J. Functional Programming 6 (1): 1–28, January 1996 © 1996 Cambridge University Press 

# *Generic functional programming with types and relations*

RICHARD BIRD, OEGE DE MOOR
*Programming Research Group, Oxford University, Wolfson Building, Parks Road, Oxford OX1 3QD, UK*

AND PAUL HOOGENDIJK
*Eindhoven University of Technology, PO Box 513, 5600 MB Eindhoven, The Netherlands*

**Abstract**

A *generic* functional program is one which is parameterised by datatype. By installing specific choices, for example lists or trees, different programs are obtained that are, nevertheless, abstractly the same. The purpose of this paper is to explore the possibility of deriving generic programs. Part of the theory of lists that deals with segments is recast as a theory about 'segments' in a wide class of datatypes, and then used to pose and solve a generic version of a well-known problem.

**Capsule Review**

There are many programming problems that can be specified on a variety of datatypes such as lists, trees and forests. Examples of such problems are parsing, pretty printing, unification, etc. This paper investigates how datatype independent programs for problems such as these can be constructed.

The main problem solved in the paper is a generalisation of the maximum segment sum problem on the datatype of lists to a large class of datatypes. Bird's paper 'An introduction to the theory of lists' from 1987 contains an elegant derivation of a linear-time program for the maximum segment sum problem on lists. In this paper, the authors show that this derivation can be generalised in a relatively straightforward way to a large class of datatypes. The main ingredients of the generalisation are functors, describing the structure of datatypes, and relations, simplifying the derivation. The constituents of the derivation are general combinators that are not only applicable to the maximum segment sum problem, but also to problems like deforestation and pattern matching.

## 1 Introduction

To what extent is it possible to construct useful programs without knowing exactly what datatypes are involved? At first sight this may seem a strange question, but take the example of pattern matching. Over lists this problem can be formulated in terms of two strings, a pattern and a text; the object is to determine if and where the pattern occurs as a segment of the text. Now, pattern matching can be generalised to other datatypes, including arrays and trees of various kinds; the essential step is to be able to define the notion of segment in these types. So the intriguing question arises: can one construct a useful algorithm, parameterised by a datatype, to solve the general problem of pattern matching?

The purpose of this paper is to explore the ideas of generic programming through a second problem, related to but simpler than pattern matching, namely the problem of computing the maximum segment sum. This problem was chosen because sufficient list theory already exists (Bird, 1987, 1989a, 1990) for one to calculate an efficient solution in a few equational steps. It turns out that the theory of segments can be generalised to a wide class of datatypes, so the calculation leads to a generic solution to the problem.

To be able to construct a generic theory of segments, we need a reformulation of the theory of lists with two new ingredients. The first ingredient is a categorical treatment of datatypes (Malcolm, 1990; Manes and Arbib, 1986; Lehmann and Smyth, 1981). In the categorical approach, datatypes are characterised in terms of certain mappings, called *functors*, and specifications can be parameterised by functors in a simple and direct manner. Although the categorical approach is becoming familiar to functional programmers (Barr and Wells, 1990; Bird and de Moor, 1996; Pierce, 1991), we will give a brief but hopefully adequate account of the essential ideas.

The second ingredient involves the move from functions to relations (Aarts *et al.*, 1992; De Moor, 1992). Introducing relations enables us to deal more smoothly with nondeterministic specifications, but it also turns out that the calculus of relations leads to substantial simplifications both in the derivation of purely functional programs and in the study of general datatypes. Again, we will give a light account of the ideas.

The rest of the paper is structured as follows. In the next section we show – quite informally – how the notion of segment can be defined in one or two other data types. After that, in section 3, we examine the structure of the derivation of the maximum segment sum problem, recalling that it depends on two results in the theory of lists, namely Horner's rule and the Scan lemma. Section 4 gives an account of the general theory of datatypes, and in section 5 we review part of the calculus of relations. Using this theory, we show in section 6 how Horner's rule can be generalised. To give an account of a general version of the Scan lemma, we need the fact, explained in section 7, that every datatype found in functional programming comes equipped with a *membership* relation. It turns out that the existence of a membership relation for a datatype is crucial for generic programming. In section 8 we generalise the Scan lemma, and in section 9 present the complete derivation of the maximum segment sum problem. Finally, Section 10 contains a discussion of the implications of this research.

## 2 Towards generality

Let us start by being more precise about what we mean by a segment of a list, indeed, what we mean by a list. There are two basic views of lists, one of which is given by the type declaration

$$listl\ A \quad ::= \quad nil \mid snoc\,(listl\ A, A).$$

Formally, this means that lists are represented as finite terms over *nil* and *snoc*. For instance, the list $[1,2,3]$ is represented by the term

$$snoc\,(snoc\,(snoc\,(nil,1),2),3).$$

Thinking of lists as terms, we see that a prefix of $x$ is really the same thing as a *subterm* of $x$. The function *subterms* takes a list and returns the set of all its subterms:

$$\begin{aligned} subterms\ nil &= \{nil\} \\ subterms\,(snoc\,(x,a)) &= subterms\ x \cup \{snoc\,(x,a)\}. \end{aligned}$$

In the theory of lists, prefixes are called *initial segments*, and the function *subterms* is called *inits*. There is the subtle difference that *inits* returns a list rather than a set, but we ignore this distinction for now, though we return to it in the next section, and it will prove to be of crucial importance later on.

Dual to the notion of prefix is that of a *suffix*. A suffix of $x$ can be obtained by substituting the empty list for a subterm of $x$. For instance, $snoc\,(snoc\,(nil,2),3)$ is obtained from the term above by replacing the subterm $snoc\,(nil,1)$ by the empty list *nil*. For the sake of a word we can say that this subterm is the result of *pruning* the original term. The function *prunings* takes a list and returns all ways in which it can be pruned:

$$\begin{aligned} prunings\ nil &= \{nil\} \\ prunings\,(snoc\,(x,a)) &= \{nil\} \cup \{snoc\,(y,a) \mid y \in prunings\ x\}. \end{aligned}$$

In the theory of lists, suffixes are called *tail* segments and *prunings* is called *tails*.

One can now define arbitrary segments by the equation

$$segments \quad = \quad union \cdot map\ prunings \cdot subterms.$$

Here *union* is the function that takes a collection of sets and returns its union, and *map* is the operator that applies a function to all elements of a set.

For comparison, consider now the other – and more familiar – view of lists, given by the type declaration

$$listr\ A \quad ::= \quad nil \mid cons\,(A, listr\ A).$$

With this datatype the role of *inits* and *tails* are reversed: *subterms* gives the tail segments of a list, while *prunings* gives the initial segments. The function *segments* is defined in the same way as before and again gives the segments of a list.

As a third example, consider binary trees as defined by

$$tree\ A \quad ::= \quad nil \mid fork\,(A, tree\ A, tree\ A).$$

The elements of this type are finite trees, this time over *nil* and *fork*, so it is again possible to define the functions *subterms* and *prunings*. The function *subterms* takes a binary tree and returns the set of all its subtrees:

$$\begin{aligned} \mathit{subterms}\ \mathit{nil} &= \{\mathit{nil}\} \\ \mathit{subterms}\ (\mathit{fork}\ (a,x,y)) &= \mathit{subterms}\ x \cup \mathit{subterms}\ y \cup \{\mathit{fork}\ (a,x,y)\}. \end{aligned}$$

The function *prunings* takes a binary tree and substitutes *nil* for its subtrees in all possible ways:

$$\begin{aligned} \mathit{prunings}\ \mathit{nil} &= \{\mathit{nil}\} \\ \mathit{prunings}\ (\mathit{fork}\ (a,x,y)) &= \{\mathit{nil}\} \cup \\ &\quad \{\mathit{fork}\ (a,u,v) \mid u \in \mathit{prunings}\ x, v \in \mathit{prunings}\ y\}. \end{aligned}$$

The segments of a tree are defined by the same equation as before. Jeuring (1989) also considered such a definition, though he spoke of *treecuts* rather than segments.

## 3 The maximum segment sum

The problem of the maximum segment sum is to compute the function *mss*, where

$$\mathit{mss} = \mathit{max} \cdot \mathit{map}\ \mathit{sum} \cdot \mathit{segments}.$$

Over lists this problem is interpreted as follows: given a list of integers, compute the sum of the elements in each segment of the list and return the maximum such sum. For example, $\mathit{mss}\ [-1,2,-1,3,-2] = 4$ because the segment $[2,-1,3]$ has maximum sum.

Given the definition of *segments* in the previous section, we can calculate

$$\begin{aligned} & \mathit{mss} \\ = \quad & \{\text{definition}\} \\ & \mathit{max} \cdot \mathit{map}\ \mathit{sum} \cdot \mathit{segments} \\ = \quad & \{\text{definition of } \mathit{segments}\} \\ & \mathit{max} \cdot \mathit{map}\ \mathit{sum} \cdot \mathit{union} \cdot \mathit{map}\ \mathit{prunings} \cdot \mathit{subterms} \\ = \quad & \{\text{since } \mathit{map}\ f \cdot \mathit{union} = \mathit{union} \cdot \mathit{map}\ (\mathit{map}\ f)\} \\ & \mathit{max} \cdot \mathit{union} \cdot \mathit{map}\ (\mathit{map}\ \mathit{sum}) \cdot \mathit{map}\ \mathit{prunings} \cdot \mathit{subterms} \\ = \quad & \{\text{since } \mathit{max} \cdot \mathit{union} = \mathit{max} \cdot \mathit{map}\ \mathit{max} \text{ (over sets of non-empty sets)}\} \\ & \mathit{max} \cdot \mathit{map}\ \mathit{max} \cdot \mathit{map}\ (\mathit{map}\ \mathit{sum}) \cdot \mathit{map}\ \mathit{prunings} \cdot \mathit{subterms} \\ = \quad & \{\text{since } \mathit{map}\ f \cdot \mathit{map}\ g = \mathit{map}\ (f \cdot g)\} \\ & \mathit{max} \cdot \mathit{map}\ (\mathit{max} \cdot \mathit{map}\ \mathit{sum} \cdot \mathit{prunings}) \cdot \mathit{subterms}. \end{aligned}$$

In the fourth step we have assumed that *prunings* returns a non-empty set, so that $\mathit{map}\ \mathit{prunings} \cdot \mathit{subterms}$ returns a set of non-empty sets; this is necessary since *max* is not defined on the empty set. So far, the calculation has been completely general (and also standard: we have merely copied from Bird (1989) modulo some changes in names), but now let us revert to the specific datatype *listl A*. In this datatype the function *sum* is defined by

$$\mathit{sum} = \mathit{foldl}\ (0,+),$$

where *foldl* $(c,f)$ is defined by

$$\begin{aligned} foldl\,(c,f)\,nil &= c \\ foldl\,(c,f)\,(snoc\,(x,a)) &= f\,(foldl\,(c,f)\,x,a). \end{aligned}$$

In standard functional programming, *foldl* $(c,f)$ is usually written in the form *foldl* $f\ c$ and is given as a function over the type *listr* $A$, since lists in functional programming are built with *nil* and *cons* rather than *nil* and *snoc*, but the above is an equivalent definition.

