An Introduction to the Bird-Meertens Formalism

Jeremy Gibbons
Department of Computer Science
University of Auckland
Private Bag 92019, Auckland, New Zealand.
Email: jeremy@cs.auckland.ac.nz

Abstract. The Bird-Meertens Formalism, or 'Squiggol', is a calculus for the construction of programs from their specifications by a process of equational reasoning. Developments are directed by considerations of *data*, as opposed to *program*, structure.

This paper presents a brief introduction to the philosophy and notation of the calculus, in the guise of the (well-known) derivation of a linear-time solution to the 'maximum segment sum' problem.

Keywords. Bird-Meertens Formalism, Squiggol, program transformation, program derivation, functional programming.

1 Introduction

In a sentence, the Bird-Meertens Formalism (or 'Squiggol') might be described as a *calculus* for the *construction of programs* by a process of *equational reasoning* from their *specifications*. A lot of meaning is packed into this sentence; let us unpack some of these terms.

By 'calculus' we mean a collection of concepts and notations, together with a body of theorems stating relationships between them. The analogy is with, for example, the differential calculus in mathematics. A calculus is concerned with a particular class of problems in a field, and it aims to embody the 'tricks of the trade' used by practitioners in that field.

By 'construction of programs from their specifications' we mean, essentially, the 'transformational programming' approach to program construction. This is described by Darlington (1981) as follows:

> Using the transformational approach to programming, a programmer does not attempt to produce directly a program that is correct, understandable and efficient, rather he initially concentrates on producing a program which is as clear and understandable as possible ignoring any question of

 This draft dated 1st November 1994. Supported by University of Auckland Research Committee grant number A18/XXXXX/62090/3414013.

> efficiency. Having satisfied himself that he has a correct program he successively transforms it to more and more efficient versions using methods guaranteed not to change the meaning of the program.

Indeed, the notion of 'program' can be relaxed to include also non-executable constructs. In this case, the programmer initially concentrates on producing a 'program' ignoring also any question of implementability (that is, a specification). This must then be transformed into more and more implementable, but still equivalent, versions.

By 'equational reasoning' we mean 'a linear, equational proof that the original specification is extensionally equal to the resulting more efficient version' (Malcolm, 1990). This is the distinguishing characteristic of the Bird-Meertens Formalism ('BMF') over other transformational approaches.

Unpacking further, by 'linear, equational proof' we mean a calculation of the form:

$$
\begin{array}{ll}
 & \text{original specification} \\
= & \quad \{\ \text{justification for first step}\ \} \\
 & \text{intermediate version} \\
= & \quad \{\ \text{justification for next step}\ \} \\
 & \qquad \vdots \\
= & \quad \{\ \text{justification for last step}\ \} \\
 & \text{more efficient version}
\end{array}
$$

The name of the BMF game is to perform as much of the development as is practical in this equational style. This is for expository reasons. It clarifies the distinction between the routine parts of the development (those that consist of straightforward calculation, using 'current technology') and the creative parts (those that we do not yet know how to calculate, but which require more inventive steps). In communicating the *essence* of a development, the calculations can be elided to just their first and last lines:

$$
\begin{array}{ll}
 & \text{original specification} \\
= & \quad \{\ \text{routine calculation}\ \} \\
 & \text{more efficient version}
\end{array}
$$

Put another way, we are interested in extending what can be calculated precisely *because* we are not interested in the calculations themselves. Developing techniques that allow more of a derivation to be calculated allows attention to be focussed on the creative parts, which cannot be calculated.

Such an equational calculational style naturally entails a powerful collection of theorems to provide the steps in the calculation. Moreover, these theorems should be expressed as equalities with just a few simple applicability conditions, so that the flow of the calculational development is not interrupted. Inductive proofs are eschewed as far as possible. The body of theorems used in the BMF arises from a theory of data structures. Ideas borrowed from universal algebra and category theory yield many powerful theorems about common patterns of computation over a data structure, given only the data type definition.

The BMF is applicable to many programming styles and application areas.

