Maximum Segment Sum, Monadically
(distilled tutorial)

Jeremy Gibbons
Department of Computer Science, University of Oxford
Wolfson Building, Parks Road, Oxford OX1 3QD, United Kingdom
jeremy.gibbons@cs.ox.ac.uk

The *maximum segment sum* problem is to compute, given a list of integers, the largest of the sums of the contiguous segments of that list. This problem specification maps directly onto a cubic-time algorithm; however, there is a very elegant linear-time solution too. The problem is a classic exercise in the mathematics of program construction, illustrating important principles such as calculational development, pointfree reasoning, algebraic structure, and datatype-genericity. Here, we take a sideways look at the datatype-generic version of the problem in terms of monadic functional programming, instead of the traditional relational approach; the presentation is tutorial in style, and leavened with exercises for the reader.

1 Introduction

Domain-specific languages are one approach to the general challenge of raising the level of abstraction in constructing software systems. Rather than making use of the same general-purpose tools for all domains of discourse, one identifies a particular domain of interest, and fashions some tools specifically to embody the abstractions relevant to that domain. The intention is that common concerns within that domain are abstracted away within the domain-specific tools, so that they can be dealt with once and for all rather than being considered over and over again for each development within the domain.

Accepting the premise of domain-specific over general-purpose tools naturally leads to an explosion in the number of tools in the programmer's toolbox—and consequently, greater pressure on the tool designer, who needs powerful meta-tools to support the lightweight design of new domain-specific abstractions for each new domain. Language design can no longer be the preserve of large committees and long gestation periods; it must be democratized and streamlined, so that individual developers can aspire to toolsmithery, crafting their own languages to address their own problems.

Perhaps the most powerful meta-tool for the aspiring toolsmith is a programming language expressive enough to host domain-specific *embedded* languages [15]. That is, rather than designing a new domain-specific language from scratch, with specialized syntax and a customized syntax-aware editor, a dedicated parser, an optimization engine, an interpreter or compiler, debugging and profiling systems, and so on, one simply writes a *library* within the host language. This can be constraining—one has to accept the host language's syntax and semantics, which might not entirely match the domain—but it is very lightweight, because one can exploit all the existing infrastructure rather than having to reinvent it.

Essentially, the requirement on the host language is that it provides the right features for capturing new abstractions—things like strong typing, higher-order functions, modules, classes, data abstraction, datatype-genericity, and so on. If the toolsmith can formalize a property of their domain, the host language should allow them to express that formalization within the language. One might say that a sufficiently expressive host language is in fact a *domain-specific language for domain-specific languages*.

Olivier Danvy, Chung-chieh Shan (Eds.): IFIP Working Conference
on Domain-Specific Languages 2011 (DSL 2011).
EPTCS 66, 2011, pp. 181–194, doi:10.4204/EPTCS.66.9

Given a suitably expressive host language, the toolsmith designs a domain-specific language as a library—of combinators, or classes, or modules, or whatever the appropriate abstraction mechanism is in that language. Typically, this consists of a collection of *constructs* (functions, methods, datatypes) together with a collection of *laws* defining an equational theory for those constructs. The pretty-printing libraries of Hughes [16] and Wadler [24] are a good example; but so is the relational algebra that underlies SQL [9].

This tutorial presents an exercise in reasoning with a collection of combinators, representative of the kinds of reasoning that can be done with the constructs and laws of any domain-specific embedded language. We will take Haskell [22] as our host language, since it provides many of the right features for expressing domain-specific embedded languages. However, Haskell is still not perfect, so we will take a somewhat abstract view of it, mixing true Haskell syntax with some mathematical idealizations—our point is the equational reasoning, not the language in which it is expressed. Section 9 provides a brief summary of our notation, and there are some exercises in Section 10.

2 Maximum segment sum

The particular problem we will be considering is a classic exercise in the mathematics of program construction, namely that of deriving a linear-time algorithm for the *maximum segment sum* problem, based on *Horner's Rule*. The problem was popularized in Jon Bentley's *Programming Pearls* column [1] in *Communications of the ACM* (and in the subsequent book [2]), but I learnt about it from my DPhil supervisor Richard Bird's lecture notes on the *Theory of Lists* [4] and *Constructive Functional Programming* [5] and his paper *Algebraic Identities for Program Calculation* [6], which he was working on around the time I started my doctorate. It seems like I'm not the only one for whom the problem is a favourite, because it has since become a bit of a cliché among program calculators; but that won't stop me revisiting it.

The original problem is as follows. Given a list of numbers (say, a possibly empty list of integers), find the largest of the sums of the contiguous segments of that list. In Haskell, this specification could be written like so:

$$\begin{array}{lcl} mss & :: & [Integer] \to Integer \\ mss & = & maximum \cdot map\, sum \cdot segs \end{array}$$

where *segs* computes the contiguous segments of a list:

$$\begin{array}{lcll} \multicolumn{4}{l}{segs, inits, tails :: [\alpha] \to [[\alpha]]} \\ segs & = & concat \cdot map\, inits \cdot tails & \\ tails & = & foldr\, f\, [[\,]] & \textbf{where}\ f\, x\, xss = (x : head\, xss) : xss \\ inits & = & foldr\, g\, [[\,]] & \textbf{where}\ g\, x\, xss = [\,] : map\, (x :)\, xss \end{array}$$

and *sum* computes the sum of a list of integers, and *maximum* the maximum of a nonempty list of integers:

$$\begin{array}{lcl} \multicolumn{3}{l}{sum, maximum :: [Integer] \to Integer} \\ sum & = & foldr\, (+)\, 0 \\ maximum & = & foldr_1\, (\sqcup) \end{array}$$

(Here, $\sqcup$ denotes binary maximum.) This specification is executable, but takes cubic time; the problem is to do better.

We can get quite a long way just using standard properties of *map*, *inits*, and so on. It is straightforward (see Exercise 1) to calculate that

$$mss = maximum \cdot map\, (maximum \cdot map\, sum \cdot inits) \cdot tails$$

If we can write $\mathit{maximum} \cdot \mathit{map}\,\mathit{sum} \cdot \mathit{inits}$ in the form $\mathit{foldr}\,h\,e$, then the *map* of this can be fused with the *tails* to yield $\mathit{scanr}\,h\,e$; this observation is known as the *Scan Lemma*. Moreover, if h takes constant time, then this gives a linear-time algorithm for *mss*.

The crucial observation is based on Horner's Rule for evaluation of polynomials, which is the first important thing you learn in numerical computing—I was literally taught it in secondary school, in my sixth-year classes in mathematics. Here is its familiar form:

$$\sum_{i=0}^{n-1} a_i x^i = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1} = a_0 + x(a_1 + x(a_2 + \cdots + x\,a_{n-1}))$$

but the essence of the rule is about sums of products (see Exercise 2):

$$\sum_{i=0}^{n-1} \prod_{j=0}^{i-1} u_j = 1 + u_0 + u_0 u_1 + \cdots + u_0 u_1 \ldots u_{n-1} = 1 + u_0(1 + u_1(1 + \cdots + u_{n-1}))$$

Expressed in Haskell, this is captured by the equation

$$\mathit{sum} \cdot \mathit{map}\,\mathit{product} \cdot \mathit{inits} = \mathit{foldr}\,(\oplus)\,e \quad \textbf{where}\ e = 1\,;\, u \oplus z = e + u \times z$$

(where $\mathit{product} = \mathit{foldr}\,(\times)\,1$ computes the product of a list of integers).