Now there is a standard result in the theory of lists, called *Horner's rule*, which says that if $f$ is monotonic with respect to $\leq$, then

$$max \cdot map\,(foldl\,(c,f)) \cdot tails \quad = \quad foldl\,(c,g),$$

where $g\,(a,b) = f\,(a,b) \sqcup c$ and $a \sqcup b$ is the greater of $a$ and $b$. We therefore obtain

$$max \cdot map\ sum \cdot tails \quad = \quad foldl\,(0,\oplus),$$

where $a \oplus b = (a+b) \sqcup 0$.

Using this result to continue the calculation of *mss*, again in the specific context of lists, we find

$$mss \quad = \quad max \cdot map\,(foldl\,(0,\oplus)) \cdot inits,$$

where we have replaced *subterms* by *inits*. The final step is to make use of an important operation on lists called *scanl*, whose definition we will see in a moment. Scans are important in functional programming; in particular, Gibbons (1991) has made a study of scans on a particular species of binary tree. Over lists, the key fact is the *Scan lemma*, which says

$$map\,(foldl\,(c,f)) \cdot inits \quad = \quad scanl\,(c,f).$$

In this equation *inits* returns a list rather than a set, and *map* is a function on lists. In fact, *inits* returns the list of initial segments of a list in ascending order of length:

$$inits\,[a_1,a_2,\ldots,a_n] \quad = \quad [[\,],[a_1],[a_1,a_2],[a_1,a_2,a_3],\ldots,[a_1,a_2,\ldots,a_n]].$$

As a function returning lists, *inits* is defined by

$$\begin{aligned} inits &= foldl\,([nil],f) \\ &\quad \text{where } f\,(x,a) = snoc\,(x, snoc\,(last\ x,a)). \end{aligned}$$

The function *scanl* $(c,f)$ is defined similarly:

$$\begin{aligned} scanl\,(c,f) &= foldl\,([c],g) \\ &\quad \text{where } g\,(x,a) = snoc\,(x, f\,(last\ x,a)). \end{aligned}$$

Note that *scanl* $(nil, snoc) =$ *inits*, so the definition of *inits* is a special case of *scanl*. The point of the scan lemma is that evaluation of

$$scanl\,(c,\oplus)\,[a_1,a_2,\ldots,a_n] \quad = \quad [c, c\oplus a_1, (c\oplus a_1)\oplus a_2, \cdots ((c\oplus a_1)\oplus\cdots)\oplus a_n]$$

can be done with $n$ evaluations of $\oplus$, whereas direct evaluation of *map* (*foldl* $(c,\oplus)$) $\cdot$ *inits* requires $O(n^2)$ evaluations of $\oplus$ on a list of length $n$.

Applying the scan lemma to the problem of computing *mss*, we end up with the result that

$$\begin{aligned} mss &= max \cdot scanl\,(0, \oplus) \\ a \oplus b &= (a + b) \sqcup 0, \end{aligned}$$

where *max* is now interpreted as a function on lists rather than sets. The above identity leads at once to a linear time algorithm.

The rest of the paper is about how to generalise this calculation to arbitrary datatypes of the kind found in functional programming. In effect, we have to state and prove versions of Horner's rule and the Scan lemma that are valid for any datatype.

## 4 Datatypes

In what follows it is important to emphasise that, unless otherwise stated, a function means a *total* function whose source and target types are sets, unlike in standard functional programming where types are complete partial orders. As a departure from tradition, we reverse the usual order of writing the source and target types in function type declarations, preferring $f : A \leftarrow B$ rather than $f : B \rightarrow A$. This notation is consistent with adjectival order in English and has the advantage that the definition of function composition now takes the smooth form: if $f : A \leftarrow B$ and $g : B \leftarrow C$, then $f \cdot g : A \leftarrow C$.

We have already seen three examples of datatype declarations, namely those of *listl A*, *listr A*, and *tree A*. For example, let us recall

$$listr\ A \quad ::= \quad nil \mid cons\,(A, listr\ A).$$

Whenever one declares a datatype a number of functions are brought into play. In part, declaring a datatype as an equation asserts the existence of an isomorphism between the types on the left and right. In the case of *listr A* this isomorphism takes the form

$$listr\ A \quad \cong \quad 1 + (A \times listr\ A).$$

The type 1 consists of just one member and serves as the source type for constants. It has the property that for each set $A$ there is precisely one function with type $1 \leftarrow A$, the usual notation for this function being $!_A$.

The type constructor $\times$ is cartesian product, and $+$ is disjoint sum, also called *coproduct*. The right-hand side of the isomorphism can be rewritten, giving

$$listr\ A \quad \cong \quad \mathsf{F}(A, listr\ A),$$

where $\mathsf{F}(A, B) = 1 + (A \times B)$ is a mapping from types to types. We can also use $\mathsf{F}$ as a mapping from functions to functions by defining

$$\mathsf{F}(f, g) \quad = \quad id_1 + (f \times g).$$

The function $id_1 : 1 \leftarrow 1$ is the identity function on 1. The cartesian product constructor $\times$ is defined as a mapping between functions in the following way: if $f : A \leftarrow C$ and $g : B \leftarrow D$, then $f \times g : A \times B \leftarrow C \times D$ satisfies

$$(f \times g)(c,d) \quad = \quad (f\,c, g\,d).$$

Similarly, the coproduct constructor $+$ can be defined on functions: applied to a left component $c$, the function $f + g : A + B \leftarrow C + D$ returns $f\,c$ as a left component of the result; dually, applied to a right component $d$, the value of $(f + g)\,d$ is the right component $g\,d$.

A function having a dual role both as a mapping between types and a mapping between functions is, provided certain properties are satisfied, called a *functor*. The functor $\mathsf{F}$ defined above takes a pair of types or functions as argument and so is sometimes called a *bifunctor*. One property we require of a functor $\mathsf{F}$ is that if $f : A \leftarrow B$, then $\mathsf{F}f : \mathsf{F}A \leftarrow \mathsf{F}B$. The other properties are the identity and composition rules:

$$\begin{aligned} \mathsf{F}id_A &= id_{\mathsf{F}A} \\ \mathsf{F}(f \cdot g) &= \mathsf{F}f \cdot \mathsf{F}g. \end{aligned}$$

The function $id_A$ is the identity function with type $A \leftarrow A$. From now on we will usually omit the subscript on $id$, relying on context to resolve ambiguity.

In the case of bifunctors the above rules give, firstly, that if $f : A \leftarrow C$ and $g : B \leftarrow D$, then $\mathsf{F}(f,g) : \mathsf{F}(A,B) \leftarrow \mathsf{F}(C,D)$; and, secondly, that

$$\begin{aligned} \mathsf{F}(id, id) &= id \\ \mathsf{F}(f \cdot g, h \cdot k) &= \mathsf{F}(f,h) \cdot \mathsf{F}(g,k). \end{aligned}$$

In particular, $\times$ and $+$ satisfy the identity and composition rules for bifunctors.

The functor $\mathsf{F}$ associated with the declaration of a datatype is called the *base* functor of the declaration. Thus, $\mathsf{F}(A,B) = 1 + (A \times B)$ is the base functor associated with *listr* $A$. A functor is called *polynomial* if it is built up from constants, finite products and coproducts. More precisely,the class of polynomial functors is defined inductively by the following clauses:

1. The identity functor $id$, defined by $id\,A = A$ and $id\,f = f$, and the constant functors $\mathsf{K}_A$, defined by $\mathsf{K}_A\,B = A$ and $\mathsf{K}_A f = id_A$, are polynomial.
2. If $\mathsf{F}$ and $\mathsf{G}$ are polynomial, then so is their composition $\mathsf{F} \cdot \mathsf{G}$, their sum $\mathsf{F} + \mathsf{G}$ and their product $\mathsf{F} \times \mathsf{G}$, where

$$\begin{aligned} (\mathsf{F} + \mathsf{G})f &= \mathsf{F}f + \mathsf{G}f \\ (\mathsf{F} \times \mathsf{G})f &= \mathsf{F}f \times \mathsf{G}f. \end{aligned}$$

We will denote functors by single letters in sans serif font (because we need capital Roman letters for types and relations, introduced below), or by identifiers in ordinary italic font. In particular, $id$ denotes both the identity function (on some given type) and also the identity functor.

The declaration of *listr* $A$ also introduces two functions

$$nil : listr\,A \leftarrow 1 \qquad \text{and} \qquad cons : listr\,A \leftarrow A \times listr\,A$$

https://doi.org/10.1017/S0956796800001556 Published online by Cambridge University Press

that serve to construct lists. We can parcel these functions together as one function

$$[nil, cons] : listr\ A \leftarrow \mathsf{F}(A, listr\ A).$$

In general, if $f : A \leftarrow B$ and $g : A \leftarrow C$, then $[f,g] : A \leftarrow B + C$ applies $f$ to left components and $g$ to right components. The function $[nil, cons]$ has a special property, which captures the fact that we can define functions on lists by pattern-matching: given any function $[c,f] : B \leftarrow \mathsf{F}(A, B)$ there is a unique function $h : B \leftarrow listr\ A$ such that

$$h \cdot [nil, cons] \quad = \quad [c,f] \cdot \mathsf{F}(id, h).$$

Unwrapping this compact equation, we get two equations

$$\begin{aligned} h \cdot nil &= c \\ h \cdot cons &= f \cdot (id \times h). \end{aligned}$$

In functional programming $h$ is written in the form $h = foldr\,(c,f)$, but we will use the alternative notation $h = (\![c,f]\!)$. A function $h$ defined in this way is called a *catamorphism*, a term meaning 'according to form'. In functional programming, catamorphisms are what are known as *fold* operators. We have already met the fold operator corresponding to the type *listl A*, namely *foldl*.

