {(u,v)=msstup(xs); b=max2(x,x+x)} in (max2(b,u),b)
```

## 4 Maximum Segment Product

Let us now turn our attention to a related but lesser 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 equations/laws related to *sum* and $+$. Specifically, the distributive law of *max* over *map* with $(x+)$ needs to be replaced by corresponding laws over $(x*)$. Interestingly, this property must be specified by a pair of laws, namely:

$$max(map((x*),xs)) = if\ x \geq 0\ then\ x*max(xs)\ else\ x*min(map(f,xs)) \quad (7)$$

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

Note the need for a dual function to *max*, namely *min*, to find the minimum value from a given list.

```
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. Thus, *min* and its auxiliary function *min2* are needed. More practical versions of the above pair of laws are obtained by combining them with law (4), as shown below.

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

$$min(map((x*) \circ f,xs)) = if\ x \geq 0\ then\ x*min(map(f,xs))\ else\ x*max(map(f,xs)) \quad (10)$$

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 }
                  max(map(prod,[x]))
                =   { unfold map }
                  max([prod([x])])
                =   { unfold max }
                  [prod([x])]
                =   { unfold 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 (2) : map(f,xr++xs)=map(f,xr)++map(f,xs) }
                  max(map(prod,inits(x:xs))++map(prod,segs(xs)))
                =   { apply law (3) : max(xr++xs)=max2(max(xr),max(xs)) }
                  max2(max(map(prod,inits(x:xs))),max(map(prod,segs(xs))))
                =   { fold with msp }
                  max2(max(map(prod,inits(x:xs))),mss(xs))
                =   { fold with a new defn for mip }
                  max2(mip(x:xs),mss(xs))
```

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

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

Its base case equation is derived as:

```
mip([x])        = x
```

The recursive case equation can also be derived, with the help of laws, as shown below.

$$
\begin{array}{lll}
mip(x{:}xs) & = & \{\text{ instantiate } xs{=}x{:}xs \ \} \\
 & & max(map(prod,inits(x{:}xs))) \\
 & = & \{\text{ unfold } inits \ \} \\
 & & max(map(prod,[x]{:}map((x{:}),inits(xs)))) \\
 & = & \{\text{ unfold } map \ \} \\
 & & max(prod([x]){:}map(prod,map((x{:}),inits(xs)))) \\
 & = & \{\text{ unfold } max \ \} \\
 & & max2(prod([x]),max(map(prod,map((x{:}),inits(xs)))) \\
 & = & \{\text{ unfold } prod \ \} \\
 & & max2(x,max(map(prod,map((x{:}),inits(xs)))) \\
 & = & \{\text{ apply law (4) : } map(f,map(g,xs)) = map(f \circ g,xs) \ \} \\
 & & max2(x,max(map(prod \circ (x{:}),inits(xs))) \\
 & = & \{\text{ unfold } prod \ \} \\
 & & max2(x,max(map((x*) \circ prod,inits(xs))) \\
 & = & \{\text{ apply law (9) of } max \text{ over } (x*) \ \} \\
 & & max2(x,if\ x{\geq}0\ then\ x{*}max(map(prod,inits(xs)))\ else\ x{*}min(map(prod,inits(xs)))) \\
 & = & \{\text{ fold } mip \ \} \\
 & & max2(x,if\ x{\geq}0\ then\ x{*}mip(xs)\ else\ x{*}min(map(prod,inits(xs)))) \\
 & = & \{\text{ introduce new } mipm \text{ definition } \} \\
 & & max2(x,if\ x{\geq}0\ then\ x{*}mip(xs)\ else\ x{*}mipm(xs)) \\
 & = & \{\text{ apply law (11) to float } if \text{ outwards } \} \\
 & & if\ x{\geq}0\ then\ max2(x,x{*}mip(xs))\ else\ max2(x,x{*}mipm(xs))
\end{array}
$$

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.)

$$
E[if\ e_1\ then\ e_2\ else\ e_3] \quad = \quad if\ e_1\ then\ E[e_2]\ else\ E[e_3] \tag{11}
$$

Another composition $x{*}min(map(prod,inits(xs)))$ was encountered. This was defined to be:

$$
mipm(xs) \quad = min(map(prod,inits(xs)))
$$

Its fusion derivation for *mipm* is very similar to *mip*. The base case instantiation simplifies to:

$$
mipm([x]) \quad = x
$$

The recursive case instantiation and transformation is outlined below.

$$
\begin{array}{lll}
mipm(x{:}xs) & = & \{\text{ instantiate } xs{=}x{:}xs \ \} \\
 & & min(map(prod,inits(x{:}xs))) \\
 & = & \{\text{ unfold } inits \ \} \\
 & & min(map(prod,[x]{:}map((x{:}),inits(xs)))) \\
 & = & \{\text{ unfold } map \ \} \\
 & & min(prod([x]){:}map(prod,map((x{:}),inits(xs)))) \\
 & = & \{\text{ unfold } min \ \} \\
 & & min2(prod([x]),min(map(prod,map((x{:}),inits(xs)))) \\
 & = & \{\text{ unfold } prod \ \} \\
 & & min2(x,min(map(prod,map((x{:}),inits(xs)))) \\
 & = & \{\text{ apply law (4) : } map(f,map(g,xs)) = map(f \circ g,xs) \ \} \\
 & & min2(x,min(map(prod \circ (x{:}),inits(xs))) \\
 & = & \{\text{ unfold } prod \ \} \\
 & & min2(x,min(map((x*) \circ prod,inits(xs))) \\
 & = & \{\text{ apply law (10) of } min \text{ over } (x*) \ \} \\
 & & min2(x,if\ x{\geq}0\ then\ x{*}min(map(prod,inits(xs)))\ else\ x{*}max(map(prod,inits(xs)))) \\
 & = & \{\text{ fold with } mipm \ \&\ mip \ \} \\
 & & min2(x,if\ x{\geq}0\ then\ x{*}mipm(xs)\ else\ x{*}mip(xs)) \\
 & = & \{\text{ apply law (11) to float } if \text{ outwards } \} \\
 & & if\ x{\geq}0\ then\ min2(x,x{*}mipm(xs))\ else\ min2(x,x{*}mip(xs))
\end{array}
$$

The completely fused program for *msp*, after unfolding *mip(x:xs)* in the RHS of *msp* and floating its inner conditional, is:

$$
msp([x]) \quad = x
$$