- Early work was based on a distinctly sequential intuition, but Skillicorn (1990) and others have shown that the BMF makes just as good a parallel programming language as a sequential one. Indeed, it makes perhaps the best parallel programming language currently available.
- The programming style may be 'functional' (in the sense that it uses higher-order operators to encapsulate common patterns of computation), but the result of a development can just as well be implemented in an imperative language. (The main disadvantage of an imperative programming style over a functional one is the relatively impoverished notion of 'type'. Implementing a functional solution in an imperative language will typically involve a non-trivial data refinement step, from the rich supply of abstract types available in the functional language to the poor supply of more concrete ones in the imperative language.)
- Current work at Oxford (Bird and de Moor, 1993a, 1993b, 1994) and Eindhoven (Backhouse et al., 1991, 1992a, 1992b; Backhouse and Hoogendijk, 1993) is focussed on generalizing the BMF to relations, as opposed to total functions. This allows non-determinism, partial functions and inverses to be expressed naturally, and refinement, rather than equality, to be used as the relationship between steps of a calculation.
- Finally, very similar techniques apply to the development of hardware circuits; the BMF shares much with the *Ruby* (Jones and Sheeran, 1990, 1992) and *Rebecca* (Luk et al., 1994) approaches to hardware design.

Backhouse (1989) describes the impact of the BMF as follows:

> From a calculational perspective [Bird and Meertens'] work represents for me the most major advance that I have encountered in my career as a computing scientist. From the traditional, foundational, perspective, however, their work presents no surprises; the basic theorems and the methods they use are well-known and have been so, in most cases, for several decades. The advance that they have made is to show how concise notation designed around functional algebraic properties can substantially increase the effectiveness of the calculational method.

The purpose of this paper is to give a brief introduction to the philosophy and notation of the BMF. This is done by example, the example in question being the (well-known) derivation of a linear-time solution to the 'maximum segment sum' problem.

The rest of this paper is structured as follows. In Section 2, we describe the maximum segment sum problem informally. In Section 3, we introduce the notation we need, and in Section 4, we use it to specify the problem formally. In Section 5, we explain *Horner's Rule*, which forms an important part of the solution to the problem. Finally, in Section 6, we calculate this solution.

2 The maximum segment sum problem

Given a list of integers, the *maximum segment sum* ('MSS') problem is to compute the maximum of the sums of any of the non-empty contiguous *segments* of that list. For example, when the list consists of the integers

$$31 \;\; -41 \;\; 59 \;\; 26 \;\; -53 \;\; 58 \;\; 97 \;\; -93 \;\; -23 \;\; 84$$

the segment with the greatest sum, 187, is the one that omits the first two and the last three elements of the list.

The specification 'the maximum of the sums of any of the non-empty contiguous segments' is executable, and takes a number of steps cubic in the length of the list. There is a linear algorithm, arising from *Horner's Rule* for simplifying polynomial evaluation. We introduce just enough notation and laws to express the problem and to derive this algorithm.

This development is due to Bird (1988, 1989), but the problem itself is due to Bentley (1986).

3 Notation

In this section, we introduce just enough notation in order to state and solve the MSS problem.

3.1 Types

We write '$\mathsf{a} : \mathsf{A}$' for the type judgement 'element a has type A'.

3.2 Functions

The identity function is written 'id'.

Function application is written with an infix '.', so that $\mathsf{id.a} = \mathsf{a}$. Function application is right-associative, so that $\mathsf{f.g.a}$ parses as $\mathsf{f.(g.a)}$, and is the tightest-binding infix operator.

We write '$\mathsf{A} \to \mathsf{B}$' for the type of functions from A to B; thus, if $\mathsf{a} : \mathsf{A}$ and $\mathsf{f} : \mathsf{A} \to \mathsf{B}$ then $\mathsf{f.a} : \mathsf{B}$.

Function composition is written with an infix '$\circ$', so that $(\mathsf{f} \circ \mathsf{g}).\mathsf{a} = \mathsf{f.g.a}$. It is the weakest-binding infix operator.

3.3 Binary operators

The BMF makes great use of infix binary operators, where more traditional notations would use prefix alphabetic function names. As Bird (1984) says:

> Not only can such operators enhance the succinctness and, used sparingly, the readability of expressions, they also allow many transformations to be expressed as algebraic laws about their distributive and other properties.