Before describing the general situation, let us give one more example. The declaration of type *tree A* in section 2 asserts $tree\ A \cong \mathsf{F}(A, tree\ A)$, where this time the base functor $\mathsf{F}$ is given by

$$\mathsf{F}(A, B) \quad = \quad 1 + A \times B \times B.$$

The functor $\mathsf{F}$ is polynomial. The declaration of *tree A* also introduces two functions

$$nil : tree\ A \leftarrow 1 \qquad \text{and} \qquad fork : A \times tree\ A \times tree\ A$$

that serve to construct trees. As before, we have

$$[nil, fork] : tree\ A \leftarrow \mathsf{F}(A, tree\ A).$$

The function $[nil, fork]$ has a special property that given any function $[c,f] : B \leftarrow \mathsf{F}(A, B)$ there is a unique function $h : B \leftarrow tree\ A$ such that

$$h \cdot [nil, tree] \quad = \quad [c,f] \cdot \mathsf{F}(id, h).$$

This time $h$ is a catamorphism on trees. Again we write $h = (\![c,f]\!)$, so the notation $(\![-]\!)$ is implicitly parameterised by the base functor of the datatype.

Let us now consider the general situation. Think of the declaration of a datatype *term A* as providing two pieces of information: a polynomial base functor $\mathsf{F}(A, B)$, and a named function $\alpha : term\ A \leftarrow \mathsf{F}(A, term\ A)$, which we will call the *constructor* of the type. The construtor $\alpha$ has the special property that given any function $f : B \leftarrow \mathsf{F}(A, B)$ there is a unique function $h$, written $h = (\![f]\!)$, satisfying

$$h \cdot \alpha \quad = \quad f \cdot \mathsf{F}(id, h). \tag{1}$$

As a consequence of this property of $\alpha$ we get that $(\![\alpha]\!) = id$. For example, $(\![nil, cons]\!)$ (which now we should write more accurately as $(\![[nil, cons]]\!)$ but won't) is the identity

https://doi.org/10.1017/S0956796800001556 Published online by Cambridge University Press

function on lists. Less obviously, it also follows from its defining property that $\alpha$ is an isomorphism, meaning

$$\alpha \cdot \alpha^{\circ} = id \qquad \text{and} \qquad \alpha^{\circ} \cdot \alpha = id,$$

where $\alpha^{\circ}$ denotes the inverse function to $\alpha$. The first $id$ is the identity on $term\ A$, and the second is the identity on $\mathsf{F}(A, term\ A)$, so $\alpha$ is the isomorphism that establishes $term\ A \cong \mathsf{F}(A, term\ A)$.

Using the fact that $h$ is uniquely characterised by (1), we obtain the following useful *fusion* rule for combining two functions into one:

$$f \cdot (\!|g|\!) = (\!|h|\!) \quad \Leftarrow \quad f \cdot g = h \cdot \mathsf{F}(id, f).$$

The proof is:

$$\begin{array}{ll}
 & f \cdot (\!|g|\!) = (\!|h|\!) \\
\equiv & \quad \{(1) \text{ for } (\!|h|\!)\} \\
 & f \cdot (\!|g|\!) \cdot \alpha = h \cdot \mathsf{F}(id, f \cdot (\!|g|\!)) \\
\equiv & \quad \{(1) \text{ for } (\!|g|\!)\} \\
 & f \cdot g \cdot \mathsf{F}(id, (\!|g|\!)) = h \cdot \mathsf{F}(id, f \cdot (\!|g|\!)) \\
\equiv & \quad \{\text{property of functors}\} \\
 & f \cdot g \cdot \mathsf{F}(id, (\!|g|\!)) = h \cdot \mathsf{F}(id, f) \cdot \mathsf{F}(id, (\!|g|\!)) \\
\Leftarrow & \quad \{\} \\
 & f \cdot g = h \cdot \mathsf{F}(id, f).
\end{array}$$

Since $\alpha$ is an isomorphism, we can move it to the other side of the defining equation for $(\!|f|\!)$. Thus, $h = (\!|f|\!)$ is the unique solution of the equation

$$h \quad = \quad f \cdot \mathsf{F}(id, h) \cdot \alpha^{\circ}.$$

We will use this fact below when we generalise to relations.

One further function is introduced whenever we declare a datatype $term\ A$; this is a function $term\ f$ with type $term\ A \leftarrow term\ B$ when $f : A \leftarrow B$. The definition is

$$term\ f \quad = \quad (\!|\alpha \cdot \mathsf{F}(f, id)|\!).$$

In the case $term = listr$ this definition expands to the equations

$$\begin{aligned} listr\ f \cdot nil &= nil \\ listr\ f \cdot cons &= cons \cdot (f \times id), \end{aligned}$$

and defines the familiar map operation on lists: $listr\ f\ x$ applies $f$ to every element of $x$. We denote the map operation on $term\ A$ by $term\ f$ because $term$ is a functor. It is immediate that $term\ id = id$, and the proof that

$$term\,(f \cdot g) \quad = \quad term\ f \cdot term\ g$$

is a simple exercise in the fusion law, which is left to the reader. We will call *term* a *type* functor. Although it is not polynomial, we can allow it in the declarations of other datatypes without altering the theory given above. For instance, the type

$$tree\ A \quad ::= node\,(A, listr\,(tree\ A))$$

introduces a datatype based on the non polynomial functor $\mathsf{F}(A, B) = A \times listr\ B$. In this case the constructor function $\alpha = node$.

We will call a datatype *inductive* if its base functor is polynomial or a type functor. In what follows we restrict attention to inductive datatypes.

## 5 Relations

Now, let us extend the foregoing theory to relations. We write $R : A \leftarrow B$ to denote that $R$ is a relation of type '$A$ from $B$'; we can think of $R$ as a subset of $A \times B$. Relational composition, like its functional counterpart, goes backwards: $R \cdot S$ is pronounced '$R$ after $S$'. We reserve single lower-case letters $f, g$ and so on, to denote functions.

Unlike functions, every relation has a converse. If $R : A \leftarrow B$ then the *converse* relation is $R^\circ : B \leftarrow A$. Converse preserves identities but reverses composition, so $(R \cdot S)^\circ = S^\circ \cdot R^\circ$.

For each $A$ and $B$ the relations of type $A \leftarrow B$ form a complete lattice with union $\cup$ and intersection $\cap$. Relations with the same type can be compared via a partial order $\subseteq$, where $R \subseteq S$ denotes $R \cap S = R$. We will sometimes use $\supseteq$ rather than $\subseteq$ in writing inequations because $\supseteq$ can be interpreted as refinement: $R \supseteq S$ if $R$ refines to $S$. Thus, a chain of inclusions

$$S \supseteq S_1 \supseteq \cdots \supseteq S_n \supseteq f$$

can be interpreted as a stepwise refinement of a relational specification $S$ into a function (an executable program) $f$.

Converse preserves $\subseteq$ and composition distributes over (arbitrary) unions, but only weakly distributes over intersection in that

$$R \cdot (S \cap T) \quad \subseteq \quad (R \cdot S) \cap (R \cdot T).$$

We will suppose in what follows that composition binds more tightly than any other operation, so the right-hand side could have been written without brackets. Using the given properties of converse, we get from the above inequation a second one:

$$(R \cap S) \cdot T \quad \subseteq \quad (R \cdot T) \cap (S \cdot T).$$

These two inequations say that composition is monotonic in both arguments under $\subseteq$. One further inequation, called the *modular law*, is adjoined to the other axioms to give a weak converse of distributivity over intersection:

$$(R \cdot S) \cap T \quad \subseteq \quad R \cdot (S \cap R^\circ \cdot T).$$

There is more to be said about the calculus of relations, which is based on Freyd's theory of allegories (Freyd and Ščedrov, 1990), but we will postpone saying it to later. For now let us concentrate on the main point, which is that everything we have said above about datatypes goes through when functions are extended to relations, provided only that we restrict attention to *monotonic* functors; that is, if $R \subseteq S$ then $\mathsf{F}R \subseteq \mathsf{F}S$. In particular, polynomial functors and type functors are monotonic. It can be shown that every functor on functions can be extended to a monotonic functor on relations in at most one way. It can also be shown that such functors preserve relational converse, so $(\mathsf{F}R)^\circ = \mathsf{F}(R^\circ)$. It follows that the expression $\mathsf{F}R^\circ$ is not ambiguous.

By extending the theory to relations we get relational catamorphisms as well as functional ones. Moreover, since converse reverses composition we get, for a relation $R : term\, A \leftarrow \mathsf{F}(A, term\, A)$, not only that $X = (\![R]\!)$ is the unique solution of

$$X \quad = \quad R \cdot \mathsf{F}(id, X) \cdot \alpha^\circ$$

but also that $X = (\![R]\!)^\circ$ is the unique solution of

$$X \quad = \quad \alpha \cdot \mathsf{F}(id, X) \cdot R^\circ.$$

By the Knaster–Tarski theorem the unique solution (if it exists) of $X = \phi X$ is also the least solution (under relational inclusion) of $X \supseteq \phi X$ and the greatest solution of $X \subseteq \phi X$, so we get all three of the following versions of the characterisation of catamorphisms:

$$\begin{array}{lcl} X = R \cdot \mathsf{F}X \cdot \alpha^\circ & \equiv & X = (\![R]\!) \\ X \subseteq R \cdot \mathsf{F}X \cdot \alpha^\circ & \Rightarrow & X \subseteq (\![R]\!) \\ X \supseteq R \cdot \mathsf{F}X \cdot \alpha^\circ & \Rightarrow & X \supseteq (\![R]\!). \end{array}$$

With relations we also get two variants of the fusion rule:

$$\begin{array}{lcl} R \cdot (\![S]\!) \subseteq (\![T]\!) & \Leftarrow & R \cdot S \subseteq T \cdot \mathsf{F}(id, R) \\ R \cdot (\![S]\!) \supseteq (\![T]\!) & \Leftarrow & R \cdot S \supseteq T \cdot \mathsf{F}(id, R). \end{array}$$

Finally, writing $(\mu X : \phi X)$ for the least fixed point of $\phi$, we get the following rule:

$$(\![R]\!) \cdot (\![S]\!)^\circ \quad = \quad (\mu X : R \cdot \mathsf{F}(id, X) \cdot S^\circ). \tag{2}$$

With $S : C \leftarrow \mathsf{F}(A, C)$ and $R : B \leftarrow \mathsf{F}(A, B)$ we have $(\![S]\!)^\circ : term\, A \leftarrow C$ and $(\![R]\!) : B \leftarrow term\, A$, so (2) is a rule for eliminating the intermediate datatype $term\, A$ from a computation.