But Horner's Rule is not restricted to sums and products; the essential properties are that addition and multiplication are associative, that multiplication has a unit, and that multiplication distributes over addition. This the algebraic structure of a *semiring* (but without needing commutativity of addition). In particular, the so-called *tropical semiring* on the integers, in which "addition" is binary maximum and "multiplication" is integer addition, satisfies the requirements. So for the maximum segment sum problem, we get

$$\mathit{maximum} \cdot \mathit{map}\,\mathit{sum} \cdot \mathit{inits} = \mathit{foldr}\,(\oplus)\,e \quad \textbf{where}\ e = 0\,;\, u \oplus z = e \sqcup (u + z)$$

Moreover, $\oplus$ takes constant time, so this gives a linear-time algorithm for *mss* (see Exercise 3).

3 Tail segments, datatype-generically

About a decade after the initial "theory of lists" work on the maximum segment sum problem, Richard Bird, Oege de Moor, and Paul Hoogendijk came up with a datatype-generic version of the problem [3]. It's fairly clear what "maximum" and "sum" mean generically, but not so clear what "segment" means for nonlinear datatypes; the point of their paper is basically to resolve that issue.

Recalling the definition of *segs* in terms of *inits* and *tails*, we see that it would suffice to develop datatype-generic notions of "initial segment" and "tail segment". One fruitful perspective is given in Bird & co's paper: a "tail segment" of a cons list is just a subterm of that list, and an "initial segment" is the list but with some tail (that is, some subterm) replaced with the empty structure.

So, representing a generic "tail" of a data structure is easy: it's a data structure of the same type, and a subterm of the term denoting the original structure. A datatype-generic definition of *tails* is a little trickier, though. For lists, you can see it as follows: *every node of the original list is labelled with the subterm of the original list rooted at that node*. I find this a helpful observation, because it explains why the *tails* of a list is one element longer than the list itself: a list with n elements has $n+1$ nodes (n conses and a nil), and each of those nodes gets labelled with one of the $n+1$ subterms of the list. Indeed, *tails* ought morally to take a possibly empty list and return a *non-empty list* of possibly empty lists—there are two different datatypes involved. Similarly, if one wants the "tails" of a data structure of a type in which some nodes have no labels (such as leaf-labelled trees, or indeed such as the "nil" constructor of lists), one needs a variant of the datatype providing labels at those positions. Also, for a data structure in which some nodes have multiple labels, or in which there are different types of labels, one needs a variant for which *every node has precisely one label.*

Bird & co call this the *labelled variant* of the original datatype; if the original is a polymorphic datatype $\mathsf{T}\,\alpha = \mu(\mathsf{F}\,\alpha)$ for some binary shape functor F, then the labelled variant is $\mathsf{L}\,\alpha = \mu(\mathsf{G}\,\alpha)$ where $\mathsf{G}\,\alpha\,\beta = \alpha \times \mathsf{F}\,1\,\beta$—whatever α-labels F may or may not have specified are ignored, and precisely one α-label per node is provided. Given this insight, it is straightforward to define a datatype-generic variant *subterms* of the *tails* function:

$$\mathit{subterms}_{\mathsf{F}} = \mathit{fold}_{\mathsf{F}}(\mathit{in}_{\mathsf{G}} \cdot \mathit{fork}(\mathit{in}_{\mathsf{F}} \cdot \mathsf{F}\,\mathit{id}\,\mathit{root}, \mathsf{F}\,!\,\mathit{id})) :: \mathsf{T}\,\alpha \to \mathsf{L}(\mathsf{T}\,\alpha)$$

where $\mathit{root} = \mathit{fst} \cdot \mathit{in}_{\mathsf{G}}^{-1} = \mathit{fold}_{\mathsf{G}}\mathit{fst} :: \mathsf{L}\,\alpha \to \alpha$ returns the root label of a labelled data structure, and $!_{\alpha} = (\lambda a\,.\,()) :: \alpha \to 1$ is the unique arrow to the unit type. (Informally, having computed the tree of subterms for each child of a node, we make the tree of subterms for the node itself by assembling all the child trees with the label for this node; the label should be the whole structure rooted at this node, which can be reconstructed from the roots of the child trees.) What's more, there's a datatype-generic scan lemma too:

$$\begin{array}{lcl} \mathit{scan}_{\mathsf{F}} & :: & (\mathsf{F}\,\alpha\,\beta \to \beta) \to \mathsf{T}\,\alpha \to \mathsf{L}\,\beta \\ \mathit{scan}_{\mathsf{F}}\,f & = & \mathsf{L}\,(\mathit{fold}_{\mathsf{F}}\,f) \cdot \mathit{subterms}_{\mathsf{F}} \\ & = & \mathit{fold}_{\mathsf{F}}(\mathit{in}_{\mathsf{G}} \cdot \mathit{fork}(f \cdot \mathsf{F}\,\mathit{id}\,\mathit{root}, \mathsf{F}\,!\,\mathit{id})) \end{array}$$

(Again, the label for each node can be constructed from the root labels of each of the child trees.) In fact, *subterms* and *scan* are paramorphisms [19], and can also be nicely written coinductively as well as inductively [20].

4 Initial segments, datatype-generically

What about a datatype-generic "initial segment"? As suggested above, that's obtained from the original data structure by replacing some subterms with the empty structure. Here I think Bird & co sell themselves a little short, because they insist that the datatype T supports empty structures, which is to say, that F is of the form $\mathsf{F}\,\alpha\,\beta = 1 + \mathsf{F}'\,\alpha\,\beta$ for some F'. This isn't necessary: for an arbitrary F, we can easily manufacture the appropriate datatype U of "data structures in which some subterms may be replaced by empty", by defining $\mathsf{U}\,\alpha = \mu(\mathsf{H}\,\alpha)$ where $\mathsf{H}\,\alpha\,\beta = 1 + \mathsf{F}\,\alpha\,\beta$.

As with *subterms*, the datatype-generic version of *inits* is a bit trickier—and this time, the special case of lists is misleading. You might think that because a list has just as many initial segments as it does tail segments, so the labelled variant ought to suffice just as well here too. But this doesn't work for non-linear data structures such as trees—in general, there are many more "initial" segments than "tail" segments (because one can make independent choices about replacing subterms with the empty structure in each child), and they don't align themselves conveniently with the nodes of the original structure.

The approach I prefer here is just to use a collection type to hold the "initial segments"; that is, a monad. This could be the monad of finite lists, or of finite bags, or of finite sets—we will defer until later the discussion about precisely which monad, and write simply M. That the monad corresponds to a collection class amounts to it supporting a "union" operator $(\uplus) :: \mathsf{M}\,\alpha \times \mathsf{M}\,\alpha \to \mathsf{M}\,\alpha$ for combining two collections (append, bag union, and set union, respectively, for lists, bags, and sets), and an "empty" collection $\emptyset :: \mathsf{M}\,\alpha$ as the unit of $\uplus$, both of which the *join* of the monad should distribute over [17]:

$$\begin{array}{lcl} \mathit{join}\,\emptyset & = & \emptyset \\ \mathit{join}\,(x \uplus y) & = & \mathit{join}\,x \uplus \mathit{join}\,y \end{array}$$

(Some authors also add the axiom $\mathit{join}\,(\mathsf{M}\,(\lambda a\,.\,\emptyset)\,x) = \emptyset$, making $\emptyset$ in some sense both a left and a right zero of composition.) You can think of a computation of type $\alpha \to \mathsf{M}\,\beta$ in two equivalent ways: as a nondeterministic mapping from an α to one of many—or indeed, no—possible βs, or as a deterministic function from an α to the collection of all such βs. The choice of monad distinguishes different flavours of nondeterminism; for example, the finite bag monad models nondeterminism in which the multiplicity of computations yielding the same result is significant, whereas with the finite set monad the multiplicity is not significant.