Another notational advantage provided by infix binary operators is known as *sectioning*. A binary operator may be given one of its arguments, yielding a function of its other argument. For example, suppose that $\oplus : \mathsf{A} \times \mathsf{B} \to \mathsf{C}$ (that is, that binary operator $\oplus$ takes a pair of arguments, one of type A and the other of type B, and returns a result of type C), and $\mathsf{a} : \mathsf{A}$ and $\mathsf{b} : \mathsf{B}$. Then the 'left section' $(\mathsf{a}\oplus)$ has type $\mathsf{B} \to \mathsf{C}$ and, when applied to b, returns $\mathsf{a} \oplus \mathsf{b}$. Similarly, the 'right section' $(\oplus\mathsf{b})$ has type $\mathsf{A} \to \mathsf{C}$ and, when applied to a, also returns $\mathsf{a} \oplus \mathsf{b}$. Thus, the two sections satisfy the property

$$(\mathsf{a}\oplus).\mathsf{b} \;=\; \mathsf{a} \oplus \mathsf{b} \;=\; (\oplus\mathsf{b}).\mathsf{a}$$

3.4 Lists

If $\mathsf{n} \geq 1$ and $\mathsf{a}_1, \ldots, \mathsf{a}_\mathsf{n}$ are all elements of type A, then $[\mathsf{a}_1, \ldots, \mathsf{a}_\mathsf{n}]$ has type $\mathsf{list.A}$ (in other words, it is a list of elements, each of which has type A). We write '$\square$' for the function that takes a to $[\mathsf{a}]$, and '$+\!\!+$' for the associative binary operator that concatenates two lists.

3.5 Map

The most important operation related to lists is the postfix operator '$*$', pronounced 'map'. For a function $\mathsf{f} : \mathsf{A} \to \mathsf{B}$, the function $\mathsf{f}*$ has type $\mathsf{list.A} \to \mathsf{list.B}$. Informally, $\mathsf{f}*$ applies f to every element of its argument. Thus:

$$\mathsf{f}*.[\mathsf{a}_1, \ldots, \mathsf{a}_\mathsf{n}] \;=\; [\mathsf{f}.\mathsf{a}_1, \ldots, \mathsf{f}.\mathsf{a}_\mathsf{n}]$$

Map is completely determined by the equations

$$\begin{aligned} \mathsf{f}*.\square.\mathsf{a} \;&=\; \square.\mathsf{f}.\mathsf{a} \\ \mathsf{f}*.(\mathsf{x} +\!\!+ \mathsf{y}) \;&=\; \mathsf{f}*.\mathsf{x} \;+\!\!+\; \mathsf{f}*.\mathsf{y} \end{aligned}$$

It distributes backwards through function composition:

$$(\mathsf{f} \circ \mathsf{g})* \;=\; \mathsf{f}* \circ \mathsf{g}*$$

3.6 Catamorphisms

It can be shown that, for a given function $\mathsf{f} : \mathsf{A} \to \mathsf{B}$ and associative binary operator $\oplus$ of type $\mathsf{B} \times \mathsf{B} \to \mathsf{B}$, there is a unique solution $\mathsf{h} : \mathsf{list.A} \to \mathsf{B}$ of the equations

$$\begin{aligned} \mathsf{h}.\square.\mathsf{a} \;&=\; \mathsf{f}.\mathsf{a} \\ \mathsf{h}.(\mathsf{x} +\!\!+ \mathsf{y}) \;&=\; \mathsf{h}.\mathsf{x} \;\oplus\; \mathsf{h}.\mathsf{y} \end{aligned}$$

This fact is known as the *unique extension property*. We write this unique solution '$(\!|\mathsf{f}, \oplus|\!)$' (it is completely determined by the f and $\oplus$), and call it a *list catamorphism*. Stated another way, the list catamorphism $(\!|\mathsf{f}, \oplus|\!)$ satisfies:

$$\begin{aligned} (\!|\mathsf{f}, \oplus|\!).\square.\mathsf{a} &= \mathsf{f.a} \\ (\!|\mathsf{f}, \oplus|\!).(\mathsf{x} +\!\!\!+ \mathsf{y}) &= (\!|\mathsf{f}, \oplus|\!).\mathsf{x} \;\oplus\; (\!|\mathsf{f}, \oplus|\!).\mathsf{y} \end{aligned}$$

(and in fact is the only solution to these equations).

