Virtual Data Structures

Doaitse Swierstra[1] *and Oege de Moor*[2]

[1] Department of Computer Science, Utrecht University, P.O. Box 80.089, 3508 TB Utrecht, The Netherlands
[2] Programming Research Group, Computing Laboratory, 11 Keble Road, Oxford OX1 3QD, United Kingdom

1 Introduction

The purpose of this paper is to demonstrate a number of techniques which may be used in calculating algorithms for sequence-oriented problems. It may be considered as a further step in the development of a programming method which started with [5]. The basic observation underlying this method is that algorithms can be the result of a systematic development, in which all design decisions and applied insights are clearly identifiable. One might even claim that the essence of an algorithm is its derivation, and not the program text that results from such a derivation.

Originally the theory for algorithm derivation was presented in an imperative setting, using predicate calculus as the main tool for reasoning about the various steps in a derivation. A complicating factor has been that, in deriving a program, the designer has to cope with two different formalisms and an often intricate relationship, namely between the programming language itself and the formalism for reasoning about the program under development. Unfortunately, the rich expressiveness of the predicate calculus, which has been used for the latter, has not forced algorithm developers to express themselves in more structured and more abstract ways. As a result this approach can still be considered as fairly ad hoc, although various attempts have been made to merge the two components of the formalism [10].

In [7] a more calculational style of program development has been advocated, in which the program itself, instead of appearing as a side-effect of a derivation, is the subject of transformations. The idea is that the process of developing an algorithm consists of a sequence of transformations, starting with an inefficient specification, which may even be nonexecutable, and resulting in an executable, efficient program. In [2] this process has been demonstrated to be effective, introducing the so-called *Segment Decomposition Theorem*, which captures many instances of the design steps *replacing a constant by a variable* and *strengthening the invariant*, which are well known from the imperative school [6].

The relative success of this approach lies in a well-chosen combination of a notation for functions, which lends itself to easy manipulation, and the introduction of higher order functions which enables a more abstract view of many algorithms. As a result of this work the insight has come that control constructs like *maps*, *filters* and *reductions*, which were originally used in a a sequence oriented setting, could be formulated more abstractly by making use of the concept of a *homomorphism*. Homomorphisms naturally come into existence when one generalises *reduce* over lists (corresponding to looping over an array in an imperative setting) to any inductively defined type. Generalisations of other oft-occurring control structures, *i.e.* recursion patterns, has led to a wealth of other classes of morphisms; an overview is given in [9].

The way data structures are considered has shifted in recent years from a more implementation and representation oriented view towards an algebraic view. Within the field of transformational functional programming, the beginning of this shift was marked by the identification of the Boom[4]-hierarchy ("boom" is also the Dutch word for tree), which shows clearly the algebraic relationship between binary trees, sequences, bags and sets. This hierarchy is established by starting from the signature describing binary trees, and subsequently adding laws to this signature:

$$\mathsf{JV} ::= \Box : \epsilon \;|\; [-] : \mathsf{A} \;|\; +\!\!\!+ : \mathsf{JV} \times \mathsf{JV},$$

where we are assuming that A is the base type over which the structures are defined and that the following law holds for the unit element $\Box$:

$$\Box +\!\!\!+\, s = s +\!\!\!+\, \Box = s$$

If $+\!\!\!+$ satisfies no further laws, JV is the data type of unlabelled binary trees. Other data types can be obtained by postulating further laws. One thus obtains the hierarchy summarised below:

associativity	$(x +\!\!\!+ y) +\!\!\!+ z = x +\!\!\!+ (y +\!\!\!+ z)$	**sequences**
commutativity	$x +\!\!\!+ y = y +\!\!\!+ x$	**bags or multisets**
idempotency	$x +\!\!\!+ x = x$	**sets**

The structure of this paper is as follows. In Sect. 2 we will introduce the class of problems that is the subject of this paper. In Sect. 3 we will introduce the concepts which are necessary to understand the derivations to be given in Sect. 5. In Sect. 4 we will introduce the concept of a *virtual data structure*, which allows one to optimise certain function compositions. The importance of this optimisation is that one can reason at the level of function composition as long as possible, only eliminating intermediate results in the final transformation step.

In this paper there is no separate section on notation. This reflects our view that at the current state of research, there is no such thing as a fixed notation. Indeed, part of the research is to discover which concepts are worth a notation of their own, and what notations are useful in finding derivations. For this reason notation will be introduced on the fly.

The calculus used in this paper will be rather informal. Current active research in this area has provided a formal underpinning of the presented notation, and of the laws formulated using the notation. These foundations are however not the subject of the research described here; the interested reader is referred to [1][11]. A reader feeling that a proper understanding of functional programming is lacking is referred to [3].

At the end of this section we want to stress that most of the techniques we are presenting here have been known for a long time in the area of compiler construction, and as such belong to the tool-box of most compiler writers. It is the different use and formulation of these concepts which makes them applicable in a calculational style of program derivation. Where appropriate we will indicate such correspondences.

2 The Problem Class

In this paper we study problems of the following form:

$$\mathsf{f} = \mathsf{compose}/ \circ \mathsf{property}^* \circ \mathsf{generator} \quad , \tag{1}$$

where $\circ$ denotes function composition. The generator is a set-valued function that generates candidates for a solution. For each of these candidates a property is computed (hence the * symbol, which indicates that the function is supposed to be applied to each element of the sequence, bag or set which is returned by the generating part), and finally all these intermediate results are used in computing the final value, by making use of a so-called *reduction* in combination with the binary function compose. Hence if $\mathsf{generator}.x = \{ a_0, a_1, \ldots, a_n \}$ then

$$\mathsf{f}.x = (\mathsf{property}.a_0)\mathsf{compose}(\mathsf{property}.a_1)\mathsf{compose}(\mathsf{property}.a_2)\ldots(\mathsf{property}.a_n) \quad .$$

A typical generator is the function segs, which returns all contiguous subsequences of its arguments:

$$\mathsf{segs}.x = \{y | \exists u, v : u \mathbin{+\!\!+} y \mathbin{+\!\!+} v = x\} \quad .$$