Now we can implement the datatype-generic version of *inits* by nondeterministically pruning a data structure by arbitrarily replacing some subterms with the empty structure; or equivalently, by generating the collection of all such prunings.

$$\mathit{prune} = \mathit{fold}_{\mathsf{F}}(\mathsf{M}\,\mathit{in}_{\mathsf{H}} \cdot \mathit{opt}\,\mathit{Nothing} \cdot \mathsf{M}\,\mathit{Just} \cdot \delta_2) :: \mu(\mathsf{F}\,\alpha) \to \mathsf{M}(\mu(\mathsf{H}\,\alpha))$$

Here, *opt* supplies a new alternative for a nondeterministic computation:

$$\mathit{opt}\,a\,x = \mathit{return}\,a \uplus x$$

and $\delta_2 :: (\mathsf{F}\,\alpha)\mathsf{M} \mathbin{\dot{\to}} \mathsf{M}(\mathsf{F}\alpha)$ distributes the shape functor F over the monad M (which can be defined for all *traversable* functors $\mathsf{F}\,\alpha$—we'll say more about this in Section 7). Informally, once you have computed all possible ways of pruning each of the children of a node, a pruning of the node itself is formed either as *Just* some node assembled from arbitrarily pruned children, or *Nothing* for the empty structure.

5 Horner's Rule, datatype-generically

As we've seen, the essential property behind Horner's Rule is one of distributivity, for example of product over sum. In the datatype-generic case, we will model this as follows. We are given an $(\mathsf{F}\,\alpha)$-algebra (β, f), and a M-algebra (β, k); you might think of these as "datatype-generic product" and "collection sum", respectively. Then there are two different methods of computing a β result from an $\mathsf{F}\,\alpha\,(\mathsf{M}\,\beta)$ structure: we can either distribute the $\mathsf{F}\,\alpha$ structure over the collection(s) of βs, compute the "product" f of each structure, and then compute the "sum" k of the resulting products; or we can "sum" each collection, then compute the "product" of the resulting structure, as illustrated in the following diagram.

$$\begin{tikzcd}
\mathsf{F}\,\alpha\,(\mathsf{M}\,\beta) \arrow[r, "\delta_2"] \arrow[d, "\mathsf{F}\,\mathit{id}\,k"'] & \mathsf{M}(\mathsf{F}\,\alpha\,\beta) \arrow[r, "\mathsf{M}\,f"] & \mathsf{M}\,\beta \arrow[d, "k"] \\
\mathsf{F}\,\alpha\,\beta \arrow[rr, "f"'] & & \beta
\end{tikzcd}$$

Distributivity of "product" over "sum" is the property that these two different methods agree. For example, with $f :: \mathsf{F}\,\mathbb{N}\,\mathbb{N} \to \mathbb{N}$ adding all the naturals in an F-structure, and $k :: \mathsf{M}\,\mathbb{N} \to \mathbb{N}$ finding the maximum of a collection of naturals (returning 0 for the empty collection), the diagram commutes (see Exercise 8). (To match up with the rest of the story, we have presented distributivity in terms of a bifunctor F, although the first parameter α plays no role. We could just have well have used a unary functor, dropping the α, and changing the distributor to $\delta :: \mathsf{F}\mathsf{M} \mathbin{\dot{\to}} \mathsf{M}\mathsf{F}$.)

Note that (β, k) is required to be an algebra for the monad M. This means that it is not only an algebra for M as a functor (namely, of type $\mathsf{M}\,\beta \to \beta$), but also it should respect the extra structure of the monad: $k \cdot \mathit{return} = \mathit{id}$ and $k \cdot \mathit{join} = k \cdot \mathsf{M}\,k$. For the special case of monads of collections, these amount to what were called *reductions* in the old Theory of Lists [4] work—functions k of the form $\oplus/$ for binary operator $\oplus :: \beta \times \beta \to \beta$, distributing over union: $\oplus/(x \uplus y) = (\oplus/x) \oplus (\oplus/y)$ (see Exercise 9). A consequence of this distributivity property is that $\oplus$ has to satisfy all the properties that $\uplus$ does—for example, if $\uplus$ is associative, then so must $\oplus$ be, and so on, and in particular, since $\uplus$ has a unit $\emptyset$, then $\oplus$ too must have a unit $e_{\oplus} :: \beta$, and $\oplus/\emptyset = e_{\oplus}$ is forced (see Exercise 10).

Recall that we modelled an "initial segment" of a structure of type $\mu(\mathsf{F}\,\alpha)$ as being of type $\mu(\mathsf{H}\,\alpha)$, where $\mathsf{H}\,\alpha\,\beta = 1 + \mathsf{F}\,\alpha\,\beta$. We need to generalize "product" to work on this extended structure, which is to say, we need to specify the value b of the "product" of the empty structure too. Then we have $\mathit{maybe}\,b\,f :: \mathsf{H}\,\alpha\,\beta \to \beta$, so that $\mathit{fold}_{\mathsf{H}}(\mathit{maybe}\,b\,f) :: \mu(\mathsf{H}\,\alpha) \to \beta$.

The datatype-generic version of Horner's Rule is then about computing the "sum" of the "products" of each of the "initial segments" of a data structure:

$$\oplus/ \cdot \mathsf{M}(\mathit{fold}_{\mathsf{H}}(\mathit{maybe}\,b\,f)) \cdot \mathit{prune}$$

We can use fold fusion to show that this composition can be computed as a single fold, $\mathit{fold}_{\mathsf{F}}((b\oplus) \cdot f)$, given the distributivity property $\oplus/ \cdot \mathsf{M}\,f \cdot \delta_2 = f \cdot \mathsf{F}\,\mathit{id}\,(\oplus/)$ above (see Exercise 12). Curiously, it doesn't seem to matter what value is chosen for b.

We're nearly there. We start with the traversable shape bifunctor F, a collection monad M, and a distributive law $\delta_2 :: (\mathsf{F}\,\alpha)\mathsf{M} \mathrel{\dot{\to}} \mathsf{M}(\mathsf{F}\alpha)$. We are given an $(\mathsf{F}\,\alpha)$-algebra (β, f), an additional element $b :: \beta$, and a M-algebra $(\beta, \oplus/)$, such that f and $\oplus$ take constant time and f distributes over $\oplus/$ in the sense above. Then we can calculate (see Exercise 13) that

$$\oplus/ \cdot \mathsf{M}(\mathit{fold}_{\mathsf{H}}(\mathit{maybe}\,b\,f)) \cdot \mathit{segs} = \oplus/ \cdot \mathit{contents}_{\mathsf{L}} \cdot \mathit{scan}_{\mathsf{F}}((b\oplus) \cdot f)$$

where

$$\mathit{segs} = \mathit{join} \cdot \mathsf{M}\,\mathit{prune} \cdot \mathit{contents}_{\mathsf{L}} \cdot \mathit{subterms} :: \mu(\mathsf{F}\,\alpha) \to \mathsf{M}(\mu(\mathsf{H}\,\alpha))$$

and where $\mathit{contents}_{\mathsf{L}} :: \mathsf{L} \mathrel{\dot{\to}} \mathsf{M}$ computes the contents of an L-structure (which, like δ_2, can be defined using the traversability of F). The scan can be computed in linear time, because its body takes constant time; moreover, the "sum" $\oplus/$ and *contents* can also be computed in linear time (indeed, they can even be fused into a single pass).