The list catamorphism $(\!|\mathsf{f}, \oplus|\!)$ can be thought of as a 'relabelling', replacing every occurrence of $\square$ by f, and every occurrence of $+\!\!\!+$ by $\oplus$. For example:

$$(\!|\mathsf{f}, \oplus|\!).(\square.\mathsf{a} +\!\!\!+ \square.\mathsf{b} +\!\!\!+ \square.\mathsf{c}) \;=\; \mathsf{f.a} \oplus \mathsf{f.b} \oplus \mathsf{f.c}$$

Many useful functions are list catamorphisms. Some examples are:

$$\begin{aligned} \mathsf{id} &= (\!|\square, +\!\!\!+|\!) \\ \mathsf{f}* &= (\!|\square \circ \mathsf{f}, +\!\!\!+|\!) \\ \mathsf{last} &= (\!|\mathsf{id}, \gg|\!) \qquad \text{where} \quad \mathsf{a} \gg \mathsf{b} = \mathsf{b} \\ \mathsf{sum} &= (\!|\mathsf{id}, +|\!) \\ \mathsf{prod} &= (\!|\mathsf{id}, \times|\!) \\ \mathsf{max} &= (\!|\mathsf{id}, \uparrow|\!) \\ \mathsf{flatten} &= (\!|\mathsf{id}, +\!\!\!+|\!) \end{aligned}$$

Here, last returns the last element of a list, the binary operator $\uparrow$ returns the greater of its two (numeric) arguments, and $\mathsf{flatten}$ concatenates a list of lists into a single long list. Note that, in each of these examples, the second component of the list catamorphism is associative.

3.7 Promotion

The most important property of list catamorphisms is the *Promotion Theorem*. If $\oplus$ and $\otimes$ are associative, and

$$\mathsf{h}.(\mathsf{x} \oplus \mathsf{y}) \;=\; \mathsf{h.x} \;\otimes\; \mathsf{h.y}$$

for all x and y (we say 'h is $\oplus$ to $\otimes$ promotable'), then

$$\mathsf{h} \circ (\!|\mathsf{f}, \oplus|\!) \;=\; (\!|\mathsf{h} \circ \mathsf{f}, \otimes|\!)$$

The proof of the Promotion Theorem is by the unique extension property. We have

$$\begin{aligned} & (\mathsf{h} \circ (\!|\mathsf{f}, \oplus|\!)).\square.\mathsf{a} \\ = & \quad \{ \text{ composition } \} \\ & \mathsf{h}.(\!|\mathsf{f}, \oplus|\!).\square.\mathsf{a} \\ = & \quad \{ \text{ catamorphisms } \} \\ & \mathsf{h.f.a} \\ = & \quad \{ \text{ composition } \} \\ & (\mathsf{h} \circ \mathsf{f}).\mathsf{a} \end{aligned}$$

and

$$
\begin{array}{ll}
& (\mathsf{h} \circ (\!|\mathsf{f}, \oplus|\!)).(\mathsf{x} +\!\!\!+ \mathsf{y}) \\
= & \quad \{ \text{ composition } \} \\
& \mathsf{h}.(\!|\mathsf{f}, \oplus|\!).(\mathsf{x} +\!\!\!+ \mathsf{y}) \\
= & \quad \{ \text{ catamorphisms } \} \\
& \mathsf{h}.((\!|\mathsf{f}, \oplus|\!).\mathsf{x} \oplus (\!|\mathsf{f}, \oplus|\!).\mathsf{y}) \\
= & \quad \{ \text{ assumption of promotability } \} \\
& \mathsf{h}.(\!|\mathsf{f}, \oplus|\!).\mathsf{x} \ \otimes \ \mathsf{h}.(\!|\mathsf{f}, \oplus|\!).\mathsf{y} \\
= & \quad \{ \text{ composition, twice } \} \\
& (\mathsf{h} \circ (\!|\mathsf{f}, \oplus|\!)).\mathsf{x} \ \otimes \ (\mathsf{h} \circ (\!|\mathsf{f}, \oplus|\!)).\mathsf{y}
\end{array}
$$

and so, by the unique extension property, the result holds.

As a corollary, we have the *catamorphism promotion* law:

$$(\!|\mathsf{f}, \oplus|\!) \circ \mathsf{flatten} \;=\; (\!|\mathsf{id}, \oplus|\!) \circ (\!|\mathsf{f}, \oplus|\!)*$$

since, by the promotion theorem, both sides are equal to $(\!|(\!|\mathsf{f}, \oplus|\!), \oplus|\!)$:

$$
\begin{array}{ll}
& (\!|\mathsf{f}, \oplus|\!) \circ \mathsf{flatten} \\
= & \quad \{ \ \mathsf{flatten} \ \} \\
& (\!|\mathsf{f}, \oplus|\!) \circ (\!|\mathsf{id}, +\!\!\!+|\!) \\
= & \quad \{ \ (\!|\mathsf{f}, \oplus|\!) \text{ is } +\!\!\!+ \text{ to } \oplus \text{ promotable } \} \\
& (\!|(\!|\mathsf{f}, \oplus|\!), \oplus|\!)
\end{array}
$$

and

$$
\begin{array}{ll}
& (\!|\mathsf{id}, \oplus|\!) \circ (\!|\mathsf{f}, \oplus|\!)* \\
= & \quad \{ \text{ map } \} \\
& (\!|\mathsf{id}, \oplus|\!) \circ (\!|\square \circ (\!|\mathsf{f}, \oplus|\!), +\!\!\!+|\!) \\
= & \quad \{ \ (\!|\mathsf{id}, \oplus|\!) \text{ is } +\!\!\!+ \text{ to } \oplus \text{ promotable } \} \\
& (\!|(\!|\mathsf{id}, \oplus|\!) \circ \square \circ (\!|\mathsf{f}, \oplus|\!), \oplus|\!) \\
= & \quad \{ \text{ catamorphisms: } (\!|\mathsf{id}, \oplus|\!) \circ \square = \mathsf{id} \ \} \\
& (\!|(\!|\mathsf{f}, \oplus|\!), \oplus|\!)
\end{array}
$$

A special case of this is called *map promotion*:

$$\mathsf{f}* \circ \mathsf{flatten} \;=\; \mathsf{flatten} \circ \mathsf{f}**$$

3.8 Rightwards fold

Informally, the rightwards fold $(\mathsf{f}, \oplus)\not\to$ satisfies

$$(\mathsf{f}, \oplus)\not\to.[\mathsf{a}_1, \mathsf{a}_2, \ldots, \mathsf{a}_n] \;=\; ((\mathsf{f}.\mathsf{a}_1 \oplus \mathsf{a}_2) \oplus \cdots) \oplus \mathsf{a}_n$$

The operator $\oplus$ need not be associative, as it must for a list catamorphism. Indeed, the most general typing has $\mathsf{f} : \mathsf{A} \to \mathsf{B}$ and $\oplus : \mathsf{B} \times \mathsf{A} \to \mathsf{B}$, and so $\oplus$ cannot be associative. In this most general case, $(\mathsf{f}, \oplus)\not\to$ has type $\mathsf{list}.\mathsf{A} \to \mathsf{B}$.

The rightwards fold $(\mathsf{f}, \oplus)\not\to$ is completely determined by the equations

$$\begin{aligned} (\mathsf{f}, \oplus)\not\to.\square.\mathsf{a} &= \mathsf{f}.\mathsf{a} \\ (\mathsf{f}, \oplus)\not\to.(\mathsf{x} + \!\!\!+\, \square.\mathsf{a}) &= (\mathsf{f}, \oplus)\not\to.\mathsf{x} \;\oplus\; \mathsf{a} \end{aligned}$$

Every list catamorphism is a rightwards fold; this fact is known as *list catamorphism specialization*. In general, when $\oplus$ is associative, $(\!|\mathsf{f}, \oplus|\!) = (\mathsf{f}, \otimes)\not\to$ where $\mathsf{u} \otimes \mathsf{b} = \mathsf{u} \oplus \mathsf{f}.\mathsf{b}$. For example, $\mathsf{sum} = (\mathsf{id}, +)\not\to$. However, there are rightwards folds that are not list catamorphisms.

3.9 Inits and tails

The two functions inits and tails each take a list and return a list of lists. The first returns all initial segments of its argument, in order of increasing length; the second returns all tail segments, in order of decreasing length. Thus:

$$\begin{aligned} \mathsf{inits}.[\mathsf{a}_1, \mathsf{a}_2, \ldots, \mathsf{a}_n] &= [[\mathsf{a}_1], [\mathsf{a}_1, \mathsf{a}_2], \ldots, [\mathsf{a}_1, \mathsf{a}_2, \ldots, \mathsf{a}_n]] \\ \mathsf{tails}.[\mathsf{a}_1, \mathsf{a}_2, \ldots, \mathsf{a}_n] &= [[\mathsf{a}_1, \mathsf{a}_2, \ldots, \mathsf{a}_n], [\mathsf{a}_2, \ldots, \mathsf{a}_n], \ldots, [\mathsf{a}_n]] \end{aligned}$$