Other examples of generators are

$$\begin{aligned} \mathsf{inits}.[a_0, a_1, \ldots, a_n] = \{ & [\,], [a_0], [a_0, a_1], \\ & \ldots, \\ & [[a_0, a_1, \ldots, a_n]] \} \end{aligned}$$

$$\begin{aligned} \mathsf{splits}.[a_0, a_1, \ldots, a_n] = \{ & ([\,], [a_0, a_1, \ldots, a_n]), \\ & ([a_0], [a_1, a_2, \ldots, a_n]), \\ & \ldots, \\ & ([a_0, a_1, \ldots, a_{n-1}], [a_n]), \\ & ([a_0, a_1, \ldots, a_n], [\,]) \} \quad . \end{aligned}$$

For example the following specification says that a sequence is ascending:

$$\mathsf{asc} = \wedge/ \circ (\leq \circ (\uparrow/ \times \downarrow/))^* \circ \mathsf{splits} \quad ,$$

where $\uparrow/$ computes the maximum value of a sequence, by reducing it with the maximum operator $\uparrow$, and $\downarrow/$ the minimum value. The functions splits is defined by

$$\mathsf{splits}.x = \{\langle u, v\rangle | u \mathbin{+\!\!+} v = x\} \quad .$$

The $\times$ operator takes two functions as its arguments and returns a function which applies these functions to both components of its argument and returns the pair of the two results, i.e.:

$$(f \times g).\langle x, y\rangle = \langle f.x, g.y\rangle \quad .$$

These pairs are subsequently each subjected to a comparison ($\leq$), and finally the conjunction of all these comparisons is taken ($\wedge/$). In words, this would mean that we consider a sequence ascending if, whichever the way a sequence has been split in two, the maximum value of the left part is always at most the minimum value of the right part.

Another *property* that might occur in our generic specification is that a sequence is *balanced*:

$$(=0) \circ +/ \quad ,$$

which describes that the sum of the elements should be equal to zero. Yet a third example of a property is *low*:

$$(\leq) \circ \langle \uparrow/, \# \rangle,$$

which indicates that the maximum value of a structure should be less than its size, returned by the function $\#$. We have used $\langle f, g \rangle .x = \langle f.x, g.x \rangle$.

A further useful operator we will use is the filter $\triangleleft$, which takes as its left operand a predicate, applies this function to all elements of a structure, and returns the structure containing all elements which satisfy the predicate. So the specification of a function which computes the length of the longest ascending sequence might be written as:

$$\mathsf{lup} = \uparrow/ \circ \#^{*} \circ \mathsf{asc} \triangleleft \circ \mathsf{subs} \quad .$$

3 Design Steps

In this section we discuss a strategy for solving problems of the aforementioned kind; in doing so we will show how different design decisions may lead to totally different solutions, which are not easily related without taking their derivations into consideration. The strategy consists of both the identification of a number of *decision points* where a design decision has to be taken, and the identification of the *alternatives* between which a choice may be made.

As we will see these decision points are described in a rather blunt way, using informal phrases like *pick the right one*. It will not always be immediately clear what will be the right choice, and learning what will be the right choice is a matter of experience, trial and error. Keep in mind, however, that once we have found the right path, it will be possible to clearly indicate which decisions have been taken, and probably even why those decisions eventually turned out to be successful. In this respect the process of deriving more and more algorithms resembles the process of putting up maps at the branches in a highly structured maze; a more useful approach than putting up maps in a flat desert without any landmarks.

3.1 Algebraic Views

The Concept of a View. When giving an algebraic specification of an abstract data type, it is easily overlooked that such a description is not unique. Most programming languages only provide a single built-in construct for a specific class of data types, and thus favour only one specific *algebraic view*. Often it is possible to provide a different view, containing different operations and different laws. What makes two different views equivalent is that it is possible to express the operations of one view in the operations of the other view and vice versa, and that the laws obeyed by these mapped operations remain valid in the target view.

The choice of an algebraic view resembles the choice for a specific context free grammar in describing a language. There may be many grammars describing the same language, as there are many views describing the same data. The parsing process corresponds in this case to a conversion from one view, i.e. a list of characters with some structure, into a value of the initial data type of another view, in this case described by the grammar.

Possible Views. In this paragraph we will discuss some useful views on sequences. We stress however that the number of alternatives is almost unbounded, and that having a library of such possible views at hand is quite a useful tool in deriving algorithms.

Most functional languages take the *cons* view for granted in which sequences over A are elements of the following initial data type for which we will reserve the word *list*:

$$\mathsf{CV} ::= \Box : \epsilon \mid \succ\!\!\!+ : \mathsf{A} \times \mathsf{CV} \quad .$$

Equally acceptable, and well known views are provided by the *join* view:

$$\mathsf{JV} ::= \Box : \epsilon \mid [-] : \mathsf{A} \mid +\!\!+ : \mathsf{JV} \times \mathsf{JV} \quad ,$$

the *snoc* view:

$$\mathsf{SV} ::= \Box : \epsilon \mid +\!\!\!\prec : \mathsf{SV} \times \mathsf{A} \quad ,$$

the *labelled tree* view:

$$\mathsf{LTV} ::= \Box : \epsilon \mid - \prec - \succ - : \mathsf{LTV} \times \mathsf{A} \times \mathsf{LTV} \quad ,$$

and the *spine* view:

$$\mathsf{SPV} ::= \Box : \epsilon \mid \prec : \mathsf{SPV} \times (\mathsf{LTV} \times \mathsf{A}) \quad ,$$

which is a mixture between the labelled tree view and the snoc view. In [13] some examples of views and the conversions between them may be found. In the sequel we will encounter such a conversion.

We will now introduce our first decision point.

> *Algebraic View: Inspect what algebraic view the program is supposed to accept, and decide whether it is useful to convert the data into a different view.*

In the example derivations we will encounter some criteria on which such a decision might be based. Notice however that this observation is not a very deep one; it is just a different way of expressing that we store the input to be treated into a convenient data structure, which is a common step in program design.

Catamorphisms. The views introduced in the previous parts may all be considered as the definitions of initial algebras, where the operators may be considered as term-constructors. Since we will encounter many (uniquely defined) homomorphisms from such initial algebras into other algebras we will give them a special name, i.e. *catamorphisms*, and introduce a special notation for such morphisms. Since such a catamorphism is uniquely defined by the algebra that is its codomain, we will denote them by summing up the operators associated with the codomain between banana-brackets.