For example, with $f :: \mathsf{F}\,\mathbb{Z}\,\mathbb{Z} \to \mathbb{Z}$ adding all the integers in an F-structure, $b = 0 :: \mathbb{Z}$, and $\oplus :: \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$ returning the greater of two integers, we get a datatype-generic version of the linear-time maximum segment sum algorithm.

6 Distributivity reconsidered

There's a bit of hand-waving in Section 5 to justify the claim that the commuting diagram there really is a kind of distributivity. What does it have to do with the familiar equation $a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$ capturing distributivity of one binary operator $\otimes$ over another, $\oplus$?

Recall that $\delta_2 :: (\mathsf{F}\,\alpha)\mathsf{M} \mathrel{\dot{\to}} \mathsf{M}(\mathsf{F}\,\alpha)$ distributes the shape functor F over the monad M in its second argument; this is the form of distribution over "effects" that crops up in the datatype-generic Maximum Segment Sum problem. More generally, this works for any idiom M; this will be important below.

Generalizing in another direction, one might think of distributing over an idiom in both arguments of the bifunctor, via an operator $\delta : \mathsf{F} \cdot (\mathsf{M} \times \mathsf{M}) \mathrel{\dot{\to}} \mathsf{M} \cdot \mathsf{F}$, which is to say, $\delta_{\beta} :: \mathsf{F}\,(\mathsf{M}\beta)\,(\mathsf{M}\beta) \to \mathsf{M}(\mathsf{F}\beta)$, natural in the β. This is the *bidist* method of the *Bitraversable* subclass of *Bifunctor* that Bruno Oliveira and I used in our paper [14] on the ITERATOR pattern; informally, it requires just that F has a finite ordered sequence of "element positions". Given δ, one can define $\delta_2 = \delta \cdot \mathsf{F}\,\mathit{pure}\,\mathit{id}$.

That traversability (or equivalently, distributivity over effects) for a bifunctor F is definable for any idiom, not just any monad, means that one can also conveniently define an operator $\mathit{contents}_{\mathsf{H}} : \mathsf{H} \dot{\to} \mathsf{List}$ for any traversable unary functor H. This is because the constant functor $\mathsf{K}_{[\beta]}$ is an idiom: the *pure* method returns the empty list, and idiomatic application appends two lists. Then one can define

$$\mathit{contents}_{\mathsf{H}} = \delta \cdot \mathsf{H}\,\mathit{wrap}$$

where *wrap* makes a singleton list. For a traversable bifunctor F, we define $\mathit{contents}_{\mathsf{F}} = \mathit{contents}_{\mathsf{F}\triangle}$ where $\triangle$ is the diagonal functor; that is, $\mathit{contents}_{\mathsf{F}} :: \mathsf{F}\,\beta\,\beta \to [\beta]$, natural in the β. (No constant functor is a monad, except in trivial categories, so this convenient definition of contents doesn't work monadically. Of course, one can use a writer monad, but this isn't quite so convenient, because an additional step is needed to extract the output.)

One important axiom of δ, suggested by Ondřej Rypáček [23], is that it should be "natural in the contents": it should leave shape unchanged, and depend on contents only up to the extent of their ordering. Say that a natural transformation $\phi : \mathsf{F} \dot{\to} \mathsf{G}$ between traversable functors F and G "preserves contents" if $\mathit{contents}_{\mathsf{G}} \cdot \phi = \mathit{contents}_{\mathsf{F}}$. Then, in the case of unary functors, the formalization of "naturality in the contents" requires δ to respect content-preserving ϕ:

$$\delta_{\mathsf{G}} \cdot \phi = \mathsf{M}\phi \cdot \delta_{\mathsf{F}} : \mathsf{TM} \dot{\to} \mathsf{MG} \quad \Leftarrow \quad \mathit{contents}_{\mathsf{G}} \cdot \phi = \mathit{contents}_{\mathsf{F}}$$

In particular, $\mathit{contents}_{\mathsf{F}} : \mathsf{F} \dot{\to} \mathsf{List}$ itself preserves contents, and so we expect

$$\delta_{\mathsf{List}} \cdot \mathit{contents}_{\mathsf{F}} = \mathsf{M}(\mathit{contents}_{\mathsf{F}}) \cdot \delta_{\mathsf{F}}$$

to hold.

Happily, the same generic operation $\mathit{contents}_{\mathsf{F}}$ provides a datatype-generic means to "fold" over the elements of an F-structure. Given a binary operator $\otimes :: \beta \times \beta \to \beta$ and an initial value $b :: \beta$, we can define an $(\mathsf{F}\,\beta)$-algebra (β, f)—that is, a function $f :: \mathsf{F}\,\beta\,\beta \to \beta$—by

$$f = \mathit{foldr}\,(\otimes)\,b \cdot \mathit{contents}_{\mathsf{F}}$$

This is a slight specialization of the presentation of the datatype-generic MSS problem; there we had $f :: \mathsf{F}\,\alpha\,\beta \to \beta$. The specialization arises because we are hoping to define such an f given a homogeneous binary operator $\otimes$. On the other hand, the introduction of the initial value b is no specialization, as we needed such a value for the "product" of an empty "segment" anyway. This "generic folding" construction is just what is provided by Ross Paterson's *Data.Foldable* Haskell library [21].

7 Reducing distributivity

The general principle about traversals underlying Rypáček's paper [23] on labelling data structures is that it is often helpful to reduce a general problem about traversal over arbitrary datatypes to a more specific one about lists, exploiting the "naturality in contents" property of traversal. We'll use that tactic for the distributivity property in the datatype-generic version Horner's Rule.

Consider the following diagram.

$$\begin{tikzcd}
\mathsf{F}\,\beta\,(\mathsf{M}\beta) \arrow[r, "\mathsf{F}\,\mathit{return}\,\mathit{id}"'] \arrow[rr, bend left, "\delta_2", "(1)"'] \arrow[dd, "\mathsf{F}\,\mathit{id}\,(\oplus/)"'] \arrow[ddr, phantom, "(4)"] & \mathsf{F}\,(\mathsf{M}\beta)\,(\mathsf{M}\beta) \arrow[r, "\delta"'] \arrow[d, "\mathit{contents}_{\mathsf{F}}"] \arrow[ddl, "\mathsf{F}\,(\oplus/)\,(\oplus/)"] \arrow[dr, phantom, "(6)"] & \mathsf{M}(\mathsf{F}\,\beta\,\beta) \arrow[d, "\mathsf{M}\mathit{contents}_{\mathsf{F}}"'] \arrow[dr, "\mathsf{M}f", "(2)"'] & \\
 & {[\mathsf{M}\beta]} \arrow[r, "\delta"'] \arrow[d, "\mathsf{List}(\oplus/)"] \arrow[dl, phantom, "(5)"] & \mathsf{M}[\beta] \arrow[r, "\mathsf{M}(\mathit{foldr}\,(\otimes)\,b)"'] \arrow[dr, phantom, "(7)"] & \mathsf{M}\beta \arrow[d, "\oplus/"] \\
\mathsf{F}\,\beta\,\beta \arrow[r, "\mathit{contents}_{\mathsf{F}}"'] \arrow[rrr, bend right, "f"', "(3)"] & {[\beta]} \arrow[rr, "\mathit{foldr}\,(\otimes)\,b"'] & & \beta
\end{tikzcd}$$