Both are rightwards folds:

$$\begin{aligned} \mathsf{inits} &= (\square \circ \square, \oplus)\not\to \qquad \text{where} \quad \mathsf{z} \oplus \mathsf{a} = \mathsf{z} + \!\!\!+\, \square.(\mathsf{last}.\mathsf{z} + \!\!\!+\, \square.\mathsf{a}) \\ \mathsf{tails} &= (\square \circ \square, \otimes)\not\to \qquad \text{where} \quad \mathsf{z} \otimes \mathsf{a} = (+\!\!\!+\square.\mathsf{a})*.\mathsf{z} + \!\!\!+\, \square.\square.\mathsf{a} \end{aligned}$$

(In fact, both are also list catamorphisms.)

3.10 Rightwards scan

The rightwards scan $(\mathsf{f}, \oplus)\not\!\!\not\to$ consists of applying the rightwards fold $(\mathsf{f}, \oplus)\not\to$ to every initial segment of a list:

$$(\mathsf{f}, \oplus)\not\!\!\not\to \;=\; (\mathsf{f}, \oplus)\not\to * \circ \mathsf{inits} \qquad \text{— (1)}$$

Thus:

$$\begin{aligned} (\mathsf{f}, \oplus)\not\!\!\not\to.[\mathsf{a}_1, \ldots, \mathsf{a}_n] \;=\; [&\mathsf{f}.\mathsf{a}_1, \\ &\mathsf{f}.\mathsf{a}_1 \oplus \mathsf{a}_2, \\ &\ldots \\ &((\mathsf{f}.\mathsf{a}_1 \oplus \mathsf{a}_2) \oplus \cdots) \oplus \mathsf{a}_n] \end{aligned}$$

It satisfies the equations

$$
\begin{aligned}
(\mathsf{f}, \oplus)\not\!/\!\!/.\Box.\mathsf{a} &= \Box.\mathsf{f}.\mathsf{a} \\
(\mathsf{f}, \oplus)\not\!/\!\!/.(\mathsf{x} \mathbin{+\!\!+} \Box.\mathsf{a}) &= (\mathsf{f}, \oplus)\not\!/\!\!/.\mathsf{x} \mathbin{+\!\!+} \Box.(\mathsf{f}, \oplus)\nrightarrow.(\mathsf{x} \mathbin{+\!\!+} \Box.\mathsf{a}) \\
&= (\mathsf{f}, \oplus)\not\!/\!\!/.\mathsf{x} \otimes \mathsf{a} \qquad \text{where} \quad \mathsf{z} \otimes \mathsf{a} = \mathsf{z} \mathbin{+\!\!+} \Box.(\mathsf{last}.\mathsf{z} \oplus \mathsf{a})
\end{aligned}
$$

and so

$$
(\mathsf{f}, \oplus)\not\!/\!\!/ \;=\; (\Box \circ \mathsf{f}, \otimes)\nrightarrow \qquad \text{where} \quad \mathsf{z} \otimes \mathsf{a} = \mathsf{z} \mathbin{+\!\!+} \Box.(\mathsf{last}.\mathsf{z} \oplus \mathsf{a}) \qquad \text{— (2)}
$$

The important fact about this last equation is that the initial characterization (1) of $(\mathsf{f}, \oplus)\not\!/\!\!/$, although executable, requires quadratically many applications of $\oplus$. Characterization (2), on the other hand, requires only linearly many applications. For example, the 'running totals' of a list of numbers is defined by $\mathsf{sum}* \circ \mathsf{inits}$, which involves quadratically many additions. Since sum is a rightwards fold, $(\mathsf{id}, +)\nrightarrow$, the running totals can be computed as a rightwards scan, $(\mathsf{id}, +)\not\!/\!\!/$, using only linearly many additions. This fact is one of the two important steps in our development of a solution for the MSS problem; the other step is Horner's Rule, as described below.

3.11 Segments

As a final piece of notation, the function segs takes a list and returns a list of lists, consisting of all non-empty contiguous segments of its argument. For example:

$$
\mathsf{segs}.[1, 2, 3] \;=\; [[1], [1, 2], [2], [1, 2, 3], [2, 3], [3]]
$$

Formally:

$$
\mathsf{segs} \;=\; \mathsf{flatten} \circ \mathsf{tails}* \circ \mathsf{inits}
$$

The ordering of the segments in the result is necessarily somewhat arbitrary; for our purposes, it will not matter what order segs returns.