As an example consider the function length, which computes the length of a list, and which might be defined as follows:

$$\begin{aligned}\mathsf{length}.(a \succ\!\!\!+\, x) &= 1 + (\mathsf{length}.x)\\ \mathsf{length}.\Box &= 0 \quad .\end{aligned}$$

In our notation this function, which is a catamorphism from lists to the algebra of integers, may be written as $(\!|0, (1\hat{+})|\!)$, where $(1\hat{+})$ is the operator corresponding to the $\succ\!\!\!+$-constructor and 0 the operator corresponding to the $\Box$-constructor. In this notation the $\hat{+}$ is a so-called lifted operator which is defined by $(f\hat{+}g).\langle x, y\rangle = (f.x) + (g.y)$, and 1 is the constant function always returning 1. We will use the expression $\otimes/$ as an abbreviation for catamorphisms of the form $(\!|1_{\otimes}, \otimes|\!)$ operating on CV or SV, when $(1_{otimes}, \otimes)$ is a monoid.

As a final notion in this section we define the so-called *accumulation* $/\!\!/$, which may be expressed in terms of reductions and inits by:

$$\oslash/\!\!/ = \oslash/^{*} \circ \mathsf{inits} \quad . \tag{2}$$

The concept is important in the derivation of many programs, since its use in general introduces a considerable reduction in the complexity of the program. So it is the case for lists that the left hand side of (2) may be computed in $O(n)$ steps, whereas a naive implementation of the right hand side will take $O(n^2)$ steps. That this is indeed the case may be deduced from the following alternative, but equivalent, definition of $/\!\!/$:

$$\begin{aligned}\oslash/\!\!/ &= \pi_2 \circ (\!|\langle 1_{\oslash}, [1_{\oslash}]\rangle, \tilde{\oslash}|\!)\\ \langle a, b\rangle \tilde{\oslash} c &= \langle a \oslash c, b +\!\!\!\prec (a \oslash c)\rangle \quad .\end{aligned}$$

3.2 Algebraic laws

Once a specific algebraic view has been chosen, there is often a further choice to be made. Due to the algebraic laws associated with the view, there may exist freedom in the way a value is represented, and thus the conversion from the given data type into the required data type is a non-deterministic function (or relation). The laws define equivalence classes of terms, and any element of such a class might be chosen. Depending on the further functions which will be applied to the representation, it may be worthwhile to make use of the available freedom to choose a specific element from the class.

Here we will discuss shortly some of the laws associated with the different views. There are no laws associated with CV and SV views on sequences: the representation of a sequence as a list is unique, and the associativity of the $+\!\!\!+$-operator is heavily used when converting from the join-view to the cons- or snoc-view. When representing bags in CV however we will get the following additional law, representing the translation of the law of commutativity as formulated in JV in the previous section.

$$a \rightarrowtail (b \rightarrowtail x) = b \rightarrowtail (a \rightarrowtail x) \quad ,$$

and the following law for representing the idempotency of set union:

$$a \rightarrowtail (a \rightarrowtail x) = (a \rightarrowtail x) \quad .$$

A similar line of reasoning holds for LTV, where we have assumed that every next law is used in the context of the laws introduced earlier, *i.e.* the formulation of the law of commutativity makes use of associativity:

$$\begin{array}{ll}
\textbf{associativity}: & (x \prec a \succ y) \prec b \succ z = x \prec a \succ (y \prec b \succ z) \\
\textbf{commutativity}: & x \prec a \succ y = y \prec a \succ x \\
\textbf{idempotency}: & \Box \prec a \succ \Box \prec a \succ y = \Box \prec a \succ y
\end{array}$$

and for the spine-view, assuming the corresponding laws for LTV part are introduced accordingly:

$$\begin{array}{ll}
\textbf{associativity}: & (x \prec \langle k, a\rangle) \prec \langle l, b\rangle = x \prec \langle (k \prec a \succ l), b\rangle \\
\textbf{commutativity}: & (x \prec \langle k, a\rangle) \prec \langle l, b\rangle = (x \prec \langle k, b\rangle) \prec \langle l, a\rangle \\
\textbf{idempotency}: & (x \prec \langle l, a\rangle) \prec \langle l, b\rangle = x \prec \langle Nil \prec a \succ Nil \prec b \succ l\rangle
\end{array}$$

We will conclude this part with formulating our second decision point:

> *Algebraic Laws: Inspect what algebraic laws apply to the chosen view, and make use of these laws to pick useful elements from congruence classes of terms.*

As a guideline in choosing the right element one may inspect properties of the generators and further functions in the expression at hand.

One may compare this freedom of choice with the use of an ambiguous context free grammar, giving rise to several different parse trees for a given sentence. So a grammar might contain a production of the form:

$$< \text{expr} >::=< \text{expr} >< \text{operator} >< \text{expr} >,$$

and a parser might use the priorities of the operators to return a parse tree which reflects the intended meaning of the expression.

3.3 Generators

Once a specific view has been chosen and a special element within this view for representing the value at hand, there is still choice left in picking a convenient function for generating the candidates for further processing. This choice is surprisingly rich.

In [2] only a two views are considered, i.e. CV and SV. Neither of these views has laws associated with it, and a single form of generating segments is studied, leading to the aforementioned segment decomposition theorem.

The generator used for generating all segments may be defined as:

$$\mathsf{segs} = +\!\!+/ \circ \mathsf{tails}^* \circ \mathsf{inits} \tag{3}$$

$$\begin{aligned} \mathsf{inits}.(x \leftarrowtail a) &= (\mathsf{inits}.x) \leftarrowtail (x \leftarrowtail a) \\ \mathsf{inits}.\square &= \square \leftarrowtail \square \end{aligned} \tag{4}$$

$$\begin{aligned} \mathsf{tails}.(a \rightarrowtail x) &= (a \rightarrowtail x) \rightarrowtail (\mathsf{tails}.x) \\ \mathsf{tails}.\square &= \square \rightarrowtail \square \quad , \end{aligned} \tag{5}$$

which corresponds to summing the segments up by grouping them according to common end-points. Unfortunately, in this definition neither inits nor tails is expressed as a catamorphism, and thus we provide a different set of definitions which are:

$$\begin{aligned} \mathsf{inits}.(a \rightarrowtail x) &= \square \rightarrowtail ((a\rightarrowtail)^* \circ \mathsf{inits}.x) \\ \mathsf{inits}.\square &= \square \rightarrowtail \square \end{aligned} \tag{6}$$

$$\begin{aligned} \mathsf{tails}.(x \leftarrowtail a) &= ((\leftarrowtail a)^* \circ \mathsf{tails}.x) \leftarrowtail \square \\ \mathsf{tails}.\square &= \square \leftarrowtail \square \quad . \end{aligned} \tag{7}$$

