Generic programming with relations and functors

Richard Bird and Oege de Moor
Programming Research Group, 11 Keble Road
Oxford OX1 3QD, United Kingdom

Paul Hoogendijk
Eindhoven University of Technology, PO Box 513
5600 MB Eindhoven, The Netherlands

October 25, 1999

Abstract

This paper explores the idea of generic programming in which programs are parameterised by data types. Part of the constructive theory of lists, specifically the part dealing with properties of segments, is generalised in two ways: from lists to arbitrary inductive data types, and from functions to relations. The new theory is used to solve a generic problem about segments.

1 Introduction

To what extent is it possible to construct programs without knowing exactly what data types are involved? At first sight this may seem a strange question, but consider the case 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 data types, 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 data type, to solve the general problem of pattern matching?

In this paper we give a positive answer to the above question, though for a problem somewhat simpler than pattern matching. The problem, the well-known maximum segment sum, was chosen because sufficient list theory exists [4, 5, 6] for one to calculate an efficient solution in a few equational steps. It turns out that we can generalise the theory of segments to more or less arbitrary data types, so the calculation leads to a generic solution to the problem.

In order 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 data types [18, 19, 17]. In the categorical approach, data types are characterised in terms

of certain functors, and specifications can be parameterised by functors in a simple and direct manner.

The second ingredient involves the move from functions to arbitrary relations [1, 8]. Introducing relations enables us to deal more smoothly with non-deterministic specifications, but it also turns out that the calculus of relations leads to substantial simplifications in the study of general data types.

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. This will provide the motivation for the formal definitions in the sequel. After that, in Sections 3 and 4 we review the calculus of relations and data types. Then in Section 5 we show how every data type comes equipped with a *membership* relation for testing whether or not an element occurs in a given data structure. These membership relations are then used in Sections 6–8 to develop a general theory of segments. In particular, we give suitably generalised statements and proofs of some results at the heart of the original theory of lists. As an application, we then obtain a generic solution to the maximum segment sum problem. Finally, Section 9 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

$$\mathsf{list}\, a \quad ::= \quad nil \mid snoc\,(\mathsf{list}\, a)\, a.$$

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

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

Thinking of lists purely as trees, we see that a prefix of x is really the same thing as a *subtree* of x. The function *subtrees* takes a list and returns the set of all its subtrees:

$$\begin{array}{rcl} subtrees\, nil & = & \{nil\} \\ subtrees\,(snoc\, x\, a) & = & subtrees\, x \cup \{snoc\, x\, a\}. \end{array}$$

In the theory of lists, prefixes are called *initial segments*, and the function *subtrees* 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 due course).

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 subtree of x. For instance, $[3, 4]$ is obtained from $[1, 2, 3, 4]$ by replacing $[1, 2]$ by the empty list. For the sake of a word we can say that the subtree $[1, 2]$ has been *pruned*. The function *prunings* takes a list and returns all ways in which it can be pruned:

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

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

One can now define arbitrary segments by the equation:

$$segments \quad = \quad union \cdot map\, prunings \cdot subtrees.$$

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. We can also define segments in terms of list concatenation, but that definition does not generalise to other data types.

For comparison, consider now the other view of lists, given by the type declaration

$$\mathsf{list}\, a \quad ::= \quad nil \mid cons\, a\, (\mathsf{list}\, a).$$

With this data type the role of *inits* and *tails* are reversed: *subtrees* 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.

Pursuing the same theme, now consider binary trees, as defined by

$$\mathsf{bintree}\, a \quad ::= \quad nil \mid bin\, a\, (\mathsf{bintree}\, a)\, (\mathsf{bintree}\, a).$$

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

$$\begin{array}{rcl} subtrees\, nil & = & \{nil\} \\ subtrees\, (bin\, a\, x\, y) & = & subtrees\, x \cup subtrees\, y \cup \{bin\, a\, x\, y\}. \end{array}$$

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

$$\begin{array}{rcl} prunings\, nil & = & \{nil\} \\ prunings\, (bin\, a\, x\, y) & = & \{nil\} \cup \{bin\, a\, s\, t \mid s \in prunings\, x, t \in prunings\, y\}. \end{array}$$

The segments of a tree are defined by the same equation as before. Jeuring [13] also considered such a definition, though he spoke of *treecuts* rather than segments.