```
msp(x:xs)    = if x≥0 then max2(max2(x,mip(xs)),msp(xs)) else max2(max2(x,mipm(xs)),msp(xs))
mip([x])     = x
mip(x:xs)    = if x≥0 then max2(x,x*mip(xs)) else max2(x,x*mipm(xs))
mipm([x])    = x
mipm(x:xs)   = if x≥0 then min2(x,x*mipm(xs)) else min2(x,x*mip(xs))
```

From the current program, tupling analysis of [Chi93, HITT97] would reveal that there are redundant calls to *mip* and *mipm*. They can be eliminated by introducing the following tuple definition.

```
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/shares common if over tuple structure }
               let (u,v,w)=msstup(xs) in
               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 somewhat more complex than that for *msstup*, even though their initial specifications are very similar. Fortunately, for us, we used essentially the same transformation techniques, namely fusion followed by tupling, to systematically obtain both space and time efficient algorithms. Specifically, fusion helps to eliminate unnecessary intermediate data structures (improving on space), while tupling helps to eliminate redundant calls (improving on time). As a result, the optimised algorithm has a variable space complexity of $O(1)$, and a time complexity of $O(n)$.

Due to our use of two modular transformation techniques, we need only provide two extra (straightforward) laws to allow distribution of *max* (and *min*) over products. Such laws are sufficient for us 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 for the MSP problem since the corresponding Horner's rule have to be invented over tupled functions. In our case, this is naturally taken care of by the tupling method. A more detailed comparison is undertaken in the next section.

# 5 Classical Derivation via Horner's rule

The MSS (and to a lesser extent the MSP) problem is not new. Formal derivation to obtain efficient linear-time algorithm was first developed by Bird [Bir88], but the problem originated from Bentley [Ben86].