The perimeter is just the commuting diagram given in Section 5—the diagram we have to justify. Face (1) is the definition of δ_2 in terms of δ. Faces (2) and (3) are the expansion of f as generic folding of an F-structure. Face (4) follows from $\oplus/$ being an M-algebra, and hence being a left-inverse of *return*. Face (5) is an instance of the naturality property of $\mathit{contents}_{\mathsf{F}} : \mathsf{F}\triangle \dot{\to} \mathsf{List}$. Face (6) is the property that δ respects the contents-preserving transformation $\mathit{contents}_{\mathsf{F}}$. Therefore, the whole diagram commutes if Face (7) does—so let's focus on Face (7):

$$\begin{tikzcd}
{[\mathsf{M}\,\beta]} \arrow[r, "\delta_{\mathsf{List}}"] \arrow[d, "\mathsf{List}(\oplus/)"'] & \mathsf{M}[\beta] \arrow[r, "\mathsf{M}(\mathit{foldr}\,(\otimes)\,b)"] & \mathsf{M}\,\beta \arrow[d, "\oplus/"] \\
{[\beta]} \arrow[rr, "\mathit{foldr}\,(\otimes)\,b"'] & & \beta
\end{tikzcd}$$

Demonstrating that this diagram commutes is not too difficult, because both sides turn out to be list folds. Around the left and bottom edges, we have a fold $\mathit{foldr}\,(\otimes)\,b$ after a map $\mathsf{List}\,(\oplus/)$, which automatically fuses to $\mathit{foldr}\,(\odot)\,b$, where $\odot$ is defined by $x \odot a = (\oplus/x) \otimes a$, or, pointlessly, $(\odot) = (\otimes) \cdot (\oplus/) \times \mathit{id}$. Around the top and right edges we have the composition $\oplus/ \cdot \mathsf{M}(\mathit{foldr}\,(\otimes)\,b) \cdot \delta_{\mathsf{List}}$. If we can write δ_{List} as an instance of *foldr*, we can then use the fusion law for *foldr* to prove that this composition equals $\mathit{foldr}\,(\odot)\,b$ (see Exercise 15).

In fact, there are various equivalent ways of writing δ_{List} as an instance of *foldr*. The definition given by Conor McBride and Ross Paterson in their original paper on idioms [18] looked like the identity function, but with added idiomness:

$$\begin{array}{lcl}
\delta_{\mathsf{List}}\,[\,] & = & \mathit{pure}\,[\,] \\
\delta_{\mathsf{List}}\,(\mathit{mb} : \mathit{mbs}) & = & \mathit{pure}\,(:) \circledast \mathit{mb} \circledast \delta_{\mathsf{List}}\,\mathit{mbs}
\end{array}$$

In the special case that the idiom is a monad, it can be written in terms of liftM_0 (aka *return*) and liftM_2:

$$\begin{array}{lcl}
\delta_{\mathsf{List}}\,[\,] & = & \mathit{liftM}_0\,[\,] \\
\delta_{\mathsf{List}}\,(\mathit{mb} : \mathit{mbs}) & = & \mathit{liftM}_2\,(:)\,\mathit{mb}\,(\delta_{\mathsf{List}}\,\mathit{mbs})
\end{array}$$

But we'll use a third equivalent definition:

$$\begin{array}{lcl} \delta_{\mathsf{List}}\,[\,] & = & \mathit{return}\,[\,] \\ \delta_{\mathsf{List}}\,(mb : mbs) & = & \mathsf{M}(:)\,(cp\,(mb, \delta_{\mathsf{List}}\,mbs)) \end{array}$$

where

$$\begin{array}{lcl} cp & :: & \mathsf{M}\,\alpha \times \mathsf{M}\,\beta \to \mathsf{M}(\alpha \times \beta) \\ cp\,(x,y) & = & \mathit{join}\,(\mathsf{M}\,(\lambda a\,.\,\mathsf{M}\,(a,)\,y)\,x) \end{array}$$

That is,

$$\delta_{\mathsf{List}} = \mathit{foldr}\,(\mathsf{M}(:) \cdot cp)\,(\mathit{return}\,[\,])$$

In the use of fold fusion in demonstrating distributivity for lists (Exercise 15), we are naturally lead to a distributivity condition

$$\oplus/ \cdot \mathsf{M}(\otimes) \cdot cp = (\otimes) \cdot (\oplus/) \times (\oplus/)$$

for *cp*. This in turn follows from corresponding distributivity properties for collections (see Exercise 16),

$$\begin{array}{lcl} \oplus/ \cdot \mathsf{M}(a\otimes) & = & (a\otimes) \cdot \oplus/ \\ \oplus/ \cdot \mathsf{M}(\otimes b) & = & (\otimes b) \cdot \oplus/ \end{array}$$

which can finally be discharged by induction over the size of the (finite!) collections (see Exercise 17).

8 Conclusion

As the title of their paper [3] suggests, Bird & co carried out their development using the relational approach set out in the *Algebra of Programming* book [8]; for example, their version of *prune* is a relation between data structures and their prunings, rather than being a function that takes a structure and returns the collection of all its prunings. There's a well-known isomorphism between relations and set-valued functions, so their relational approach roughly looks equivalent to the monadic one taken here.

I've known their paper well for over a decade (I made essential use of the "labelled variant" construction in my own papers on generic downwards accumulations [10, 11]), but I've only just noticed that although they discuss the maximum segment sum problem, they don't discuss problems based on other semirings, such as the obvious one of integers with addition and multiplication—which is, after all, the origin of Horner's Rule. Why not? It turns out that the relational approach doesn't work in that case!

There's a hidden condition in the calculation, which relates back to our earlier comment about which collection monad—finite sets, finite bags, lists, etc—to use. When M is the set monad, distribution over choice ($\oplus/(x \uplus y) = (\oplus/x) \oplus (\oplus/y)$)—and consequently the condition $\oplus/ \cdot \mathit{opt}\,b = (b\oplus) \cdot \oplus/$ that we used in proving Horner's Rule—requires $\oplus$ to be idempotent, because $\uplus$ itself is idempotent; but addition is not idempotent. For exactly this reason, the distributivity property does not in fact hold for addition with the set monad. But everything does work out with the bag monad, for which $\uplus$ is not idempotent. The bag monad models a flavour of nondeterminism in which multiplicity of results matters—as it does for the sum-of-products instance of the problem, when two copies of the same segment should be treated differently from just one copy. Similarly, if the order of results matters—if, for example, we were looking for the "first" solution—then we would have to use the list monad rather than bags or sets. The moral of the story is that the relational approach is *programming with just one monad*, namely the set monad; if that monad doesn't capture your effects faithfully, you're stuck.