## 6 Prunings and Horner's rule

Let us now proceed to formalise the notion of pruning introduced in section 2, and to state and prove a general version of Horner's rule. For simplicity, we will assume that the base functor $\mathsf{F}$ of the datatype $term\, A$ takes a particular form, namely

$$\mathsf{F}(A, B) \quad = \quad 1 + \mathsf{G}(A, B),$$

for some binary functor $\mathsf{G}$ which is not further specified. We follow Backhouse (Aarts *et al.*, 1992) in calling such functors *pointed*. Since $\mathsf{F}(A, B)$ is a coproduct, the constructor function $\alpha : term\, A \leftarrow \mathsf{F}(A, term\, A)$ can be written in the form $\alpha = [\alpha_0, \alpha_1]$, where $\alpha_0 : term\,A \leftarrow 1$ and $\alpha_1 : term\,A \leftarrow \mathsf{G}(A, term\,A)$. One can think of $\alpha_0$ as a constant returning the empty element of $term\,A$; in the sequel we will also use $\alpha_0$ as a constant function of type $term\,A \leftarrow B$. More precisely, this constant function should be written as $\alpha_0 \cdot !_B$, where $!_B$ is the unique function of type $1 \leftarrow B$. The same notational abbreviation will be used for other constants; thus, if $c : A \leftarrow 1$ we will also write $c$ for the constant function $c \cdot !_B : A \leftarrow B$.

Now, to prune a term $x$ means to substitute $\alpha_0$ for some (zero or more) subterms of $x$. The relation $prune : term\,A \leftarrow term\,A$ takes a term and prunes it in some arbitrary way:

$$prune \quad = \quad (\![\alpha_0, \alpha_0 \cup \alpha_1]\!).$$

The first $\alpha_0$ has type $term\,A \leftarrow 1$, while the second has type $term\,A \leftarrow \mathsf{G}(A, term\,A)$.

The function $prunings$ in Section 2 is the set of possible results returned by $prune$:

$$prunings\ x \quad = \quad \{y \mid y\ prune\ x\}.$$

In the relational calculus the mapping that associates with each relation the corresponding set-valued function is denoted by $\Lambda$. Thus, if $R : A \leftarrow B$, then $\Lambda R : \mathsf{P}A \leftarrow B$, where $\mathsf{P}A$ denotes the powerset of $A$. In particular, $prunings = \Lambda prune$. Writing $\mathsf{P}$ in sans serif font suggests that it is a functor; and indeed it is: $\mathsf{P}f$ is the function that applies $f$ to every element of a set. Thus $\mathsf{P}f$ is the function that we wrote as $map\,f$ in section 3.

So as not to lose the main thread in the mass of detail to come, we will now state the general version of Horner's rule, ignoring the fact that its formulation contains some concepts that we have not yet formally defined:

*Lemma 6.1*
(Horner's rule). Let $\mathsf{F}$ be a pointed, monotonic functor and $f = [f_0, f_1] : B \leftarrow \mathsf{F}(A, B)$ be a function. Furthermore, suppose $R : B \leftarrow B$ is a preorder such that $f$ is monotonic under $R$. Then

$$max\ R \cdot \mathsf{P}(\![f]\!) \cdot \Lambda prune \quad \supseteq \quad (\![max\ R \cdot \Lambda[f_0, f_0 \cup f_1]]\!).$$

Horner's rule gives conditions under which one computation can be refined by another; the computation on the left takes the set of all prunings, applies a functional catamorphism to each pruning, and takes a maximum under a relation $R$; the computation on the right is a relational catamorphism that selects a maximum at each step.

Let us now explain the additional concepts in the statement of Horner's rule. First, a preorder is a reflexive, transitive relation, so $R : B \leftarrow B$ is a preorder if $id \subseteq R$ and $R \cdot R \subseteq R$. Next, and for the moment informally, $max\ R : B \leftarrow \mathsf{P}B$ is a relation that, given a set, returns some maximum element under $R$. Finally, a function $f : B \leftarrow \mathsf{F}(A, B)$ is monotonic under $R$ if

$$f \cdot \mathsf{F}(id, R) \quad \subseteq \quad R \cdot f.$$

Here are two examples to explain the definition of monotonicity:

1. *Addition.* Let $R = leq$, where $leq : Nat \leftarrow Nat$ denotes the relation $\leq$ on natural numbers, and let $f = plus$, where $plus : Nat \leftarrow Nat \times Nat$ denotes binary addition. Taking $\mathsf{F}(A,B) = A \times B$ the monotonicity condition translates to

$$plus \cdot (id \times leq) \quad \subseteq \quad leq \cdot plus$$

and says that $x = y + z$ and $z \leq z'$ implies $x \leq y + z'$. This is just the (true) statement that addition is monotonic in its right argument. We can also take $\mathsf{F}(A,B) = B \times B$, in which case the monotonicity condition is

$$plus \cdot (leq \times leq) \quad \subseteq \quad leq \cdot plus$$

and asserts that addition is monotonic in both arguments.

2. *Cons lists.* Let $R = lex$, where $lex$ is the lexicographic ordering on lists, and $f = [nil, cons]$. With $\mathsf{F}(A,B) = 1 + A \times B$ the monotonicity condition translates to

$$\begin{aligned} nil \quad &\subseteq \quad lex \cdot nil \\ cons \cdot (id \times lex) \quad &\subseteq \quad lex \cdot cons. \end{aligned}$$

The first equation follows at once from the reflexivity of $lex$, and the second asserts the true statment that $cons$ is monotonic with respect to the lexicographic ordering.

To prove Horner's rule we need to define $max\ R$ formally in the relational calculus, and also to be more precise about the relationship between $\mathsf{P}$ and $\Lambda$. We begin with the latter.

### 6.1 Powersets

Formally, the isomorphism between relations and set-valued functions can be described in the following suitably abstract form. For every set $A$ there exists a set $\mathsf{P}A$, called the *powerset* of $A$, and a relation $\in : A \leftarrow \mathsf{P}A$, called the *membership* relation on $A$, which together are characterised by the following property: for every relation $R : A \leftarrow B$, there exists a function $\Lambda R : \mathsf{P}A \leftarrow B$ such that

$$(f = \Lambda R) \quad \equiv \quad (\in \cdot f = R) \quad \text{for all } f : \mathsf{P}A \leftarrow B.$$

The function $\Lambda R$ is said to be the *power transpose* of $R$ and can be defined in set theory by $(\Lambda R)\, b = \{a \mid aRb\}$. (In fact, much of set theory can be recovered using just this universal property of powersets, plus the relational calculus. This observation lies at the heart of the categorical approach to sets, namely the theory of *toposes* (Johnstone, 1977; Barr and Wells, 1985; Goldblatt, 1986).)

It is immediate from the universal property of $\Lambda$ that

$$\in \cdot \Lambda R \quad = \quad R.$$

Below we refer to this fact by the hint '$\Lambda$ cancellation'. It also follows from the universal property that $id : \mathsf{P}A \leftarrow \mathsf{P}A$ satisfies

$$id \quad = \quad \Lambda(\in).$$

Using $\Lambda$ we can define the *existential image* of a relation $R : A \leftarrow B$; this is a function $\mathsf{E}R : \mathsf{P}A \leftarrow \mathsf{P}B$ defined by

$$\mathsf{E}R \quad = \quad \Lambda(R \cdot \in).$$

In set theory we have

$$(\mathsf{E}R)\,x \quad = \quad \{a \mid (\exists b : a\,R\,b \wedge b \in x)\}.$$

It is clear from its definition that $\mathsf{E}id = id$, and below we will show that $\mathsf{E}(R \cdot S) = \mathsf{E}R \cdot \mathsf{E}S$, so $\mathsf{E}$ is a functor (taking its action on types to be $\mathsf{E}A = \mathsf{P}A$). It is not, however, a monotonic functor on relations because it returns a function and $f \subseteq g$ if and only if $f = g$.

To show $\mathsf{E}$ is a functor we first prove $\Lambda(R \cdot S) = \mathsf{E}R \cdot \Lambda S$:

$$\begin{array}{cl} & \Lambda(R \cdot S) = \mathsf{E}R \cdot \Lambda S \\ \equiv & \quad \{\text{definition of } \Lambda\} \\ & R \cdot S = \in \cdot\, \mathsf{E}R \cdot \Lambda S \\ \equiv & \quad \{\text{definition of } \mathsf{E}\} \\ & R \cdot S = \in \cdot\, \Lambda(R \cdot \in) \cdot \Lambda S \\ \equiv & \quad \{\Lambda \text{ cancellation (twice)}\} \\ & \mathit{true} \end{array}$$

Now, taking $S = T \cdot \in$, we get $\mathsf{E}(R \cdot T) = \mathsf{E}R \cdot \mathsf{E}T$.

Although $\mathsf{E}$ is not monotonic, there does exist a variant of $\mathsf{E}$ which is, namely the *powerset* functor $\mathsf{P}$. Over functions, this functor has the same action as $\mathsf{E}$, so $\mathsf{P}f = \mathsf{E}f$. For a relation $R$ the definition of $\mathsf{P}R$ in set theory is

$$x\,\mathsf{P}R\,y \quad = \quad (\forall a \in x : \exists b \in y : aRb) \;\wedge\; (\forall b \in y : \exists a \in x : aRb).$$

For use below, we note the following two identities, in which the function *union* : $\mathsf{P}A \leftarrow \mathsf{PP}A$ is defined by *union* $= \mathsf{E}(\in)$; this function returns the union of a collection of sets. The identities are:

$$\Lambda(R \cdot S) \quad = \quad \mathit{union} \cdot \mathsf{P}\Lambda R \cdot \Lambda S \tag{3}$$

$$\mathsf{P}f \cdot \mathit{union} \quad = \quad \mathit{union} \cdot \mathsf{PP}f. \tag{4}$$

Equation (4) appeared in section 3. Both equations are easy consequences of the definitions above and we omit details; for a further discussion of $\mathsf{P}$ and its relation to $\mathsf{E}$, see De Moor (1992).