4 Specifying the problem

We can now express the MSS problem formally:

$$
\mathsf{mss} \;=\; \mathsf{max} \circ \mathsf{sum}* \circ \mathsf{segs}
$$

The functions segs produces quadratically many segments, and finding the sum of each takes linear time. Hence this executable specification takes cubic time overall.

5 Horner's Rule

Horner's Rule is a well-known technique for reducing the number of arithmetic operations required to evaluate a polynomial. Instantiated to three terms, it states that:

$$
(\mathsf{a} \times \mathsf{b} \times \mathsf{c}) + (\mathsf{b} \times \mathsf{c}) + \mathsf{c} \;=\; ((\mathsf{a} + 1) \times \mathsf{b} + 1) \times \mathsf{c}
$$

We can express this for an arbitrary number—in fact, a list—of terms:

$$
\llparenthesis \mathsf{id}, + \rrparenthesis \circ \llparenthesis \mathsf{id}, \times \rrparenthesis * \circ \mathsf{tails} \;=\; (\mathsf{id}, \circledast)\nrightarrow \qquad \text{where} \quad \mathsf{a} \circledast \mathsf{b} = (\mathsf{a} + 1) \times \mathsf{b}
$$

The left-hand side requires quadratically many multiplications, the right-hand side only linearly many.

Horner's Rule can be generalized further, abstracting from addition and multiplication the actual algebraic properties required. Suppose that

- $\otimes$ distributes backwards through $\oplus$:

$$(\mathsf{a} \oplus \mathsf{b}) \otimes \mathsf{c} \;=\; (\mathsf{a} \otimes \mathsf{c}) \oplus (\mathsf{b} \otimes \mathsf{c})$$

- $\otimes$ has left unit e:

$$\mathsf{e} \otimes \mathsf{a} \;=\; \mathsf{a}$$

Then

$$(\!|\mathsf{id}, \oplus|\!) \circ (\!|\mathsf{id}, \otimes|\!)* \circ \mathsf{tails} \;=\; (\mathsf{id}, \circledast)\not\rightarrow \qquad \text{where} \quad \mathsf{a} \circledast \mathsf{b} = (\mathsf{a} \oplus \mathsf{e}) \otimes \mathsf{b}$$

(In fact, a more general result involving rightwards folds rather than list catamorphisms holds, but we do not need it here.) The proof of Horner's Rule is a straightforward induction.

This generalized Horner's Rule illustrates a major theme of the BMF: to encapsulate common patterns of computation as higher-order operations, and to identify general-purpose theorems concerning these higher-order operations—often with algebraic properties of their component operations as side conditions. Such general-purpose theorems can lead to efficient solutions to algorithmic problems. The MSS problem is a case in point, as we shall now see.

6 Calculating a solution to the MSS problem

We now have all of the machinery needed to derive the linear solution to the MSS problem. In fact, this derivation is particularly attractive because it proceeds as a purely sequential calculation from the clear and simple specification to a very elegant yet non-obvious algorithm. That is, the notations we have developed, together with Horner's Rule, reduce the MSS problem to 'mere calculation'; no creativity is needed in the derivation.

We have:

$$\begin{array}{ll}
& \mathsf{max} \circ \mathsf{sum}* \circ \mathsf{segs} \\
= & \quad \{\ \mathsf{segs}\ \} \\
& \mathsf{max} \circ \mathsf{sum}* \circ \mathsf{flatten} \circ \mathsf{tails}* \circ \mathsf{inits} \\
= & \quad \{\ \text{map promotion}\ \} \\
& \mathsf{max} \circ \mathsf{flatten} \circ \mathsf{sum}** \circ \mathsf{tails}* \circ \mathsf{inits} \\
= & \quad \{\ \text{catamorphism promotion}\ \} \\
& \mathsf{max} \circ \mathsf{max}* \circ \mathsf{sum}** \circ \mathsf{tails}* \circ \mathsf{inits}
\end{array}$$

$$
\begin{array}{ll}
= & \{ \text{ map distributes through composition } \} \\
& \mathsf{max} \circ (\mathsf{max} \circ \mathsf{sum}* \circ \mathsf{tails})* \circ \mathsf{inits} \\
= & \{ \text{ Horner's Rule: let } \mathsf{a} \circledast \mathsf{b} = (\mathsf{a} \uparrow 0) + \mathsf{b} \ \} \\
& \mathsf{max} \circ (\mathsf{id}, \circledast)\not\to* \circ \mathsf{inits} \\
= & \{ \text{ scan } \} \\
& \mathsf{max} \circ (\mathsf{id}, \circledast)\not\!\!\!/\!\!\!\to
\end{array}
$$

Horner's Rule is applicable because $+$ has identity 0 and distributes (backwards) over $\uparrow$.