3 A calculus of relations

Let us now give a brief review of the relational calculus used in the sequel. The calculus is based on Freyd's theory of allegories [10]; basically, an allegory is a category with additional axioms designed to capture the essential facts about relations. However, we will not take a fully axiomatic approach, relying instead on appeal to naive set theory, though we do assume a nodding acquaintance with categories, functors and natural transformations.

Relations The basis of the calculus is a category *Rel* whose objects are sets and whose arrows are relations. Arrows go backwards: we write $R : A \leftarrow B$ to denote that R is a relation of type 'A from B' and we can think of R as a subset of $A \times B$. Relational composition, like its functional counterpart, also goes backwards: $R \cdot S$ is pronounced 'R

after S'. Composition is associative with the identity relation $id : A \leftarrow A$ as unit, all of which says *Rel* is a category. For any A and B there exists a smallest relation $0 : A \leftarrow B$, which is the zero of composition, and a largest relation $\Pi : A \leftarrow B$ which is essentially the cartesian product $A \times B$.

Each relation $R : A \leftarrow B$ has a *converse* relation $R^\circ : B \leftarrow A$, which preserves identities but reverses composition (so $(R \cdot S)^\circ = S^\circ \cdot R^\circ$), all of which says that converse is a contravariant functor from *Rel* to itself. By assumption, the converse functor is its own inverse, $(R^\circ)^\circ = R$, and so is an isomorphism.

For each A and B the arrows $A \leftarrow B$ form a complete lattice with union $\cup$ and intersection $\cap$. These arrows can be compared via a partial order $\subseteq$, where $R \subseteq S$ denotes $R = R \cap S$. 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:

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

Again, taking converses, we get the symmetric version

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

Division Because relational composition distributes over arbitrary unions, it has a weak inverse, called *division*, which is characterised by the equivalence

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

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

Entire and simple relations There are three subcategories of *Rel* of particular interest: the entire (or total) relations, the simple (or single-valued) relations, and functions (which by definition are those relations that are both entire and simple). A relation $R : A \leftarrow B$ is *entire* if $id \subseteq R^\circ \cdot R$, and *simple* if $R \cdot R^\circ \subseteq id$. Identity arrows are both entire and simple, and composition preserves both properties, so both kinds of relations form subcategories of *Rel*. It follows that the category *Fun* of functions is also a subcategory of *Rel*. Functions will be denoted by lower case identifiers, $f, g, h, \ldots$.

To illustrate these definitions, let us prove that

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

for all functions f:

$$\begin{array}{ll} & (R \cap S) \cdot f \\ \subseteq & \quad \{\text{distributing composition over } \cap\} \\ & (R \cdot f) \cap (S \cdot f) \\ \subseteq & \quad \{\text{modular law}\} \\ & ((R \cdot f \cdot f^\circ) \cap S) \cdot f \\ \subseteq & \quad \{\text{monotonicity of composition, since } f \cdot f^\circ \subseteq id\} \\ & (R \cap S) \cdot f. \end{array}$$

Note that we used only the fact that f was simple.

Another useful result that applies to functions only is the *shunting rule*:

$$\begin{array}{lcl} f \cdot R \subseteq S & \equiv & R \subseteq f^\circ \cdot S \\ R \subseteq S \cdot f & \equiv & R \cdot f^\circ \subseteq S. \end{array}$$

The proof is an easy consequence of the definition of a function as an entire and simple relation.

Relators Functors will be denoted using sans serif letters. A functor $\mathsf{F} : Rel \leftarrow Rel$ is said to be *monotonic* if $R \subseteq S$ implies $\mathsf{F}R \subseteq \mathsf{F}S$. Monotonic functors have many nice algebraic properties, most of which can be derived from the fact that they preserve converse, that is, $\mathsf{F}(R^\circ) = (\mathsf{F}R)^\circ$. Monotonic functors take entire relations to entire relations, simple relations to simple ones, and so functions to functions. Furthermore, any functor $\mathsf{F} : Fun \leftarrow Fun$ has at most one monotonic extension $\mathsf{F} : Rel \leftarrow Rel$ that coincides with F on functions. More precisely, write $\mathsf{J} : Rel \leftarrow Fun$ for the inclusion functor of functions into relations. Then for all monotonic functors $\mathsf{F}, \mathsf{G} : Rel \leftarrow Rel$ we have

$$(\mathsf{F} = \mathsf{G}) \quad \equiv \quad (\mathsf{F} \cdot \mathsf{J} = \mathsf{G} \cdot \mathsf{J}).$$

For a proof see [7]. Following Backhouse [1], we will call an endofunctor of *Fun* a *relator* if it has a monotonic extension in *Rel*, and also use this term simply as an abbreviation for a monotonic functor of *Rel*.