A different approach for generating the tails in SV and CV, which will be useful in finding a different solution for the problem of the maximum segment sum, is to remove all initial segments from a sequence:

$$\mathsf{tails} = +\!\!+/ \circ \overrightarrow{\mathsf{X}}_{\gg} \circ \langle \mathsf{id}, \mathsf{inits} \rangle \tag{8}$$

with:

$$\overrightarrow{\mathsf{X}}_{\gg} . \langle l, m \rangle = (l \gg)^*.m \quad ,$$

where $\gg$ is defined by $(a +\!\!+ b) \gg a = b$ and provided of course that the list l contains only proper prefixes of m, which is indeed the case in (8).

For JV there are several approaches too for summing up all segments, e.g.:

$$\begin{aligned} \mathsf{segs}.\square \quad &= \square \rightarrowtail \square \\ \mathsf{segs}.(x +\!\!+ y) &= \mathsf{segs}.x +\!\!+ \mathsf{segs}.y +\!\!+ +\!\!+^*.(\mathsf{tails}.x \mathsf{X} \mathsf{inits}.y) \quad . \end{aligned}$$

This generator makes it easy to identify all segments which span a specific point in the sequence. Of course the function mapping the input into this view should select that representation where the point to be inspected first ends up highest in the term representing the sequence in JV. A useful generator corresponding to LTV is defined as follows;

$$\begin{aligned} \mathsf{segs}.\square &= \square \rightarrowtail \square \\ \mathsf{segs}.(x \prec a \succ y) &= \mathsf{segs}.x +\!\!+ \mathsf{segs}.y +\!\!+ (a\odot)^*.(\mathsf{tails}.x \ \mathsf{X} \ \mathsf{inits}.y) \\ &\quad \textbf{where } a \odot \langle u, v \rangle = u +\!\!+ [a] +\!\!+ v \end{aligned} \tag{9}$$

The advantage of this view is that it may be used to separate out those segments which have a specific element in common, again assuming of course that a useful representative has been chosen.

We will now formulate our last decision point.

Choice of generator: Choose a useful generator. This choice depends primarily on the chosen view, and on further functions to be applied to the result of the chosen generator.

We finally want to note that in providing the different generators we have used the JV for representing the generated segments. One should keep in mind however that, when considering these segments as a bag, similar choices are to be made for these intermediate data structures.

4 Virtual Data Structures

In this section we will introduce an important transformation, which enables us to construct a link with the imperative style of programming. Through the abundant use of function composition in functional programs, many data structures are first constructed and immediately afterwards inspected and destroyed by the next function to be applied.

In many implementations it will, by using lazy evaluation in the machine model, not be the case that the data structure is ever completely present in memory. In these models the constructing phase and the destructing phase act like coroutines, with the constructing phase producing the next part of the data structure only when it is needed by the destructing phase. This phenomenon is also present when using the *pipe* construct in the Unix operating system. One process is filling the pipe, as if it were writing on a file, and another process is reading from the pipe as if it were a file. The complete file, however, is never present as such. It is to be noticed that this technique may introduce a considerable space optimisation, but does not reduce the number of computation steps.

A simple example of this phenomenon may be found in the following program, which sums the integers from 1 to n:

$$\begin{aligned} a \ \mathsf{upto}\ b &= a \succ\!\!\!+ (a+1)\ \mathsf{upto}\ b, \mathbf{if}\ a \le b \\ a \ \mathsf{upto}\ b &= \square, \mathbf{otherwise} \\ \mathsf{sum}.1.n &= +/.(1\ \mathsf{upto}\ n) \quad . \end{aligned}$$

In this example first a list of integers is constructed, and in the next step this list is consumed by the catamorphism $+/$. If one would translate this algorithm directly into an imperative program this would give rise to two loops: one for constructing the list, and one for iterating over the constructed list. It would be hard to imagine someone really programming it like this, because every programmer would immediately merge the two loops into one, and not construct the list at all. This shows a remarkable difference between the two styles of programming. In the functional style one would use a number of higher order constructs, in this case function composition, to construct an algorithm out of existing components, whereas the imperative programmer would write this program from scratch. The question which now arises is whether it is possible to keep the elegance of the functional style, while achieving the efficiency and compactness of the imperative style.

When a catamorphism is applied to a constructed data structure the optimisation is straightforward: *substitute the operators of the catamorphism directly into the data structure building algorithm, at those places where the operators of the initial algebra* *of its domain contribute in constructing the result.* As a result it is not the input for the catamorphism which is constructed, but the result of applying the catamorphism to this input. Since this transformation can always be performed, and often there is not a specific point in the derivation at which this transformation is done at best, one is free to postpone this optimisation until the algorithm derivation has been completed.

We will call data structures which have played a catalytic rôle in the derivation of the algorithm, but have disappeared from the final algorithm *virtual data structures*.

This approach is again well known in the compiler construction world, and corresponds to the use of an attribute grammar which has synthesised attributes only. Here the optimisation may be used in which not the parse tree as such is constructed by the parser, but the computations which were to be performed on the parse tree, are performed by the parser directly. In a simpler form this transformation is also known as vertical loop-fusion to imperative programmers.

In [14] this process has been dubbed *deforestation*, indicating how the intermediate tree constructions and reductions have disappeared from the final execution trace.

5 Derivations

In this section we will provide a number of derivations of segment based problems using the views and choices introduced before. As a first example we will present the problem of computing the maximum segment sum [5]. This derivation shows how, using two different segment constructors, two completely different derivations may be given. Our first derivation is heavily influenced by [2]. The second problem we will treat is the computation of the longest low segment. This problem illustrates the advantage of choosing the right algebraic view on lists and has been surprisingly hard to solve when attention is confined to the snoc and cons view.