### 6.2 *Division*

To define *max* $R$ we need the operation of relational division. Because relational composition distributes over arbitrary unions, it has a weak inverse, called *division*, which is characterised by the equivalence

$$T \subseteq R/S \quad \equiv \quad T \cdot S \subseteq R \quad \text{for all } T.$$

https://doi.org/10.1017/S0956796800001556 Published online by Cambridge University Press

The operator / can be defined in set theory by

$$a\,(R/S)\,b \quad\equiv\quad (\forall c : b\,S\,c : a\,R\,c).$$

A second division operator \ can be introduced by defining $R\backslash S = (S^\circ/R^\circ)^\circ$, so

$$T \subseteq R\backslash S \quad\equiv\quad R \cdot T \subseteq S \quad \text{for all } T.$$

As a predicate we have $a\,(R\backslash S)\,b \equiv (\forall c : c\,R\,a : c\,S\,b)$.

Above we defined $\mathsf{P}R$ in set theoretical terms; using division we can define

$$\mathsf{P}R \quad=\quad \in\backslash(R \cdot \in) \cap (\ni \cdot R)/\ni,$$

where $\ni$ is shorthand for $\in^\circ$. The expression on the right is the translation of the earlier one into the relational calculus.

Using division we can define the relation $max\ R : A \leftarrow \mathsf{P}A$ by

$$max\ R \quad=\quad \in \cap\,(R^\circ/\ni).$$

This definition corresponds to the usual one in set theory: $a(max\ R)x$ holds when $a$ is an element of $x$ (the first term) and $x$ has upper bound $a$ (the second term), that is, for all $b \in x$, we have $bRa$. Although the definition of $max\ R$ does not require $R$ to be a preorder, it is useful only when $R$ is one, so we will tacitly assume it so.

### *6.3 Properties of max*

There are two properties of *max* that we will need. First of all,

$$X \subseteq max\ R \cdot \Lambda S \quad\equiv\quad (X \subseteq S) \wedge (X \cdot S^\circ \subseteq R^\circ). \tag{5}$$

We give the proof of (5) because it is typical of the kind of manipulations found in the relational calculus. The calculation makes use of the following two rules in whuch $f$ is a function and $R$ and $S$ arbitrary relations. Firstly, we have the distributive law

$$(R \cap S) \cdot f \quad=\quad (R \cdot f) \cap (S \cdot f),$$

and secondly the *shunting* law

$$R \subseteq S \cdot f \quad\equiv\quad R \cdot f^\circ \subseteq S.$$

The proof of (5) is:

$$\begin{array}{cl}
& X \subseteq max\ R \cdot \Lambda S \\
\equiv & \quad \{\text{definition of } max\ R\} \\
& X \subseteq (\in \cap\,(R^\circ/\ni)) \cdot \Lambda S \\
\equiv & \quad \{\text{distributive law (above)}\} \\
& X \subseteq (\in \cdot \Lambda S) \cap ((R^\circ/\ni) \cdot \Lambda S) \\
\equiv & \quad \{\Lambda \text{ cancellation and universal property of } \cap\} \\
& (X \subseteq S) \wedge (X \subseteq (R^\circ/\ni) \cdot \Lambda S).
\end{array}$$

https://doi.org/10.1017/S0956796800001556 Published online by Cambridge University Press

Continuing with the second term:

$$
\begin{aligned}
& X \subseteq (R^\circ/\ni) \cdot \Lambda S \\
\equiv & \quad \{\text{shunting law (above)}\} \\
& X \cdot (\Lambda S)^\circ \subseteq R^\circ/\ni \\
\equiv & \quad \{\text{universal property of } /\} \\
& X \cdot (\Lambda S)^\circ \cdot \ni \subseteq R^\circ \\
\equiv & \quad \{\text{converse and } \Lambda \text{ cancellation}\} \\
& X \cdot S^\circ \subseteq R^\circ.
\end{aligned}
$$

The second fact is that *max R* weakly distributes over *union*:

$$
\mathit{max}\ R \cdot \mathit{union} \quad \supseteq \quad \mathit{max}\ R \cdot \mathsf{P}(\mathit{max}\ R). \tag{6}
$$

We will sketch the proof since it uses (5). Since $\mathit{union} = \mathsf{E}(\in) = \Lambda(\in \cdot \in)$, condition (5) shows it is sufficient to prove:

$$
\begin{aligned}
\mathit{max}\ R \cdot \mathsf{P}(\mathit{max}\ R) & \subseteq \in \cdot \in \\
\mathit{max}\ R \cdot \mathsf{P}(\mathit{max}\ R) \cdot \ni \cdot \ni & \subseteq R^\circ.
\end{aligned}
$$

The first is easy using $\mathit{max}\ R \subseteq \in$ and $\in \cdot \mathsf{P}X \subseteq X \cdot \in$, and for the second we use $\mathsf{P}X \cdot \ni \subseteq \ni \cdot X$ and $\mathit{max}\ R \cdot \ni \subseteq R^\circ$ to write the inclusion in the form $R^\circ \cdot R^\circ \subseteq R^\circ$, which follows from the transitivity of $R$.

We saw a version of (6) in section 3 where it appeared as an equation, but with the qualification that it applied only to sets of non-empty sets. It is one of the advantages of a relational approach that we can omit the qualification provided we replace equality by refinement.

### *6.4 Proof of Horner's rule*

We are now ready for the proof of Horner's rule. Our aim is to apply the universal property for *max R*, so we begin

$$
\begin{aligned}
& \mathit{max}\ R \cdot \mathsf{P}(\!|f|\!) \cdot \Lambda \mathit{prune} \\
= & \quad \{\text{since } \mathsf{P} = \mathsf{E} \text{ on functions}\} \\
& \mathit{max}\ R \cdot \mathsf{E}(\!|f|\!) \cdot \Lambda \mathit{prune} \\
= & \quad \{\text{since } \mathsf{E}X \cdot \Lambda Y = \Lambda(X \cdot Y)\} \\
& \mathit{max}\ R \cdot \Lambda((\!|f|\!) \cdot \mathit{prune}) \\
= & \quad \{\text{definition of } \mathit{prune}\} \\
& \mathit{max}\ R \cdot \Lambda((\!|f|\!) \cdot (\!|\alpha_0, \alpha_0 \cup \alpha_1|\!)) \\
= & \quad \{\text{fusion (see below)}\} \\
& \mathit{max}\ R \cdot \Lambda(\!|f_0, f_0 \cup f_1|\!).
\end{aligned}
$$

https://doi.org/10.1017/S0956796800001556 Published online by Cambridge University Press

The condition for fusion is that

$$(\![f]\!) \cdot [\alpha_0, \alpha_0 \cup \alpha_1] \quad = \quad [f_0, f_0 \cup f_1] \cdot \mathsf{F}(id, (\![f]\!))$$

and is easily verified.

Abbreviating $[f_0, f_0 \cup f_1]$ by $S$ and appealing to (5), we get Horner's rule by showing

$$\begin{array}{rcl} (\![max\ R \cdot \Lambda S]\!) & \subseteq & (\![S]\!) \\ (\![max\ R \cdot \Lambda S]\!) \cdot (\![S]\!)^\circ & \subseteq & R^\circ. \end{array}$$

The first inequation is easy, since $max\ R \cdot \Lambda S \subseteq \in \cdot \Lambda S = S$ and catamorphisms are monotonic. For the second inequation we appeal to (2) and show

$$max\ R \cdot \Lambda S \cdot \mathsf{F}(id, R^\circ) \cdot S^\circ \quad \subseteq \quad R^\circ.$$

To do this, we need the monotonicity of $f$ under $R$. It is easy to show that this condition implies that $S$ is also monotonic under $R$ in the sense that

$$S \cdot \mathsf{F}(id, R) \quad \subseteq \quad R \cdot S.$$

Taking converses, and recalling the assumption that $\mathsf{F}$ is a monotonic functor and so preserves converse, we obtain

$$\mathsf{F}(id, R^\circ) \cdot S^\circ \quad \subseteq \quad S^\circ \cdot R^\circ.$$

Now we can argue:

$$\begin{array}{ll} & max\ R \cdot \Lambda S \cdot \mathsf{F}(id, R^\circ) \cdot S^\circ \\ \subseteq & \quad \{\text{above}\} \\ & max\ R \cdot \Lambda S \cdot S^\circ \cdot R^\circ \\ \subseteq & \quad \{\text{since } \Lambda X \cdot X^\circ \subseteq \ni \text{ by shunting and } \Lambda \text{ cancellation}\} \\ & max\ R \cdot \ni \cdot R^\circ \\ \subseteq & \quad \{\text{since } max\ R \subseteq R^\circ / \ni \text{ and universal property of } /\} \\ & R^\circ \cdot R^\circ \\ \subseteq & \quad \{\text{converse and transitivity of } R\} \\ & R^\circ. \end{array}$$

The proof of Horner's rule is complete.

Before we can solve the problem of defining *subterms* and generalising the Scan lemma, we need to give some more theory about datatypes.

## 7 Natural transformations and membership

Datatypes record the presence of elements, so one would expect a type *term* $A$ to come equipped with a *membership* relation $\delta_A : A \leftarrow term\ A$ such that $a \delta_A x$ precisely when $a$ is an element of $x$. Note that $\delta_A$ is a *collection* of relations, one for each type $A$. We have already seen one membership relation, namely $\in_A : A \leftarrow \mathsf{P}A$, the ordinary membership relation for sets. To define the notion of membership for an arbitrary datatype, we first explain what it means for a collection of relations to be a natural transformation.

https://doi.org/10.1017/S0956796800001556 Published online by Cambridge University Press

### 7.1 Natural transformations

A number of functions that we have met already are really collections of functions; for example

$$\begin{aligned} id_A &: A \leftarrow A \\ \alpha_A &: term\ A \leftarrow \mathsf{F}(A, term\ A) \\ prunings_A &: \mathsf{P}(term\ A) \leftarrow term\ A. \end{aligned}$$

The functions in each collection do not depend in any essential way on the parameter $A$, a fact which is captured by a suitable 'type changing' rule. For example, for any function $f : A \leftarrow B$ we have

$$\begin{aligned} f \cdot id_B &= id_A \cdot f \\ term\, f \cdot \alpha_B &= \alpha_A \cdot \mathsf{F}(f, term\, f). \end{aligned}$$

The second identity comes from the definition $term\, f = (\!|\alpha \cdot \mathsf{F}(f, id)|\!)$. Formally, a collection of functions $\phi_A : \mathsf{F}A \leftarrow \mathsf{G}A$ is called a *natural transformation* if for all $f : A \leftarrow B$ we have

$$\mathsf{F}f \cdot \phi_B = \phi_A \cdot \mathsf{G}f.$$

We write $\phi : \mathsf{F} \leftarrow \mathsf{G}$ to indicate that $\phi$ is natural. In particular, we have $id : id \leftarrow id$, and $\alpha : term \leftarrow \mathsf{G}$, where $\mathsf{G}A = \mathsf{F}(A, term\ A)$.