(On the other hand, there are aspects of the problem that work much better relationally than they do functionally. We have carefully used maximum only for a linear order, namely the usual ordering of the integers. A non-antisymmetric order is more awkward monadically, because there need not be a unique maximal value. For example, it is not so easy to compute “the” segment with maximal sum, because there may be several such. We could refine the ordering by sum on segments to make it once more a partial order, perhaps breaking ties lexically; but we have to take care to preserve the right distributivity properties. Relationally, however, finding the maximal elements of a finite collection under a partial order works out perfectly straightforwardly. We can try the same trick of turning the relation “maximal under a partial order” into the collection-valued function “all maxima under a partial order”, but the equivalent trick on the ordering itself—turning the relation “$<$” into the collection-valued function “all values less than this one”—runs into problems by taking us outside the world of *finite* nondeterminism.)

References

[1] Jon Bentley (1984): *Programming Pearls: Algorithm Design Techniques*. Communications of the ACM 27(9), pp. 865–873, doi:10.1145/358234.381162.

[2] Jon Bentley (1986): *Programming Pearls*. Addison-Wesley.

[3] Richard Bird, Oege de Moor & Paul Hoogendijk (1996): *Generic Functional Programming with Types and Relations*. Journal of Functional Programming 6(1), pp. 1–28, doi:10.1017/S0956796800001556.

[4] Richard S. Bird (1987): *An Introduction to the Theory of Lists*. In Manfred Broy, editor: Logic of Programming and Calculi of Discrete Design, Springer-Verlag, pp. 3–42. Available at `http://web.comlab.ox.ac.uk/publications/publication3837-abstract.html`. NATO ASI Series F Volume 36.

[5] Richard S. Bird (1988): *Lectures on Constructive Functional Programming*. In Manfred Broy, editor: Constructive Methods in Computer Science, Springer-Verlag, pp. 151–218. Available at `http://web.comlab.ox.ac.uk/publications/publication3849-abstract.html`. NATO ASI Series F Volume 55.

[6] Richard S. Bird (1989): *Algebraic Identities for Program Calculation*. Computer Journal 32(2), pp. 122–126, doi:10.1093/comjnl/32.2.122.

[7] Richard S. Bird (1998): *Introduction to Functional Programming Using Haskell*. Prentice-Hall.

[8] Richard S. Bird & Oege de Moor (1996): *Algebra of Programming*. International Series in Computer Science, Prentice-Hall.

[9] E. F. Codd (1970): *A Relational Model of Data for Large Shared Data Banks*. Communications of the ACM 13, pp. 377–387, doi:10.1145/362384.362685.

[10] Jeremy Gibbons (1998): *Polytypic Downwards Accumulations*. In Johan Jeuring, editor: Proceedings of Mathematics of Program Construction, Lecture Notes in Computer Science 1422, Springer-Verlag, Marstrand, Sweden, pp. 207–233, doi:10.1007/BFb0054292. Available at `http://www.comlab.ox.ac.uk/oucl/work/jeremy.gibbons/publications/polyda.ps.gz`.

[11] Jeremy Gibbons (2000): *Generic Downwards Accumulations*. Science of Computer Programming 37, pp. 37–65, doi:10.1016/S0167-6423(99)00022-2. Available at `http://www.comlab.ox.ac.uk/oucl/work/jeremy.gibbons/publications/genda.ps.gz`.

[12] Jeremy Gibbons (2003): *Origami Programming*. In Gibbons & de Moor [13], pp. 41–60. Available at `http://www.comlab.ox.ac.uk/oucl/work/jeremy.gibbons/publications/origami.pdf`.

[13] Jeremy Gibbons & Oege de Moor, editors (2003): *The Fun of Programming*. Cornerstones in Computing, Palgrave. ISBN 1-4039-0772-2.

[14] Jeremy Gibbons & Bruno César dos Santos Oliveira (2009): *The Essence of the Iterator Pattern*. Journal of Functional Programming 19(3,4), pp. 377–402, doi:10.1017/S0956796809007291. Available at `http://www.comlab.ox.ac.uk/jeremy.gibbons/publications/iterator.pdf`.

[15] Paul Hudak (1996): *Building Domain-Specific Embedded Languages*. ACM Computing Surveys 28, doi:10.1145/242224.242477.

[16] John Hughes (1995): *The Design of a Pretty-Printing Library*. In Johan Jeuring & Erik Meijer, editors: *Advanced Functional Programming*, *Lecture Notes in Computer Science* 925, Springer-Verlag, pp. 53–96, doi:10.1007/3-540-59451-5_3. Lecture notes from the First International Spring School on Advanced Functional Programming Techniques, Båstad, Sweden.

[17] S. Kazem Lellahi & Val Tannen (1997): *A Calculus for Collections and Aggregates*. In Eugenio Moggi & Giuseppe Rosolini, editors: *Category Theory and Computer Science*, *Lecture Notes in Computer Science* 1290, Springer, pp. 261–280, doi:10.1007/BFb0026993.

[18] Conor McBride & Ross Paterson (2008): *Applicative Programming with Effects*. *Journal of Functional Programming* 18(1), pp. 1–13, doi:10.1017/S0956796807006326.

[19] Lambert Meertens (1992): *Paramorphisms*. *Formal Aspects of Computing* 4(5), pp. 413–424, doi:10.1007/BF01211391. Available at `http://computerscience.nl/research/techreps/RUU-CS-90-04.html`.

[20] Erik Meijer, Maarten Fokkinga & Ross Paterson (1991): *Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire*. In John Hughes, editor: *Functional Programming Languages and Computer Architecture*, *Lecture Notes in Computer Science* 523, Springer-Verlag, pp. 124–144, doi:10.1007/3540543961_7.

[21] Ross Paterson (2005): *Data.Foldable library for Haskell*. Available at `http://www.haskell.org/ghc/docs/6.12.2/html/libraries/base-4.2.0.1/src/Data-Foldable.html`.

[22] Simon Peyton Jones (2003): *The Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, doi:10.1017/S0956796803000315. Available at `http://haskell.org/onlinereport/`.

[23] Ondřej Rypáček (2010): *Labelling Polynomial Functors: A Coherent Approach*. Available at `http://www.dcs.kcl.ac.uk/staff/rypacek/labelers.pdf`. Manuscript.

[24] Philip Wadler (2003): *A Prettier Printer*. In Gibbons & de Moor [13], pp. 223–243. Available at `http://homepages.inf.ed.ac.uk/wadler/papers/prettier/prettier.pdf`. ISBN 1-4039-0772-2.

9 Appendix: Notation

For the benefit of those not fluent in Haskell and the Algebra of Programming approach, this appendix presents some basic notations. For a more thorough introduction, see the books by Richard Bird [7, 8] and my lecture notes on “origami programming” [12].

Types: Our programs are typed; the statement “$x :: \alpha$” declares that variable or expression x has type α. We use product types $\alpha \times \beta$ (with morphism $\mathit{fork} :: (\alpha \to \beta) \times (\alpha \to \gamma) \to (\alpha \to \beta \times \gamma)$), sum types $\alpha + \beta$, and function types $\alpha \to \beta$. We assume throughout that types represent sets, and functions are total.

Functions: Function application is usually denoted by juxtaposition, “$f\,x$”, and is left-associative and tightest-binding. Function composition is backwards, so $(f \cdot g)\,x = f\,(g\,x)$.

Operators: It is often convenient to write binary operators in infix notation; this makes many algebraic equations more perspicuous. We use sections $(a\oplus)$ and $(\oplus b)$ for partially applied binary operators, so that $(a\oplus)\,b = a \oplus b = (\oplus b)\,a$. In contrast to Haskell, we consider binary operators uncurried; for example, $(+) :: \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$.