Before delving into the details of these algorithms we will introduce some laws which will be used over and over again, and thus deserve a separate introduction. Those interested in the proofs of these indentities may consult [1][11].

$$\begin{array}{ll} \textbf{map-distributivity} & f^* \circ g^* \quad = (f \circ g)^* \\ \textbf{map-promotion} & f^* \circ +\!\!+/ = +\!\!+/ \circ f^{**} \\ \textbf{reduce-promotion} & \otimes/ \circ +\!\!+/ = \otimes/ \circ (\otimes/)^* \quad . \end{array}$$

This last law is an instance of the so-called *promotion* law. The promotion law says that

$$h \circ \oplus/ = \odot/ \circ h^*$$

if

$$h.(a \oplus b) = h.a \odot h.b \text{ and } h.1_\oplus = 1_\odot \quad .$$

5.1 Maximum Segment Sum

The specification for the computation of the maximum segment sum reads as follows:

$$\uparrow/ \circ +/^* \circ \mathsf{segs}$$

and is an instantiation of the problem class given in (1).

Inits/Tails Decomposition. Recall the generic specification (1). The $\mathsf{property}$ in the maximum segment sum to be computed for every segment is a catamorphism. Therefore we start with deriving some properties of such catamorphisms, generalising the $+/$ to $\oplus/$, and the $\mathsf{compose}$ operator $\uparrow$ to $\otimes$. This will be done assuming that SV is the chosen view, and $\mathsf{segs} = +\!\!+/ \circ \mathsf{tails}^* \circ \mathsf{inits}$ the chosen generator, leading to:

$$\mathsf{mss} = \otimes/ \circ \oplus/^* \circ +\!\!+/ \circ \mathsf{tails}^* \circ \mathsf{inits} \quad .$$

We start by moving the $+\!\!+/$ part forward:

$$
\begin{array}{l}
\otimes/ \circ \oplus/^* \circ +\!\!+/ \circ \mathsf{tails}^* \circ \mathsf{inits} \\
\quad = \text{map-promotion} = \\
\otimes/ \circ +\!\!+/ \circ \oplus/^{**} \circ \mathsf{tails}^* \circ \mathsf{inits} \\
\quad = \text{reduce-promotion} = \\
\otimes/ \circ \otimes/^* \circ \oplus/^{**} \circ \mathsf{tails}^* \circ \mathsf{inits} \\
\quad = \text{map-distributivity} = \\
\otimes/ \circ (\otimes/ \circ \oplus/^* \circ \mathsf{tails})^* \circ \mathsf{inits} \\
\quad = \textbf{define } \mathsf{F} = \otimes/ \circ \oplus/^* \circ \mathsf{tails} = \\
\otimes/ \circ \mathsf{F}^* \circ \mathsf{inits}
\end{array}
$$

Using point-wise reasoning we now derive some properties of F. In this calculation we shall accumulate a number of desirable properties of $\otimes$ and $\oplus$. These properties become the applicability conditions of one of our theorems. Our first assumption is that $\otimes$ and $\oplus$ have unit elements $1_\otimes$ and $1_\oplus$ respectively.

$$
\begin{array}{l}
\mathsf{F}.(x \mathbin{+\!\!<} a) \\
\quad = \text{definition of } \mathsf{F} = \\
\otimes/ \circ \oplus/^* \circ \mathsf{tails}.(x \mathbin{+\!\!<} a) \\
\quad = \text{definition of } \mathsf{tails} = \\
\otimes/ \circ \oplus/^*.(((\mathbin{+\!\!<} a)^* \circ \mathsf{tails}.x) \mathbin{+\!\!<} \Box) \\
\quad = \text{definition of map, reduce on } \Box = \\
\otimes/.((\oplus/^* \circ (\mathbin{+\!\!<} a)^* \circ \mathsf{tails}.x) \mathbin{+\!\!<} 1_\oplus) \\
\quad = \text{map distributivity} = \\
\otimes/.(((\oplus/ \circ (\mathbin{+\!\!<} a))^* \circ \mathsf{tails}.x) \mathbin{+\!\!<} 1_\oplus) \\
\quad = \text{definition of reduction} = \\
\otimes/.((((\oplus a) \circ \oplus/)^* \circ \mathsf{tails}.x) \mathbin{+\!\!<} 1_\oplus) \\
\quad = \text{map distributivity} = \\
\otimes/.(((\oplus a)^* \circ \oplus/^* \circ \mathsf{tails}.x) \mathbin{+\!\!<} 1_\oplus) \\
\quad = \text{definition of reduction} = \\
(\otimes/ \circ (\oplus a)^* \circ \oplus/^* \circ \mathsf{tails}.x) \otimes 1_\oplus \\
\quad = \textbf{assume } (\oplus a) \text{ factors out of } \otimes = \\
((\oplus a) \circ \otimes/ \circ \oplus/^* \circ \mathsf{tails}.x) \otimes 1_\oplus \\
\quad = \text{folding the definition of } \mathsf{F} = \\
((\oplus a) \circ \mathsf{F}.x) \otimes 1_\oplus
\end{array}
$$

As similar derivation may be given for the empty case:

$$\begin{array}{l} \mathsf{F}.\square \\ \quad = \text{definition of } \mathsf{F} = \\ \otimes/ \circ \oplus/^* \circ \mathsf{tails}.\square \\ \quad = \text{definition of tails} = \\ \otimes/ \circ \oplus/^*.(\square \prec\!\!\!+ \square) \\ \quad = \text{definition of map} = \\ \otimes/.(\square \prec\!\!\!+ 1_\oplus) \\ \quad = \text{definition of reduction} = \\ 1_\oplus \otimes 1_\otimes \\ \quad = \text{definition of } 1_\otimes = \\ 1_\oplus \end{array}$$

Based on these derivations we may now conclude that, using the binary operator $x \odot a \stackrel{\text{def}}{=} (x \oplus a) \otimes 1_\oplus$, F may be written as a catamorphism $(\!|1_\oplus, \odot|\!)$. Note that this derivation would not have been possible when using the first definition of tails, since this definition is not a homomorphism (it is a *paramorphism*, [8]).