The notion of natural transformation extends to relations: a collection of relations $\phi : \mathsf{F} \leftarrow \mathsf{G}$ is natural if for all relations $R : A \leftarrow B$, we have $\mathsf{F}R \cdot \phi_B = \phi_A \cdot \mathsf{G}R$. For example, we have $\in : id \leftarrow \mathsf{E}$.

There is a weaker notion of natural transformation, more useful when dealing with relations: a collection of relations $\phi_A : \mathsf{F}A \leftarrow \mathsf{G}A$ is a *weak* natural transformation when for all $R : A \leftarrow B$ we have

$$\mathsf{F}R \cdot \phi_B \supseteq \phi_A \cdot \mathsf{G}R.$$

We write $\phi : \mathsf{F} \hookleftarrow \mathsf{G}$ to indicate that $\phi$ is weakly natural. It is a fact that $\phi$ is weakly natural if and only if $\mathsf{F}f \cdot \phi_B = \phi_A \cdot \mathsf{G}f$ for all functions $f : A \leftarrow B$. For a proof see Carboni *et al.* (1991). Thus, we can show $\phi$ is weakly natural for relations by showing it is natural for functions.

An important example is $\in : id \hookleftarrow \mathsf{P}$; we have $R \cdot \in \supseteq \in \cdot \mathsf{P}R$ for all $R$, but this inclusion cannot be strengthened to an equality.

### 7.2 Membership

Now, let us return to membership. Formally, a collection of arrows $\delta_A : A \leftarrow \mathsf{F}A$ is a *membership* relation for $\mathsf{F}$ if for each $R : A \leftarrow B$

$$\mathsf{F}R \cdot (\delta_B \backslash id_B) = \delta_A \backslash R.$$

Rather than attempt to explain this definition, we will give a number of properties that justify it. For formal proofs of these properties, see De Moor (1993). First, the above equation has at most one solution for $\delta$ because any $\delta$ satisfying it is the largest weak natural transformation of type $id \hookleftarrow \mathsf{F}$.

Second, although not every functor has membership, all polynomial functors do. These membership relations are given inductively by the following clauses:

$$\begin{array}{rcl} \delta_{\mathsf{id}} & = & id \\ \delta_{\mathsf{K}A} & = & \emptyset \\ \delta_{\mathsf{F}+\mathsf{G}} & = & [\delta_{\mathsf{F}}, \delta_{\mathsf{G}}] \\ \delta_{\mathsf{F}\times\mathsf{G}} & = & \delta_{\mathsf{F}} \cdot outl \;\cup\; \delta_{\mathsf{G}} \cdot outr \\ \delta_{\mathsf{F}\cdot\mathsf{G}} & = & \delta_{\mathsf{G}} \cdot \delta_{\mathsf{F}} \end{array}$$

In the second clause $\emptyset$ denotes the empty relation; in the fourth clause the functions $outl : A \leftarrow A \times B$ and $outr : B \leftarrow A \times B$ are the projection functions. These functions have the property that for every two functions $f : A \leftarrow C$ and $g : B \leftarrow C$ there is a unique function $\langle f, g \rangle : A \times B \leftarrow C$ such that $outl \cdot \langle f, g \rangle = f$ and $outr \cdot \langle f, g \rangle = g$. Since we can define $f \times g = \langle f \cdot outl, g \cdot outr \rangle$ we get that $outl$ and $outr$ satisfy

$$\begin{array}{rcl} f \cdot outl & = & outl \cdot (f \times g) \\ g \cdot outr & = & outr \cdot (f \times g), \end{array}$$

and so are natural transformations over functions. However, these equations are weakened when we consider relations:

$$\begin{array}{rcl} R \cdot outl & \supseteq & outl \cdot (R \times S) \\ S \cdot outr & \supseteq & outr \cdot (R \times S). \end{array}$$

These inclusions cannot be strengthened because, for instance, $R \times \emptyset = \emptyset$.

For type functors the question of membership is more complicated. Recall that $term f$ was defined as a catamorphism, so one might expect that its membership relation can also be expressed as a catamorphism. However, this is only true for datatypes that do not contain constants. For example, consider the type

$$listr^+ A \quad ::= \quad wrap\, A \mid cons^+ (A, listr^+ A),$$

of non-empty lists, with base functor $\mathsf{F}(A, B) = A + A \times B$. The membership relation $\delta$ for these lists is given by

$$\delta \quad = \quad (\![id, outl \cup outr]\!).$$

In words, an arbitrary member is obtained at each stage either by selecting the new element or by retaining the chosen element from the previous stage.

With the type $listr\, A$, in which the constructor $wrap$ is replaced by $nil$, the membership relation is *not* given by

$$\delta \quad = \quad (\![\emptyset, outl \cup outr]\!)$$

because the right-hand side is the empty relation $\emptyset$, as one can easily check.

Fortunately, there is another approach to membership of type functors, one that makes use of two auxiliary relations, which we will call *root* and *spur*. To fix

https://doi.org/10.1017/S0956796800001556 Published online by Cambridge University Press

notation, let $\alpha : term\ A \leftarrow \mathsf{F}(A, term\ A)$ be the constructor of $term\ A$ and let

$$\begin{aligned} left &: A \leftarrow \mathsf{F}(A,B) \\ right &: B \leftarrow \mathsf{F}(A,B) \end{aligned}$$

be the two membership relations associated with the binary functor F. More precisely, *left* is the membership relation for the functor $\mathsf{G}_B(A) = \mathsf{F}(A,B)$ in which $B$ is fixed; similarly for *right*. The relations $root : A \leftarrow term\ A$ and $spur : term\ A \leftarrow term\ A$ are defined by

$$\begin{aligned} root &= left \cdot \alpha^\circ \\ spur &= right \cdot \alpha^\circ . \end{aligned}$$

Let us explain these relations with the help of some examples.

1. *Snoc lists.* With $\mathsf{F}(A,B) = 1 + (B \times A)$, we get $left = [\emptyset, outr]$ and $right = [\emptyset, outl]$, so

   $$root = left \cdot \alpha^\circ = [\emptyset, outr] \cdot [nil, snoc]^\circ = outr \cdot snoc^\circ .$$

   This uses the law $[R,S] \cdot [T,U]^\circ = R \cdot T^\circ \cup S \cdot U^\circ$. Similarly, we get $spur = outl \cdot snoc^\circ$. In words, *root* is *last*, the partial function that returns the last element of a (nonempty) list, and *spur* is *init*, the partial function that removes the last element.
2. *Cons lists.* Dually, with the base functor $\mathsf{F}(A,B) = 1 + (A \times B)$, we get $root = outl \cdot cons^\circ$ and $spur = outr \cdot cons^\circ$, so *root* is the partial function that returns the first element of a list and *spur* removes it.
3. *Binary trees.* With $\mathsf{F}(A,B) = 1 + (A \times (B \times B))$, we get the type *tree A* described in section 2. Here we have

   $$\begin{aligned} left &= [\emptyset, outl] \\ right &= [\emptyset, (outl \cup outr) \cdot outr], \end{aligned}$$

   so

   $$\begin{aligned} root &= outl \cdot fork^\circ \\ spur &= (outl \cup outr) \cdot outr \cdot fork^\circ . \end{aligned}$$

   The partial function *root* returns the root of a nonempty tree, and *spur* returns one of its immediate subtrees.

Now we can define the membership relation $\delta : A \leftarrow term\ A$ by

$$\delta = root \cdot spur^* ,$$

where $R^*$ denotes the reflexive transitive closure of $R$. In the case of snoc-lists, an arbitrary member of a list is obtained by taking the last element of an arbitrary prefix; in the case of cons-lists, an arbitrary member is obtained by taking the first element of an arbirtrary suffix; and in the case of trees, an arbitrary member is obtained by taking the root of an arbitrary subtree.

The last sentence suggests that we now have a way to define an arbitrary subterm

https://doi.org/10.1017/S0956796800001556 Published online by Cambridge University Press

of a term: define $subterm = spur^{*}$. The set of subterms is then obtained as $\Lambda subterm$. This is perfectly correct, except that in the next section we will define *subterms* not as a set of terms but as a certain datatype containing terms as elements. The reason for this lies in the formulation of the Scan lemma, which depends critically in the case of lists on *subterms* returning a *list* of lists.

Since we will be using such datatypes to represent sets, we will need a way of moving from one to the other. The function $setify : \mathsf{P}A \leftarrow term\, A$ takes an element of the datatype $term\, A$ and returns a set of its elements. The definition is very simple: $setify = \Lambda\delta$, where $\delta$ is the membership relation for $term$. The function $setify$ is a weak natural transformation $setify : \mathsf{P} \hookleftarrow term$, so

$$\mathsf{P}R \cdot setify \quad \supseteq \quad setify \cdot term\, R. \tag{7}$$

The proof, which we omit, uses the formal definition of membership for a datatype.

Finally, we will need to know how to implement the relation $max\, R$ on datatypes other than sets. With *left* and *right* as above we have

$$max\, R \cdot setify \quad \supseteq \quad (\![maxR \cdot \Lambda(left \cup right)]\!). \tag{8}$$

Note that in the catamorphism on the right the relations *left* and *right* both have type $A \leftarrow \mathsf{F}(A, A)$. Note also that the inclusion cannot be strengthened to an equality because there may be constant terms that have no elements.