Acknowledgements

Thanks are due to Sue Gibbons, for improving the presentation of this paper considerably.

References

Roland Backhouse and Paul Hoogendijk (1993). *Elements of a relational theory of datatypes.* In Möller et al. (1993), pages 7–42.

Roland Backhouse, Peter de Bruin, Grant Malcolm, Ed Voermans, and Jaap van der Woude (1991). *Relational catamorphisms.* In B. Möller, editor, *IFIP TC2/WG2.1 Working Conference on Constructing Programs from Specifications*, pages 287–318. North-Holland.

Roland Backhouse, Peter de Bruin, Paul Hoogendijk, Ed Voermans, and Jaap van der Woude (1992a). *Polynomial relators.* In M. Nivat, C. S. Rattray, T. Rus, and G. Scollo, editors, *Proceedings of the 2nd Conference on Algebraic Methodology and Software Technology, AMAST '91.* Springer-Verlag.

Roland Backhouse, Peter de Bruin, Grant Malcolm, Ed Voermans, and Jaap van der Woude (1992b). *A relational theory of datatypes.* In STOP project (1992).

Roland Backhouse (1989). *Making formality work for us.* Bulletin of the European Association for Theoretical Computer Science, (38):219–249. Text of a lecture given at 5th British Colloquium on Theoretical Computer Science, RHBNC, London, March 1989.

Jon Bentley (1986). *Programming Pearls.* Addison-Wesley.

Richard S. Bird and Oege de Moor (1993a). *From dynamic programming to greedy algorithms.* In Möller et al. (1993).

Richard S. Bird and Oege de Moor (1993b). *Solving optimisation problems with catamorphisms.* In R. S. Bird, C. C. Morgan, and J. C. P. Woodcock, editors, *LNCS 669: Mathematics of Program Construction*, pages 45–66. Springer-Verlag.

Richard Bird and Oege de Moor (1994). *Hybrid dynamic programming*. Programming Research Group, Oxford.

Richard S. Bird (1984). *The promotion and accumulation strategies in transformational programming*. ACM Transactions on Programming Languages and Systems, 6(4):487–504. See also (Bird, 1985).

Richard S. Bird (1985). *Addendum to "The promotion and accumulation strategies in transformational programming"*. ACM Transactions on Programming Languages and Systems, 7(3):490–492.

Richard S. Bird (1988). *Lectures on constructive functional programming*. In Manfred Broy, editor, *Constructive Methods in Computer Science*. Springer-Verlag. Also available as Technical Monograph PRG-69, from the Programming Research Group, Oxford University.

Richard S. Bird (1989). *Algebraic identities for program calculation*. Computer Journal, 32(2):122–126.

John Darlington (1981). *The structured description of algorithm derivations*. In J. W. deBakker and H. van Vliet, editors, *Algorithmic Languages*, pages 221–250. Elsevier North-Holland, New York.

Geraint Jones and Mary Sheeran (1990). *Circuit design in Ruby*. In Jørgen Staunstrup, editor, *Formal Methods for VLSI Design*. North-Holland.

Geraint Jones and Mary Sheeran (1992). *Designing arithmetic circuits by refinement in Ruby*. In STOP project (1992).

Wayne Luk, Tim Cheung, Quentin Miller, and Graham Hutton (1994). *Simulating and compiling designs using Rebecca*. Unpublished draft, Programming Research Group, Oxford.

Grant Malcolm (1990). *Data structures and program transformation*. Science of Computer Programming, 14:255–279.

Bernhard Möller, Helmut Partsch, and Steve Schumann, editors (1993). *LNCS 755: IFIP TC2/WG2.1 State-of-the-Art Report on Formal Program Development*. Springer-Verlag.

David B. Skillicorn (1990). *Architecture independent parallel computation*. IEEE Computer, 23(12):38–51.

STOP project (1992). *STOP 1992 Summerschool on Constructive Algorithmics*.