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. 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 $\otimes$ are 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:

$$(\oplus\ /)\ map(\otimes\ /,\ tails([a_1, \ldots, a_n])) \quad = \quad \circledast \not\triangleright_{1_\otimes} [a_1, \ldots, a_n] \tag{12}$$

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

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

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

$$\begin{array}{ll}
mss(xs) & = (max2\ /)(map((+\ /),segs'(xs))) \\
& = \quad \{\ \text{unfold } segs'(xs){=}flatten(map(tails,inits(xs)))\ \} \\
& \quad (max2\ /)(map((+\ /),flatten(map(tails,inits(xs))))) \\
& = \quad \{\ \text{apply law : } map(f,flatten(xss)){=}flatten(map(\backslash xs.map(f,xs),xss))\ \} \\
& \quad (max2\ /)(flatten(map(\backslash z.map((+\ /),z),map(tails,inits(xs))))) \\
& = \quad \{\ \text{apply law : } max(flatten(xss)){=}max(map(max,xss))\ \} \\
& \quad (max2\ /)(map((max2\ /),map(\backslash z.map((+\ /),z),map(tails,inits(xs))))) \\
& = \quad \{\ \text{apply law : } map(f,map(g,xs)) = map(f \circ g,xs) \text{ twice}\ \} \\
& \quad (max2\ /)(map(\backslash y.\ (max2\ /)(map(\backslash z.map((+\ /),z),tails(y))),\ inits(xs))) \\
& = \quad \{\ \text{apply Horner's rule : } (\oplus\ /)map(\otimes\ /,tails\ xs) = \circledast \not\triangleright_{1_\otimes}\ xs\ \} \\
& \quad (max2\ /)(map(\circledast \not\triangleright_0,\ inits(xs)))\ \text{where } a \circledast b = max2(a{+}b,0) \\
& = \quad \{\ \text{apply scan law : } map(\odot \not\triangleright_0,inits(xs)){=}\odot \not\triangleright\!\!\!\triangleright_0\ xs\ \} \\
& \quad (max2\ /)(\circledast \not\triangleright\!\!\!\triangleright_0\ xs)
\end{array}$$

Note that we used a non-recursive definition of *segs'* which returns segments in a different order (from *segs* given in Sec. 1). Also, a number of other functions are used, including:

$$\begin{array}{ll}
\odot \not\triangleright\!\!\!\triangleright_e\ Nil & = [e] \\
\odot \not\triangleright\!\!\!\triangleright_e\ (xs{+\!\!+}[y]) & = (\odot \not\triangleright\!\!\!\triangleright_e\ xs) {+\!\!+} [last(\odot \not\triangleright\!\!\!\triangleright_e\ xs) \odot y] \\
last(xs{+\!\!+}[y]) & = y \\
flatten(Nil) & = Nil \\
flatten(xs{:}xss) & = xs{+\!\!+}flatten(xss)
\end{array}$$

The final algorithm obtained for *mss* has a linear time complexity, and also a linear (variable) space complexity due to an intermediate list from $(\circledast \not\triangleright\!\!\!\triangleright_0\ xs)$. This slightly worse space behaviour may be improved by fusion transformation. Although this classical derivation, based on Horner's rule, looks more concise that our proposed derivation, it requires larger derivation steps (i.e. more complex laws/theorems).

Often, suitable algebraic properties are also required as side-conditions to such laws/theorems. They may be difficult to check, and especially hard to ensure. 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:

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

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 the $\oplus$ and $\otimes$ operators for its Horner's rule, he suggested that the following tupled functions be used instead.

$$\begin{aligned} (a_1, b_1) \oplus (a_2, b_2) &= (min2(a_1, a_2), max2(b_1, b_2)) \\ (a,b) \otimes c &= \text{if } c \geq 0 \text{ then } (a*c,b*c) \text{ else } (b*c,a*c) \end{aligned}$$

Generalised in this way, it is possible to prove that $\otimes$ distributes backwards through $\oplus$, as follows:

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

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 can help break-up difficult theorems/insights required through the use of simpler transformation techniques, where possible.

# 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:

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

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

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

Both *seg'* and *seg* yields the same set of segments, except that these segments are returned in a different order. Unfortunately, this innocous change seems to have an effect in the kind of derivation which can be performed.