## 8 Subterms and scans

Recall from section 2 that we defined *subterms* as a set-valued function that returned all possible subterms of a given struture. In analogy with the definition of *prunings* it is therefore tempting to define $subterms = \Lambda subterm$, where *subterm* was defined above. But also recall the Scan lemma in section 3 which depends on $inits = subterms$ returning a *list* of subterms.

In an attempt to construct a generic version of the scan lemma, one might think of generating *subterms* as a list of structures, but that would still involve lists in an essential way. We really want to think of a *structure* of structures: a list of lists, a tree of trees, and so on. The way to achieve this is to create a new type of *labelled* structures in which each 'node' is labelled with the corresponding subterm. Not every datatype allows the labelling of nodes (think of unlabelled binary trees), but there is a canonical way of introducing labels into a datatype. We consider this first, returning to scans at the end of the section.

### *8.1 Labelled datatypes*

Again, let $\alpha : term\, A \leftarrow \mathsf{F}(A, term\, A)$ be the constructor of $term\, A$. Define another bifunctor $\mathsf{F}^{+}$ by

$$\mathsf{F}^{+}(A, B) \quad = \quad \mathsf{F}(1, B) \times A,$$

and let this be the base functor of a datatype $term^{+} A$ of 'labelled terms'. Let $\alpha^{+} : term^{+} A \leftarrow \mathsf{F}^{+}(A, term^{+} A)$ be its constructor. Here are some examples to clarify the idea.

https://doi.org/10.1017/S0956796800001556 Published online by Cambridge University Press

1. *Cons lists.* With $\mathsf{F}(A,B) = 1 + (A \times B)$ we get

$$\mathsf{F}^+(A,B) \;=\; (1 + (1 \times B)) \times A \;\cong\; A + (A \times B).$$

So labelled cons-lists are isomorphic to the type $listr^+$ of non-empty cons lists.

2. *Non-empty cons lists.* What happens when we try and label non-empty cons lists? Here $\mathsf{F}(A,B) = A + (A \times B)$ and so

$$\mathsf{F}^+(A,B) \;=\; (1 + (1 \times B)) \times A \;\cong\; A + (A \times B),$$

so labelling does not change non-empty cons lists in any essential way.

3. *Binary trees.* With $\mathsf{F}(A,B) = 1 + (A \times B \times B)$, we find

$$\mathsf{F}^+(A,B) \;=\; (1 + (1 \times B \times B)) \times A \;\cong\; A + (A \times B \times B).$$

Labelled trees are therefore isomorphic to the type

$$tree^+\,A \quad ::= \quad tip\,A \mid fork^+\,(A, tree^+\,A, tree^+\,A)$$

of non-empty labelled trees.

### 8.2 Subterms

We can now define $subterms : term^+\,(term\,A) \leftarrow term\,A$ as the catamorphism

$$subterms \quad = \quad (\![acc\,\alpha]\!), \tag{9}$$

where for $R : B \leftarrow \mathsf{F}(A,B)$ the relation $acc\,R$ (short for 'accumulate with $R$') has type

$$acc\,R \;:\; term^+\,B \leftarrow \mathsf{F}(A, term^+\,B)$$

and is given by the complicated expression

$$acc\,R \quad = \quad \alpha^+ \cdot \langle \mathsf{F}(!, id), R \cdot \mathsf{F}(id, root^+)\rangle,$$

and where $root^+ : term\,A \leftarrow term^+\,(term\,A)$ is the (total!) function $root = outl \cdot (\alpha^+)^\circ$. The definition is somewhat opaque, so we will give some examples.

1. *Snoc lists.* We have just seen that labelled snoc lists are isomorphic to the type

$$listl^+\,A \quad ::= \quad wrap\,A \mid snoc^+\,(listl^+\,A, A)$$

of non-empty snoc-lists. Here, $root^+ = last$, the (total!) function that returns the last element of a list of type $listl^+\,A$. Let $\sigma$ denote the isomorphism

$$\sigma \;:\; A + (B \times A) \leftarrow (1 + (1 \times B)) \times A.$$

Now we have

$$\begin{array}{ll} & acc\,\alpha \\ = & \quad \{\text{definitions, and } \mathsf{F}(A,B) = 1 + (B \times A)\} \\ & [wrap, snoc^+] \cdot \sigma \cdot \langle id + (id \times !), [nil, snoc] \cdot (id + last \times id)\rangle \\ = & \quad \{\text{coproduct law}\} \end{array}$$

https://doi.org/10.1017/S0956796800001556 Published online by Cambridge University Press

$$
\begin{aligned}
& [wrap, snoc^+] \cdot \sigma \cdot \langle id + (id \times !), [nil, snoc \cdot (last \times id)] \rangle \\
= \quad & \quad \{\text{property of } \sigma\} \\
& [wrap, snoc^+] \cdot (nil + \langle id, snoc \cdot (last \times id) \rangle) \\
= \quad & \quad \{\text{coproduct law}\} \\
& [wrap \cdot nil, snoc^+ \cdot \langle id, snoc \cdot (last \times id) \rangle].
\end{aligned}
$$

The coproduct law used above is that $[R, S] \cdot (T + U) = [R \cdot T, S \cdot U]$. Writing $[nil]$ for the constant returned by $wrap \cdot nil$ and $snoc$ for $snoc^+$ (thereby embedding the type $listl^+ A$ in $listl\ A$), we get

$$
\begin{aligned}
subterms &= foldl\,([nil], f) \\
f\,(xs, a) &= snoc\,(xs, snoc\,(last\ xs, a)).
\end{aligned}
$$

This is precisely the definition of the function *inits* we met in section 3.

2. *Binary trees.* Here labelled trees are isomorphic to the type

$$
tree^+ A \quad ::= \quad tip\ A \mid fork^+ (A, tree^+ A, tree^+ A)
$$

of non-empty labelled trees. This time $root^+$ is the total function that returns the element at the root of the tree. As before, we can calculate

$$
acc\ \alpha \quad = \quad [tip \cdot nil, fork^+ \cdot \langle id \times id, fork \cdot (id \times root^+ \times root^+) \rangle].
$$

As in the case of snoc-lists we can embed $tree^+ A$ in $tree\ A$ by taking $tip\ a = fork\,(a, nil, nil)$. The result is a computation for *subterms* that a functional programmer would write in the form

$$
\begin{aligned}
subterms &= foldtree\,(c, f) \\
c &= fork\,(nil, nil, nil) \\
f\,(a, x, y) &= fork\,(fork\,(a, root\ x, root\ y), x, y) \\
\\
root\,(fork\,(a, x, y)) &= a \\
\\
foldtree\,(c, f)\, nil &= c \\
foldtree\,(c, f)\,(fork\,(a, x, y)) &= f\,(a, foldtree\,(c, f)\, x, foldtree\,(c, f)\, y)
\end{aligned}
$$

### 8.3 *Properties of subterms*

Let us now look at two properties of *subterms*. First, *subterms* is a function because only functions appear in its definition (recall that $root^+ = outl \cdot (\alpha^+)^\circ$ is a total function).

The second, more important fact is that

$$
setify \cdot subterms = \Lambda subterm, \qquad (10)
$$

where $subterm = spur^*$. The proof, which we omit, depends on the fact that $spur^*$ is

https://doi.org/10.1017/S0956796800001556 Published online by Cambridge University Press

the unique solution for $X$ of the equation

$$X \quad = \quad id \cup (X \cdot spur).$$

### 8.4 Scans and the scan lemma

Now let us return to scans and the scan lemma. We saw in section 3 that $inits = scanl\,(nil, snoc)$. The same is true of the general scan, which is simply $(\![acc\,R]\!)$. The scan lemma therefore takes the form

*Lemma 8.1*
(Scan lemma) For any $R : B \leftarrow \mathsf{F}(A, B)$ we have

$$term^+\,(\![R]\!) \cdot (\![acc\,\alpha]\!) \quad \supseteq \quad (\![acc\,R]\!).$$

*Proof*
The proof is an exercise in fusion. We get the required result by showing

$$term^+\,(\![R]\!) \cdot acc\,\alpha \quad \supseteq \quad acc\,R \cdot \mathsf{F}(id, term^+\,(\![R]\!)).$$

In the argument that follows we make use of some laws of products that we have not formally stated:

$$\begin{array}{ll}
 & term^+\,(\![R]\!) \cdot acc\,\alpha \\
= & \quad \{\text{definition of } acc\} \\
 & term^+\,(\![R]\!) \cdot \alpha^+ \cdot \langle \mathsf{F}(!, id), \alpha \cdot \mathsf{F}(id, root^+)\rangle \\
= & \quad \{\text{definition of type functor } term^+\} \\
 & \alpha^+ \cdot \mathsf{F}^+((\![R]\!), term^+\,(\![R]\!)) \cdot \langle \mathsf{F}(!, id), \alpha \cdot \mathsf{F}(id, root^+)\rangle \\
= & \quad \{\text{since } \mathsf{F}^+(A, B) = \mathsf{F}(1, B) \times A\} \\
 & \alpha^+ \cdot (\mathsf{F}(id_1, term^+\,(\![R]\!)) \times (\![R]\!)) \cdot \langle \mathsf{F}(!, id), \alpha \cdot \mathsf{F}(id, root^+)\rangle \\
= & \quad \{\text{since } (R \times S) \cdot \langle U, V\rangle = \langle R \cdot U, S \cdot V\rangle;\ \text{bifunctors}\} \\
 & \alpha^+ \cdot \langle \mathsf{F}(!, term^+\,(\![R]\!)), (\![R]\!) \cdot \alpha \cdot \mathsf{F}(id, root^+)\rangle \\
= & \quad \{\text{catamorphism } (\![R]\!);\ \text{bifunctors}\} \\
 & \alpha^+ \cdot \langle \mathsf{F}(!, term^+\,(\![R]\!)), R \cdot \mathsf{F}(id, (\![R]\!) \cdot root^+)\rangle \\
\supseteq & \quad \{\text{since } root^+ : id \hookleftarrow term^+\} \\
 & \alpha^+ \cdot \langle \mathsf{F}(!, term^+\,(\![R]\!)), R \cdot \mathsf{F}(id, root^+ \cdot term^+\,(\![R]\!))\rangle \\
\supseteq & \quad \{\text{bifunctors and } \langle X \cdot Z, Y \cdot Z\rangle \supseteq \langle X, Y\rangle \cdot Z\} \\
 & \alpha^+ \cdot \langle \mathsf{F}(!, id), R \cdot \mathsf{F}(id, root^+)\rangle \cdot \mathsf{F}(id, term^+\,(\![R]\!)) \\
= & \quad \{\text{definition of } acc\,R\} \\
 & acc\,R \cdot \mathsf{F}(id, term^+\,(\![R]\!)).
\end{array}$$