Completing the derivation of the segment decomposition schema is now straightforward by using (2). Using these results we have derived the following schema for segment problems:

$$\begin{array}{l} \otimes/ \circ \oplus/^* \circ \mathsf{segs} = \otimes/ \circ \odot\!/\!\!/ \\ \qquad\qquad\qquad\qquad \textbf{where } x \odot a = (x \oplus a) \otimes 1_\oplus \\ \qquad\qquad\qquad\qquad \textbf{provided } (b \otimes c) \oplus a = (b \oplus a) \otimes (c \oplus a) \quad . \end{array}$$

By now noticing that indeed it is the fact that $(b \uparrow c) + a = (b + a) \uparrow (c + a)$ we may conclude that:

$$\begin{array}{l} \mathsf{mss} = \uparrow/ \circ \odot\!/\!\!/ \\ \qquad\quad \textbf{where } x \odot a = (x + a) \uparrow 1_+ = (x + a) \uparrow 0. \end{array}$$

Finally we may substitute the operations of $\uparrow/$ in the definition of $/\!\!/$, considering this as a virtual data structure, giving:

$$\begin{array}{l} \mathsf{mss} = \pi_2 \circ (\!|\langle 1_\odot, 0\rangle, \tilde{\odot}|\!) \\ \qquad\quad \textbf{where} \\ \qquad\quad \langle a, b\rangle \tilde{\odot} c = \langle a \odot c, (a \odot c) \uparrow b\rangle \end{array}$$

which corresponds directly to its imperative counterpart, using a single loop and one additional variable:

$$\begin{array}{l} a, b, i := 0, 0, 0 \\ \textbf{do } i \neq n \rightarrow a := (a + x[i]) \uparrow 0; b := b \uparrow a; i := i + 1 \textbf{ od} \end{array}$$

Inits/Inits Decomposition. In this section we will present a different derivation for the maximum segment problem, starting from the alternative definition of tails as given in (8).

We already discussed this generator in Sect. 3.3. Before embarking upon the derivation, it will be expedient to mention two algebraic properties that will be useful in the sequel. We start by noticing:

$$+/.b = +/.((a +\!\!+\, b) \gg a) = (+/.(a +\!\!+\, b) - +/.a)$$

and consequently:

$$\begin{aligned}
& +/^* \circ \overrightarrow{\mathsf{X}}_{\gg} \circ \langle \mathsf{id}, \mathsf{inits} \rangle \\
= {} & \text{relation between } \gg,\ + \text{ and } - = \\
& \overrightarrow{\mathsf{X}}_{-} \circ \langle +/ \circ \mathsf{id}, +/^* \circ \mathsf{inits} \rangle
\end{aligned} \tag{10}$$

A second important observation is:

$$(\otimes/ \circ \oplus/\!\!/) * \circ \mathsf{inits} = \mathsf{tl} \circ \otimes/\!\!/ \circ \oplus/\!\!/ \quad , \tag{11}$$

where $\mathsf{tl}.(a \mathbin{>\!\!\!+} x) = x$. Using these properties we may now derive:

$$\begin{aligned}
& \uparrow/ \circ +/^* \circ \overrightarrow{\mathsf{X}}_{\gg} \circ \langle \mathsf{id}, \mathsf{inits} \rangle \\
& \quad = (10) = \\
& \uparrow/ \circ \overrightarrow{\mathsf{X}}_{-} \circ \langle +/, +/^* \circ \mathsf{inits} \rangle \\
& \quad = \text{definition of accumulate} = \\
& \uparrow/ \circ \overrightarrow{\mathsf{X}}_{-} \circ \langle +/, +/\!\!/ \rangle \\
& \quad = (a - b) \uparrow (a - c) = a - (b \downarrow c) = \\
& (-) \circ \langle +/, \downarrow/ \circ +/\!\!/ \rangle
\end{aligned}$$

We present now the following derivation of mss:

$$\begin{aligned}
& \mathsf{mss} \\
& \quad = \mathsf{inits/inits} \text{ generator, promotion} = \\
& \uparrow/ \circ (\uparrow/ \circ +/^* \circ \overrightarrow{\mathsf{X}}_{\gg} \langle \mathsf{id}, \mathsf{inits} \rangle)^* \circ \mathsf{inits} \\
& \quad = \text{above derivation} = \\
& \uparrow/ \circ ((-) \circ \langle +/, \downarrow/ \circ +/\!\!/ \rangle)^* \circ \mathsf{inits} \\
& \quad = \text{map distributivity} = \\
& \uparrow/ \circ (-)^* \circ \langle +/, \downarrow/ \circ +/\!\!/ \rangle^* \circ \mathsf{inits} \\
& \quad = \langle f, g \rangle^* = \mathsf{zip} \circ \langle f^*, g^* \rangle = \\
& \uparrow/ \circ (-)^* \circ \mathsf{zip} \circ \langle +/^* \circ \mathsf{inits}, (\downarrow/ \circ +/\!\!/)^* \circ \mathsf{inits} \rangle \\
& \quad = \text{definition of accumulate, property (11)} = \\
& \uparrow/ \circ (-)^* \circ \mathsf{zip} \circ \langle +/\!\!/, \mathsf{tl} \circ \downarrow/\!\!/ \circ +/\!\!/ \rangle \\
& \quad = \text{product} = \\
& \uparrow/ \circ (-)^* \circ \mathsf{zip} \circ \langle \mathsf{id}, \mathsf{tl} \circ \downarrow/\!\!/ \rangle \circ +/\!\!/
\end{aligned}$$

By making the intermediate data structures virtual again, this directly corresponds to the following imperative program:

$$s, m, r := 0, \infty, -\infty \quad \{\text{unit elements for } +, \downarrow, \uparrow\}$$
$$r, i := 0, 0 \quad \{\text{initialise loop and compute } \mathsf{tl}\}$$
$$\mathbf{do}\ i \neq n \rightarrow s := s + x[i]; m := m \downarrow s; r := r \uparrow (s - m); i := i + 1\ \mathbf{od}$$

In this program we maintain in s the sum of the elements seen thus far, in m the minimum value seen thus far, and in r the greatest difference between these values seen thus far. Since one is only interested in this difference the previous program can be seen as an optimisation of this program with an extra invariant $a = (s - m)$. An interesting aspect of this program schema is that the accumulation of the input data is explicitly maintained. If one tries to solve the computation of the *longest balanced segment* this generator is a useful starting point.

This algorithm resembles the Wall-street approach, where the list of numbers indicates the daily changes in the Dow-Jones index, and the question to be answered is what would have been the best time for a one-time investment.