For example, if *segs* were used by the classical derivation, we will need a different type of Horner's and scan rules, which are oriented for right-to-left reduction, as opposed to left-to-right ones. {♠ Is this correct? } Correspondingly, if *segs'* were used by our modular approach to derivation, we may require equations based on right-to-left evaluation, 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? And 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:

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

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

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

Our second question was *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 unfriendly to tupling! For example, consider the fusion of *segs'* below.

```
segs'(x:xs)     =   { instantiate xs=x:xs }
                  flatten(map(tails,inits(x:xs)))
                =   { unfold inits }
                  flatten(map(tails,[x]:map((x:),inits(xs))))
                =   { unfold map }
                  flatten(tails([x]):map(tails,map((x:),inits(xs))))
                =   { unfold flatten }
                  tails([x])++flatten(map(tails,map((x:),inits(xs))))
                =   { apply law (4) : map(f,map(g,xs)) = map(f ∘ g,xs) }
                  tails([x])++flatten(map(tails ∘ (x:),inits(xs)))
```

After several steps, we are still unable to fold as we encountered a slightly enlarged expression of the form *flatten(map(tails ∘ (x:),inits(xs))*. As reported elsewhere [Bir84] and [?]{♠ Please pass your reference in NGC } ), this calls for the use of an *accumulation tactic* to overcome the problem of meeting ever larger expressions during transformation. Specifically, we need to define:

```
asegs'(w,xs)    = flatten(map(tails ∘ (w++),inits(xs)))
```

With a new generalised parameter *w*, we can now re-apply fusion to obtain:

```
asegs'(w,[x])   = tails(w++[x])
asegs'(w,x:xs)  = tails(w++[x])++asegs'(w++[x],xs)
```

In general, this accumulation tactic is bad for two reasons. Firstly, the presence of an accumulating (list) parameter has indicated that fusion has not been totally successful (at least it can be said to have failed for the accumulating parameter). Secondly, the resulting function (with an accumulating parameter) is actually unsuitable for tupling since its redundant calls may now 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).

Hence, we should *avoid (or delay)* the application of accumulating tactic, where possible. One way to avoid the accumulation tactic is to turn to *snoc*-based equations, whenever the use of accumulation is inevitable. In the case of *seg'*, the corresponding fusion transformation using *snoc*-based equations can proceed (without accumulation), as follows:

```
segs'(xs++[y])  =   { instantiate xs=xs++[y] }
                  flatten(map(tails,inits(xs++[y])))
                =   { unfold inits }
                  flatten(map(tails,inits(xs)++[xs++[y]]))
                =   { apply law (2) : map(f,xr++xs) = map(f,xr)++map(f,xs) }
                  flatten(map(tails,inits(xs))++map(tails,[xs++[y]]))
                =   { apply law : flatten(xr++xs) = flatten(xr)++flatten(xs) }
                  flatten(map(tails,inits(xs)))++flatten(map(tails,[xs++[y]]))
                =   { fold with segs' }
                  segs'(xs)++flatten(map(tails,[xs++[y]]))
                =   { unfold map }
                  segs'(xs)++flatten([tails(xs++[y])])
                =   { unfold flatten }
                  segs'(xs)++ [tails(xs++[y])]
```

With this version of *segs'*, the main *mss* function can now be optimised by fusion to yield:

```
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)
```

The redundant calls in the above fused program can now be eliminated via tupling without being hindered by the presence of accumulating parameters. Our advice is therefore : *to avoid/delay the application of accumulation tactic, where possible.* As suggested here, one way to achieve this is to rely on *snoc*-based equations, should the *cons*-based counterparts be found to be inadequate for fusion.

# 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. The main advantage of our proposal is a clear division of program derivation into two phases, for the eliminations of intermediate data and redundant calls, respectively. While the steps taken may be longer than the traditional BMF approach, the opportunities for mechanisation are much higher since we rely on less insightful laws/theorems to perform these transformations. In particular, simple laws are only used 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. Other modular transformation techniques are likely to be helpful too. Finding a good collection of these techniques could be instrumental towards an improved methodology for developing useful 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.

[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.

[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 WN. Chin. Parallelization in calculational forms. In *25th Annual ACM Symposium on Principles of Programming Languages*, San Diego, California, January 1998. ACM Press (to appear).

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

[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.

[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.