□

### 8.5 Deforestation

The structure built up by an scan is usually not the final result of a computation; in practice, the labelled term that results is often evaluated by a catamorphism. This means that the labelled term need never be built up as a whole, for one can merge the process of its construction and its evaluation. This technique is very common in functional programming; it has been called *deforestation* by Wadler (1990), and Swierstra and De Moor (1992) speak of *virtual data structures*. Our final lemma shows how deforestation can be used in the present context:

*Lemma 8.2*
(Deforestation) Let $R : B \leftarrow \mathsf{F}^+(C, B)$ and $S : C \leftarrow \mathsf{F}(A, C)$, so $(\!|R|\!) : B \leftarrow term^+\, C$ and $(\!|acc\, S|\!) : term^+\, C \leftarrow term\, A$. Then

$$(\!|R|\!) \cdot (\!|acc\, S|\!) \quad \supseteq \quad outl \cdot (\!|reduce\,(R, S)|\!),$$

where $reduce\,(R, S) : (B \times C) \leftarrow \mathsf{F}(A, B \times C)$ is given by

$$reduce\,(R, S) \quad = \quad \langle R, outr\rangle \cdot \langle \mathsf{F}(!, outl), S \cdot \mathsf{F}(id, outr)\rangle.$$

Again the proof is an exercise in fusion, and we will not go into the details. If the final program is going to be evaluated in a lazy programming language, this lemma does not offer a real improvement in efficiency: the intermediate data structure in $(\!|S|\!) \cdot (\!|acc\, R|\!)$ never exists in its entirety anyway. It is probably for this reason that the above result was never stated in the theory of lists.

## 9 Segments and segment decomposition

Having dealt with all the necessary ingredients, we can now give the general version of the calculation given in section 3. First, we define $segment : term\, A \leftarrow term\, A$ by

$$segment \quad = \quad prune \cdot subterm.$$

Now we have

*Theorem 9.1*
(Segment Decomposition)

Let $\mathsf{F}$ be a pointed bifunctor with membership relations *left* and *right*. Suppose $f = [f_0, f_1] : B \leftarrow \mathsf{F}(A, B)$ is monotonic under the preorder $R : B \leftarrow B$. Then

$$max\, R \cdot \mathsf{P}(\!|f|\!) \cdot \Lambda segment \quad \supseteq \quad outl \cdot (\!|reduce\,(S, T)|\!),$$

where $S = max\, R \cdot \Lambda(left \cup right)$ and $T = max\, R \cdot \Lambda[f_0, f_0 \cup f_1]$.

*Proof*
We argue:

$$\begin{array}{cl} & max\, R \cdot \mathsf{P}(\!|f|\!) \cdot \Lambda segment \\ = & \quad \{\text{definition of } segment\} \\ & max\, R \cdot \mathsf{P}(\!|f|\!) \cdot \Lambda(prune \cdot subterm) \end{array}$$

https://doi.org/10.1017/S0956796800001556 Published online by Cambridge University Press

$$
\begin{aligned}
&= \quad \{(3)\} \\
&\quad max\ R \cdot \mathsf{P}(\![f]\!) \cdot union \cdot \mathsf{P}\Lambda prune \cdot \Lambda subterm \\
&= \quad \{(4) \text{ and functors}\} \\
&\quad max\ R \cdot union \cdot \mathsf{P}(\mathsf{P}(\![f]\!) \cdot \Lambda prune) \cdot \Lambda subterm \\
&\supseteq \quad \{(6) \text{ and functors}\} \\
&\quad max\ R \cdot \mathsf{P}(max\ R \cdot \mathsf{P}(\![f]\!) \cdot \Lambda prune) \cdot \Lambda subterm \\
&\supseteq \quad \{\text{Horner's rule with } X = [f_0, f_0 \cup f_1]\} \\
&\quad max\ R \cdot \mathsf{P}(\![max\ R \cdot \Lambda X]\!) \cdot \Lambda subterm \\
&= \quad \{(10)\} \\
&\quad max\ R \cdot \mathsf{P}(\![max\ R \cdot \Lambda X]\!) \cdot setify \cdot subterms \\
&\supseteq \quad \{(7)\} \\
&\quad max\ R \cdot setify \cdot term^{+}\,(\![max\ R \cdot \Lambda X]\!) \cdot subterms \\
&\supseteq \quad \{\text{Scan Lemma}\} \\
&\quad max\ R \cdot setify \cdot (\![acc\,(max\ R \cdot \Lambda X)]\!) \\
&\supseteq \quad \{(8)\} \\
&\quad (\![max\ R \cdot \Lambda(left \cup right)]\!) \cdot (\![acc\,(max\ R \cdot \Lambda X)]\!) \\
&\supseteq \quad \{\text{Deforestation Lemma, definitions of } S \text{ and } T\} \\
&\quad outl \cdot (\![reduce\,(S, T)]\!).
\end{aligned}
$$

□

Application of the segment decomposition theorem gives an efficient solution for the maximum segment sum problem on any type that allows the definition of *sum*.

## 10 Concluding remarks

We have demonstrated how much of the original theory of lists can be parameterised by an arbitrary data type. The result is, in our opinion, at least a linguistic improvement; the theory is no longer cluttered by the syntactic idiosyncracies of lists. It is debatable, however, whether by itself any mere linguistic improvement would justify the flood of definitions and results given above. What is of greater interest is the possibility that this style of generic programming can be applied to more challenging problems. An obvious candidate for further work is the so-called *sliding tails* lemma, which underlies all efficient pattern matching algorithms on lists. If this lemma can be parameterised by an arbitrary data type, the way is open for a generic theory of pattern matching. Such a generic theory is likely to benefit by the work of Backhouse (1992), who has shown how many theorems about regular algebra can be generalised to datatypes.

Finally, another important direction for future research is the design of a programming language in which data types are first-class citizens, in the sense that they can be passed as parameters to generic programs. It seems that research in the design of functional programming languages is also heading in this direction; in particular the work of Jones (1993) on *constructor classes* is relevant in this connection.

## Acknowledgements

Part of this work was done while Oege de Moor visited Roland Backhouse at Eindhoven University. Oege de Moor also wishes to thank Masato Takeichi for providing an inspiring working environment at Tokyo University, where this paper was finished. Many of the ideas and examples presented here are implicit in the work of Jeuring and Gibbons; the influence of their pioneering efforts can be traced throughout the paper. Both Jeuring and Jaap Van der Woude scrutinized drafts of this paper and suggested many improvements. The authors are also grateful to the anonymous referees for the same reason.

## References

Aarts, C. J., Backhouse, R. C., Hoogendijk, P., Voermans, E. and Van der Woude, J. C. S. P. (1992) *A relational theory of datatypes.* Available via anonymous ftp from `ftp.win.tue.nl` in directory `pub/math.prog.construction`.

Barr, M. and Wells, C. (1985) *Toposes, triples and theories.* Grundlehren der Math. Wissenschaften, 278. Springer-Verlag.

Barr, M. and Wells, C. (1990). *Category Theory for Computing Science.* Prentice–Hall.

Bird, R. S. (1987) An introduction to the theory of lists. In: Broy, M. (ed), *Logic of Programming and Calculi of Discrete Design.* NATO ASI Series F, **36**, pp. 3–42. Springer-Verlag.

Bird, R. S. (1989a) Lectures on constructive functional programming. In: Broy, M. (ed), *Constructive Methods in Computing Science.* NATO ASI Series F, **55**, pp. 151–216. Springer-Verlag.

Bird, R. S. (1990) A calculus of functions for program derivation. In: Turner, D. A. (ed), *Research Topics in Functional Programming*, pp. 287–308. University of Texas at Austin Year of Programming Series. Addison-Wesley.

Bird, R. and de Moor, O. (1996) *The Algebra of Programming.* Prentice Hall International.

Bird, R. S. (1989b) Algebraic identities for program calculation. *The Computer J.*, **32**: 122–126.

Carboni, A., Kelly, G. M. and Wood, R. J. (1991) A 2-categorical approach to geometric morphisms I. *Cahiers de topologie et geometrie differentielle categoriques*, **32**(1): 47–95.

De Moor, O. (1992) *Categories, relations and dynamic programming.* DPhil thesis. Technical Monograph PRG-98. Computing Laboratory, Oxford University.

De Moor, O. (1993) *Working notes on membership of data types.* Unpublished manuscript.

Freyd, P. J. and Ščedrov, A. (1990) *Categories, allegories.* Mathematical Library, vol. 39. North-Holland.

Gibbons, J. (1991) *Algebras for tree algorithms.* DPhil thesis, Programming Research Group, Computing Laboratory, Oxford University.

Goldblatt, R. (1986) *Topoi – the categorial analysis of logic.* Studies in Logic and the Foundations of Mathematics, vol. 98. North-Holland.

Jeuring, J. (1989) Deriving algorithms on binary labelled trees. In: Apers, P. M. G., Bosman, D. and Van Leeuwen, J. (eds), *Proc. sion Computing Science in the Netherlands*, pp. 229–249.

Johnstone, P. T. (1977) *Topos Theory.* Academic Press.

Jones, M. P. (1993) A system of constructor classes: overloading and implicit higher-order polymorphism. *Proc. FPCA*. To appear.

Lehmann, D. J. and Smyth, M. B. (1981) Algebraic specification of data types: A synthetic approach. *Math. Systems Theory*, **14**: 97–139.

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

Manes, E. G. and Arbib, M. A. (1986) *Algebraic Approaches to Program Semantics*. Texts and Monographs in Computer Science. Springer-Verlag.

Pierce, B. C. (1991) *Basic Category Theory for Computer Scientists*. MIT Press.

Swierstra, S. D. and De Moor, O. (1992) Virtual data structures. *Proc. State-of-the-Art Seminar on Formal Program Development*, Rio de Janeiro, Brazil (to appear).

Wadler, P. (1990) Deforestation. *Theoretical Computer Science*, **73**(2): 231–248.