5.2 Length of a Longest Low Segment

In this paragraph we will derive a solution for the problem of computing the *length of a longest low segment.* A very similar problem is the computation of a *largest rectangle under a histogram*, the solution and derivation of which may be found in [12], and which may be compared with the derivation given here.

The problem may be stated as follows:

$$\mathsf{llls} = \uparrow/ \circ \#^* \circ \mathsf{low} \triangleleft \circ \mathsf{segs}$$
$$\mathbf{where}\ \mathsf{low} = (\leq) \circ \langle \uparrow/, \# \rangle$$

with $(a \uparrow_f b) = (f.a) \uparrow (f.b)$.

We start by noticing first that the predicate used in the filter is monotonic in the second component:

$$\uparrow/ \circ (a \leq) \triangleleft = (a \leq)? \circ \uparrow/ \tag{12}$$

where the ? is a one-point filter that either returns its argument, or returns minus infinity. Let v_a denote a set of segments with a common maximum element a. We may reason as follows:

$$\begin{array}{l} \uparrow/ \circ \#^* \circ \mathsf{low} \triangleleft . v_a \\ \quad = \text{expand definition of } \mathsf{low} = \\ \uparrow/ \circ \#^* \circ ((\leq) \circ \langle \uparrow/, \# \rangle) \triangleleft . v_a \\ \quad = \text{maximum of elements of } v_a \text{ equals } a = \\ \uparrow/ \circ \#^* \circ ((a \leq) \circ \#) \triangleleft . v_a \\ \quad = \text{move }\ ^* \text{ through } \triangleleft = \\ \uparrow/ \circ (a \leq) \circ \#^* . v_a \\ \quad = (12) = \\ (a \leq)? \circ \uparrow/ \circ \#^* . v_a \end{array} \tag{13}$$

This suggests that, when summing up the segments by common maximum element, we might be very efficient in skipping some of these elements in the generating process, and thereby getting a more efficient algorithm. We thus will try to push the filter into the generating process. Since we want to sum up the elements by common element we choose the LTV, with the generator given in (9).

Since it furthermore is the case that $x \in \mathsf{segs}.y \Rightarrow \uparrow/.x \leq \uparrow/.y$, it is profitable to first sum up all the segments with a common maximum value, and then the rest of the segments grouped according to their maximum value.

We will now make use of our freedom in choosing an element in LTV which makes this an easy task. Since the generator introduced in (9) sums up segments by common element, it is now sufficient if we have an element in the equivalence class generated by the associative law, for which it is the case that a label value in the tree is at least the maximum of the label values in the subtrees, *i.e.* it is a heap.

$$
\begin{array}{l}
\mathsf{llls}.(x \prec a \succ y) \\
\quad = \text{definition } \mathsf{llls} = \\
\uparrow/ \circ \#^* \circ \mathsf{low} \triangleleft \circ \mathsf{segs}.(x \prec a \succ y) \\
\quad = \text{unfold definition } (9) = \\
\uparrow_{\#}/ \circ \#^* \circ \mathsf{low} \triangleleft (\mathsf{segs}.x +\!\!+ \mathsf{segs}.y +\!\!+ (a\odot)^*.(\mathsf{tails}.x \mathbin{\mathsf{X}} \mathsf{inits}.y)) \\
\quad = \text{filter and reduce promotion, folding } \mathsf{llls} = \\
(\mathsf{llls}.x) \uparrow (\mathsf{llls}.y) \uparrow (\uparrow/ \circ \#^* \circ \mathsf{low} \triangleleft \circ (a\odot)^*.(\mathsf{tails}.x \mathbin{\mathsf{X}} \mathsf{inits}.y))
\end{array}
$$

Although this already looks like a catamorphism it isn't one as yet. We now concentrate on the last part, which may be rewritten, assuming that the predicate

$$a = \uparrow/.(x +\!\!+ [a] +\!\!+ y) \tag{14}$$

holds:

$$
\begin{array}{l}
(\uparrow/ \circ \#^* \circ \mathsf{low} \triangleleft \circ (a\odot)^* \circ (\mathsf{tails}.x \mathbin{\mathsf{X}} \mathsf{inits}.y) \\
\quad = (13) = \\
(a \leq)? \circ \uparrow/ \circ \#^* \circ (a\odot)^* \circ (\mathsf{tails}.x \mathbin{\mathsf{X}} \mathsf{inits}.y) \\
\quad = \text{map distributivity, } (\# \circ (a\odot) = (1+) \circ \# \circ +\!\!+) = \\
(a \leq)? \circ \uparrow/ \circ (1+)^* \circ (\# \circ +\!\!+)^* \circ (\mathsf{tails}.x \mathbin{\mathsf{X}} \mathsf{inits}.y) \\
\quad = \text{promotion: } (1 + a) \uparrow (1 + b) = 1 + (a \uparrow b) = \\
(a \leq)? \circ (1+) \circ \uparrow/ \circ (\# \circ +\!\!+)^*.(\mathsf{tails}.x \mathbin{\mathsf{X}} \mathsf{inits}.y) \\
\quad = \text{length is a homomorphism} = \\
(a \leq)? \circ (1+) \circ \uparrow/ \circ (+ \circ (\# \times \#))^*.(\mathsf{tails}.x \mathbin{\mathsf{X}} \mathsf{inits}.y) \\
\quad = \text{map distributivity, map over cross} = \\
(a \leq)? \circ (1+) \circ \uparrow/ \circ (+)^*.(\#^* \circ \mathsf{tails}.x \mathbin{\mathsf{X}} \#^* \circ \mathsf{inits}.y) \\
\quad = \text{addition is monotonic} = \\
(a \leq)? \circ (1+) \circ (+).\langle \uparrow/ \circ \#^* \circ \mathsf{tails}.x, \uparrow/ \circ \#^* \circ \mathsf{inits}.y \rangle \\
\quad = \text{longest tail of } x \text{ is } x \text{, idem for inits} = \\
(a \leq)? \circ (1+) \circ (+).\langle \#.x, \#.y \rangle \\
\quad = \text{notation} = \\
(a \leq)?(1 + x + y)
\end{array}
$$

Tupling this expression with the computations of $\#.x$ and $\#.y$ leads now to the following catamorphism on a suitably chosen element in the LTV:

$$\mathsf{llls} = \pi_1 \circ (\!|\langle 0,0 \rangle, f|\!) \circ \mathsf{SVintoLTV} \tag{15}$$

$$\textbf{where } f.\langle \langle m,x \rangle, a, \langle n,y \rangle \rangle = \langle m \uparrow n \uparrow ((a \leq)?(x+1+y)), (x+1+y) \rangle \tag{16}$$

The only problem which now remains open is to find the suitable representation for the input in LTV-form; this is easily solved by using the well-known algorithm for the construction of a heap. The algorithm used corresponds directly to precedence parsing. The only difference is that all operators are equal to the empty tree, and the priority of the operators is the reverse of what one finds usual. We thus use SPV as an intermediate representation: $\mathsf{svintoltv} = \mathsf{SPVintoLTV} \circ \mathsf{SVintoSPV}$.