Lists: We use the Haskell syntax “$[\alpha]$” for a list type, “$[\,]$” for the empty list, “$a : x$” for cons, “$+\!\!+$” for append, and “$[1,2,3]$” for a list constant. The fold $\mathit{foldr} :: (\alpha \times \beta \to \beta) \to \beta \to [\alpha] \to \beta$ is ubiquitous; it has the universal property

$$h = \mathit{foldr}\,f\,e \quad \Leftrightarrow \quad h\,[\,] = e \wedge h \cdot (:) = f \cdot \mathit{id} \times h$$

and as a special case of this, the fusion law

$$h \cdot \mathit{foldr}\, f\, e = \mathit{foldr}\, f'\, e' \quad \Leftarrow \quad h\, e = e' \wedge h \cdot f = f' \cdot \mathit{id} \times h$$

The function $\mathit{map} :: (\alpha \to \beta) \to [\alpha] \to [\beta]$ is an instance, via $\mathit{map}\, f = \mathit{foldr}\, ((:) \cdot f \times \mathit{id})\, []$. So is *scanr*, which computes the fold of every tail of a list:

$$\begin{array}{lcl} \mathit{scanr} & :: & (\alpha \times \beta \to \beta) \to \beta \to [\alpha] \to [\beta] \\ \mathit{scanr}\, f\, e & = & \mathit{foldr}\, h\, [e] \quad \textbf{where}\ h\, a\, (b : x) = f\, a\, b : (b : x) \end{array}$$

We also use the variant $\mathit{foldr}_1\, f\, (x +\!\!+ [a]) = \mathit{foldr}\, f\, a\, x$ on non-empty lists.

Functors: Datatypes are modelled as functors, which are operations on both types and functions; so for F a functor, $\mathsf{F}\,\alpha$ is a type whenever α is, and if $f :: \alpha \to \beta$ then $\mathsf{F}\, f :: \mathsf{F}\,\alpha \to \mathsf{F}\,\beta$. Moreover, F respects the compositional structure of functions, preserving identity ($\mathsf{F}\, \mathit{id}_\alpha = \mathit{id}_{\mathsf{F}\alpha}$) and composition ($\mathsf{F}\,(f \cdot g) = \mathsf{F}\, f \cdot \mathsf{F}\, g$). For example, List is a functor, with $\mathsf{List}\,\alpha = [\alpha]$ and $\mathsf{List}\, f = \mathit{map}\, f$. We generalize this also to bifunctors, which are binary operators functorial in each argument; for example, we will see the bifunctor $\mathsf{L}\,\alpha\,\beta = 1 + \alpha \times \beta$ below, as the "shape functor" for lists.

Naturality: Polymorphic functions are modelled as natural transformations between functors. A natural transformation $\phi : \mathsf{F} \dot{\to} \mathsf{G}$ is a family of functions $\phi_\alpha :: \mathsf{F}\,\alpha \to \mathsf{G}\,\alpha$, one for each α, coherent in the sense of being related by the naturality condition $\mathsf{G}\, h \cdot \phi_\alpha = \phi_\beta \cdot \mathsf{F}\, h$ whenever $h :: \alpha \to \beta$.

Datatype-genericity: Datatype-generic programming is expressed in terms of parametrization by a functor. In particular, for a large class of bifunctors F (including all those built from constants and the identity using sums and products—the polynomial bifunctors), we can form a kind of least fixed point $\mathsf{T}\,\alpha = \mu(\mathsf{F}\,\alpha)$ of F in its second argument, giving an inductive datatype. It is a "fixed point" in the sense that $\mathsf{T}\,\alpha \simeq \mathsf{F}\,\alpha\,(\mathsf{T}\,\alpha)$; so $\mathsf{List}\,\alpha = \mu(\mathsf{L}\,\alpha)$, where L is the shape functor for lists defined above. We sometimes use Haskell-style datatype definitions, which conveniently name the constructors too:

$$\textbf{data}\ \mathsf{List}\,\alpha = \mathit{Nil} \mid \mathit{Cons}\,(\alpha, \mathsf{List}\,\alpha)$$

Algebras: An F-algebra is a pair (α, f) such that $f :: \mathsf{F}\,\alpha \to \alpha$. A homomorphism between F-algebras (α, f) and (β, g) is a function $h :: \alpha \to \beta$ such that $h \cdot f = g \cdot \mathsf{F}\, h$. One half of the isomorphism by which an inductive datatype is a fixed point is given by the constructor $\mathit{in}_\mathsf{F} :: \mathsf{F}\,\alpha\,(\mathsf{T}\,\alpha) \to \mathsf{T}\,\alpha$, through which $(\mathsf{T}\,\alpha, \mathit{in}_\mathsf{F})$ forms an $(\mathsf{F}\,\alpha)$-algebra. The datatype is the "least" fixed point in the sense that there is a unique homomorphism to any other $(\mathsf{F}\,\alpha)$-algebra (β, f); we say that $(\mathsf{T}\,\alpha, \mathit{in}_\mathsf{F})$ is the initial $(\mathsf{F}\,\alpha)$-algebra. We write $\mathit{fold}_\mathsf{F}\, f$ for that unique homomorphism; its uniqueness is captured in the universal property

$$h = \mathit{fold}_\mathsf{F}\, f \quad \Leftrightarrow \quad h \cdot \mathit{in}_\mathsf{F} = f \cdot \mathsf{F}\, h$$

Monads: A monad M is a functor with two additional natural transformations, a multiplication $\mathit{join} : \mathsf{M}\mathsf{M} \dot{\to} \mathsf{M}$ and a unit $\mathit{return} : \mathsf{Id} \dot{\to} \mathsf{M}$ (where Id is the identity functor), that satisfy three laws:

$$\begin{array}{lcl} \mathit{join} \cdot \mathit{return} & = & \mathit{id} \\ \mathit{join} \cdot \mathsf{M}\, \mathit{return} & = & \mathit{id} \\ \mathit{join} \cdot \mathsf{M}\, \mathit{join} & = & \mathit{join} \cdot \mathit{join} \end{array}$$

Collection types such as finite lists, bags, and sets form monads; in each case, *return* yields a singleton collection, and *join* unions a collection of collections into a collection. Another monad we will use is Haskell's "maybe" datatype and associated morphism

$$\textbf{data}\ \mathsf{Maybe}\,\alpha = \mathit{Nothing} \mid \mathit{Just}\,\alpha$$

$$\begin{array}{lcl} \mathit{maybe}\, e\, f\, \mathit{Nothing} & = & e \\ \mathit{maybe}\, e\, f\, (\mathit{Just}\, a) & = & f\, a \end{array}$$

for which $return = Just$ and $join = maybe\,Nothing\,id$. An algebra for a monad M is an M-algebra (α, f) for M as a functor, satisfying the extra conditions

$$\begin{array}{lcl} f \cdot return &=& id \\ f \cdot join &=& f \cdot \mathsf{M}\, f \end{array}$$

Idioms: An idiom M is a functor with two additional natural transformations, whose components are $pure_\alpha :: \alpha \rightarrow \mathsf{M}\,\alpha$ and $\circledast_{\alpha,\beta} :: \mathsf{M}\,(\alpha \rightarrow \beta) \times \mathsf{M}\,\alpha \rightarrow \mathsf{M}\,\beta$, satisfying four laws:

$$\begin{array}{lcl} pure\,id \circledast u &=& u \\ pure\,(\cdot) \circledast u \circledast v \circledast w &=& u \circledast (v \circledast w) \\ pure\,f \circledast pure\,a &=& pure\,(f\,a) \\ u \circledast pure\,a &=& pure\,(\lambda f\,.\,f\,a) \circledast u \end{array}$$

Any monad induces an idiom; so does any constant functor K_α, provided that there is a monoidal structure on α.