The transformation of SV into SPV is a straightforward application of parsing, where the values in the LTV-parts are always at most their associated value in the pair in the spine:

$$\begin{aligned}\mathsf{SVintoSPV} &= (\!|\Box, (\mathsf{id}\hat{\ominus}\langle\Box, -\rangle)|\!) \\ &\quad \textbf{where } \; (s \prec \langle t, v\rangle) \ominus (u, w) = \begin{array}[t]{l} v \geq w \rightarrow (s \prec \langle t, v\rangle) \prec \langle u, w\rangle \\ v \leq w \rightarrow s \ominus \langle (t \prec v \succ u), w\rangle \end{array} \\ &\quad\qquad \Box \ominus \langle u, w\rangle = \Box \prec \langle u, w\rangle \quad .\end{aligned} \tag{17}$$

The function converting an SPV-value into an LTV-value (maintaining the heap property) is given by

$$\mathsf{SPVintoLTV}.s = \pi_1 \circ \pi_2 \circ \prec^{-1} \circ (s \ominus \langle\Box, \infty\rangle) \quad . \tag{18}$$

So we are finished, and gathering the intermediate results in (15), (17) and (18), we get

$$\mathsf{llls} = \pi_1 \circ (\!|\langle 0, 0\rangle, f|\!) \circ \pi_1 \circ \pi_2 \circ \prec^{-1} \circ (\ominus\langle\Box, \infty\rangle) \circ (\!|\Box, (\mathsf{id}\hat{\ominus}\langle\Box, -\rangle)|\!) \quad .$$

To this result again the virtual data structure optimisation may be applied, effectively preventing the labelled tree coming into existence at all. In the final result only a parse stack is maintained, containing the results and lengths of already processed subtrees, and label values which have not found their corresponding right subtree parts.

The final algorithm now becomes:

$$\begin{aligned}&\mathsf{llls} = \\ &\pi_1 \circ (\oslash\langle\langle 0, 0\rangle, \infty\rangle) \circ (\!|\Box, (\mathsf{id}\hat{\oslash}\langle\langle 0, 0\rangle, -\rangle)|\!) \\ &\textbf{where } \; s \prec \langle\langle m, x\rangle, v\rangle \oslash \langle\langle n, y\rangle, w\rangle = \begin{array}[t]{l} v \geq w \rightarrow (s \prec \langle\langle m, x\rangle, v\rangle) \prec \langle\langle n, y\rangle, w\rangle \\ v \leq w \rightarrow s \oslash \langle\langle m \uparrow n \uparrow ((v \leq)?(xy1)), xy1\rangle, w\rangle \\ \textbf{where } xy1 = x + y + 1 \end{array} \\ &\qquad \Box \oslash \langle\langle n, y\rangle, w\rangle = \Box \prec \langle\langle n, y\rangle, w\rangle \quad .\end{aligned}$$

6 Acknowledgements

The authors wish to thank Lambert Meertens, Richard Bird, and the participants and lecturers of the Ameland meetings for working together in a cooperative way in developing the calculus employed in this paper. This research was supported by the Dutch Organization for Scientific Research, grant NFI 62-518.

References

1. R.C. Backhouse. An exploration of the Bird–Meertens formalism. Computing Science Note CS 8810, Department of Computing Science, Groningen University, P.O. Box 800, 9700 AV Groningen, The Netherlands, 1988.
2. R.S. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, volume 36 of *NATO ASI Series F*, pages 3–42. Springer–Verlag, 1987.
3. R.S. Bird and P. Wadler. *Introduction to Functional Programming.* Prentice–Hall, 1988.
4. Hendrik Boom. private communication, 1979. IFIP Working Group 2.1, Jablonna, Warschau.
5. E.W. Dijkstra. *A Discipline of Programming.* Prentice–Hall, 1976.
6. David Gries. *The Science of Programming.* Springer, 1983. 366p.
7. L. Meertens. Algorithmics — towards programming as a mathematical activity. In J.W. de Bakker, M. Hazewinkel, and J.K. Lenstra, editors, *Mathematics and Computer Science*, volume 1 of *CWI Monographs*, pages 3–42. North–Holland, 1987.
8. L. Meertens. Paramorphisms. Technical Report CS-R9005, CWI, Amsterdam 1990. To appear in Formal Aspects of Computing.
9. Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *FPCA*, LNCS. Springer-Verlag, 1991.
10. C.C. Morgan. *Programming from Specifications.* Prentice–Hall, 1990.
11. M. Spivey. A categorical approach to the theory of lists. In J.L.A. van de Snepscheut, editor, *Mathematics of Program Construction*, volume 375 of *Lecture Notes in Computer Science*, pages 399–408. Springer–Verlag, 1989.
12. J.C.S.P. van der Woude. Rabbitcount := rabbitcount - 1. In J.L.A. van de Snepscheut, editor, *Mathematics of Program Construction*, volume 375 of *Lecture Notes in Computer Science*, pages 409–420. Springer–Verlag, 1989.
13. Philip Wadler. Views: a way for pattern matching to cohabit with data abstraction. In *Proc. 14th Symposium of Principles of Programming Languages*, pages 307–313. ACM, January 1987.
14. Philip Wadler. Deforestation: transforming programs to eliminate trees. In H. Ganzinger, editor, *ESOP '88 (= Proc. 2nd European Symposium on Programming*, volume 300 of *LNCS*, pages 345–358. Springer-Verlag, March 1988.

This article was processed using the LaTeX macro package with LMAMULT style