10 Appendix: Exercises

1. (See page 182.) Calculate that

$$mss = maximum \cdot map\,(maximum \cdot map\,sum \cdot inits) \cdot tails$$

just using the definitions of $mss, inits, tails$, together with (i) distributivity of map over function composition, (ii) naturality of $concat$, that is, $map\,f \cdot concat = concat \cdot map\,(map\,f)$, and (iii) that $maximum$ is a list homomorphism, that is, $maximum \cdot concat = maximum \cdot map\,maximum$.

2. (See page 183.) Use the sum-of-products version of Horner's Rule to prove the more familiar polynomial version.

3. (See page 183.) Hand-simulate the execution of the linear-time algorithm for *mss*

$$mss = foldr\,(\oplus)\,e \quad \textbf{where}\; e = 0\,; u \oplus z = e \sqcup (u + z)$$

on the list $[4, -5, 6, -3, 2, 0, -4, 5, -6, 5]$. Do you understand how it works?

4. (See page 183.) Apart from $(+, \times)$ and $(\sqcup, +)$, what other semirings do you know, and what variations on the "maximum segment sum" problem do they suggest?

5. (See page 184.) Verify that the labelled variant of the usual datatype of lists (namely, $\mathsf{List}\,\alpha = \mu(\mathsf{F}\,\alpha)$ where shape functor F is given by $\mathsf{F}\,\alpha\,\beta = 1 + \alpha \times \beta$) is a datatype of nonempty lists. What is the labelled variant of externally-labelled binary trees, whose shape functor is $\mathsf{F}\,\alpha\,\beta = \alpha + \beta \times \beta$? That of internally-labelled binary trees, whose shape functor is $\mathsf{F}\,\alpha\,\beta = 1 + \alpha \times \beta \times \beta$? And homogeneous binary trees, whose shape functor is $\mathsf{F}\,\alpha\,\beta = \alpha + \alpha \times \beta \times \beta$?

6. (See page 184.) If you're familiar with paramorphisms and with anamorphisms (unfolds), write $subterms_\mathsf{F}$ and $scan_\mathsf{F}$ as instances of these.

7. (See page 185.) Hand-simulate the execution of *prune* in the finite bag monad on a small homogeneous binary tree, such as the term $Fork\,1\,(Leaf\,2)\,(Fork\,3\,(Leaf\,1)\,(Leaf\,4))$ of type

$$\textbf{data}\; \mathsf{Tree}\,\alpha = Leaf\,\alpha \mid Fork\,(\alpha, \mathsf{Tree}\,\alpha, \mathsf{Tree}\,\alpha)$$

What happens on externally-labelled binary trees? Internally-labelled? How does the result differ if you let M be sets rather than bags?

8. (See page 185.) Pick a shape functor F and a collection monad M; give suitable definitions of $f :: \mathsf{F}\,\mathbb{N}\,\mathbb{N} \to \mathbb{N}$ to sum all naturals in an F-structure and $k :: \mathsf{M}\,\mathbb{N} \to \mathbb{N}$ to find the maximum of a collection of naturals; and verify that the rectangle in Section 5 commutes.

9. (See page 186.) Given an M-algebra k, show that k distributes over $\uplus$—there exists a binary operator $\oplus$ such that $k\,(x \uplus y) = k\,x \oplus k\,y$. (Hint: define $a \oplus b = k\,(\mathit{return}\,a \uplus \mathit{return}\,b$.)

10. (See page 186.) Given an M-algebra $\oplus/$ that distributes over $\uplus$, show that $\oplus$ is associative if $\uplus$ is; similarly for commutativity and idempotence. Show also that if $\uplus$ has a unit $\emptyset$, then $\oplus$ also has a unit, which must equal $\oplus/\emptyset$. (Hint: show first that $\oplus/$ is surjective.)

11. (See page 186.) Using the universal property of $\mathit{fold}_{\mathsf{F}}$, prove the fold fusion law

$$h \cdot \mathit{fold}_{\mathsf{F}}\,f = \mathit{fold}_{\mathsf{F}}\,g \quad \Leftarrow \quad h \cdot f = g \cdot \mathsf{F}\,h$$

Use this to prove the special case of fold-map fusion

$$\mathit{fold}_{\mathsf{F}}\,f \cdot \mathsf{T}\,g = \mathit{fold}_{\mathsf{F}}\,(f \cdot \mathsf{F}\,g\,\mathit{id})$$

where $\mathsf{T}\,\alpha = \mu(\mathsf{F}\,\alpha)$.

12. (See page 186.) Use fold fusion to calculate a characterization of $\oplus/\cdot\mathsf{M}(\mathit{fold}_{\mathsf{H}}(\mathit{maybe}\,b\,f))\cdot\mathit{prune}$ as a fold, assuming the distributivity property $\oplus/\cdot\mathsf{M}\,f\cdot\delta_2 = f\cdot\mathsf{F}\,\mathit{id}\,(\oplus/)$.

13. (See page 186.) Show by calculation that

$$\oplus/\cdot\mathsf{M}(\mathit{fold}_{\mathsf{H}}(\mathit{maybe}\,b\,f))\cdot\mathit{segs} = \oplus/\cdot\mathit{contents}_{\mathsf{L}}\cdot\mathit{scan}_{\mathsf{F}}((b\oplus)\cdot f)$$

14. (See page 188.) Convert the big commuting diagram in Section 7 into an equational proof of the distributivity property $\oplus/\cdot\mathsf{M}\,f\cdot\delta_2 = f\cdot\mathsf{F}\,\mathit{id}\,(\oplus/)$, assuming the properties captured by each of the individual faces.

15. (See page 188.) Use fold fusion to prove that

$$\mathit{foldr}\,(\otimes)\,b\cdot\mathsf{List}\,(\oplus/) = \oplus/\cdot\mathsf{M}(\mathit{foldr}\,(\otimes)\,b)\cdot\delta_{\mathsf{List}}$$

assuming the distributivity property $\oplus/\cdot\mathsf{M}(\otimes)\cdot\mathit{cp} = (\otimes)\cdot(\oplus/)\times(\oplus/)$ of cp.

16. (See page 189.) Prove the distributivity property for cartesian product

$$\oplus/\cdot\mathsf{M}(\otimes)\cdot\mathit{cp} = (\otimes)\cdot(\oplus/)\times(\oplus/)$$

assuming the two distributivity properties $\oplus/\cdot\mathsf{M}(a\otimes) = (a\otimes)\cdot\oplus/$ and $\oplus/\cdot\mathsf{M}(\otimes b) = (\otimes b)\cdot\oplus/$ for collections.

17. (See page 189.) Prove the two distributivity properties for collections

$$\begin{aligned} \oplus/\cdot\mathsf{M}(a\otimes) &= (a\otimes)\cdot\oplus/ \\ \oplus/\cdot\mathsf{M}(\otimes b) &= (\otimes b)\cdot\oplus/ \end{aligned}$$

by induction over the size of the (finite!) collection, assuming that binary operator $\otimes$ distributes over $\oplus$ in the familiar sense (that is, $a\otimes(b\oplus c) = (a\otimes b)\oplus(a\otimes c)$).