Natural transformations In the relational calculus it is necessary to distinguish between two kinds of natural transformation. The first is the standard notion of a natural transformation in category theory. That is, given two functors F and G, we have

$$\phi : \mathsf{F} \leftarrow \mathsf{G} \quad \equiv \quad (\forall R :: \mathsf{F}R \cdot \phi = \phi \cdot \mathsf{G}R).$$

Such natural transformations will be called *proper*. In the second notion of natural transformation, the equation is weakened to an inequality:

$$\phi : \mathsf{F} \leftarrow\!\!\!\shortmid \mathsf{G} \quad \equiv \quad (\forall R :: \mathsf{F}R \cdot \phi \supseteq \phi \cdot \mathsf{G}R).$$

Since this weak type of natural transformation is more common in the relational calculus than the proper one, we shall simply call them natural transformations (the usual term is *weak*, or *lax*, natural transformation). Every natural transformation is proper when restricted to functions, that is,

$$(\phi : \mathsf{F} \leftarrow\!\!\!\shortmid \mathsf{G}) \quad \equiv \quad (\phi : \mathsf{F} \cdot \mathsf{J} \leftarrow \mathsf{G} \cdot \mathsf{J}).$$

Again the proof can be found in [7]. Note that ϕ need not itself be a function.

Powersets The view of relations adopted here is essentially that a relation $A \leftarrow B$ is a subset of the cartesian product $A \times B$. However, one can also view relations as functions $\mathsf{P}A \leftarrow B$, where $\mathsf{P}A$ denotes the powerset of A. Formally, the isomorphism between these two representations of relations 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. The powerset $\mathsf{P}A$ and the relation $\in$ 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 ΛR is said to be the *power transpose* of R and can be defined in set theory by $(\Lambda R)\, b = \{a \,|\, aRb\}$. 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, the theory of *toposes* [14, 3, 12]. Below we illustrate how various familiar operators from set theory can be defined in terms of Λ.

First of all, it is immediate from the universal property of Λ that $id : \mathsf{P}A \leftarrow \mathsf{P}A$ satisfies $id = \Lambda(\in)$. Next, the *existential image* of $R : A \leftarrow B$ is a function $\mathsf{E}R : \mathsf{P}A \leftarrow \mathsf{P}B$ defined by $\mathsf{E}R = \Lambda(R \cdot \in)$. In set theory we have

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

We have $\mathsf{E}id = id$, and below we will show that $\mathsf{E}(R \cdot S) = \mathsf{E}R \cdot \mathsf{E}S$, so E is a functor. It is not, however, a relator because it is not monotonic: inclusion of functions is equality.

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

$$\begin{array}{ll} & \Lambda(R \cdot S) = \mathsf{E}R \cdot \Lambda S \\ \equiv & \quad \{\text{definition of } \Lambda\} \end{array}$$

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

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

From the definition of E (plus the fact that $\in$ cancels Λ) we get $R \cdot \in = \in \cdot \mathsf{E}R$, which says that $\in : \mathsf{id} \leftarrow \mathsf{E}$ is a proper natural transformation.

Although E is not a relator, there does exist a variant of E which is. The restriction of E to functions is called P, so $\mathsf{P} = \mathsf{E} \cdot \mathsf{J}$. The functor $\mathsf{P} : \mathit{Fun} \leftarrow \mathit{Fun}$ is a relator and its unique extension to relations turns out to be

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

where $\ni$ denotes the converse of $\in$. Using the pointwise interpretation of the division operators, this formula reads

$$
\begin{array}{lll}
x(\mathsf{P}R)y & \equiv & (\forall a \in x : \exists b \in y : aRb) \wedge \\
 & & (\forall b \in y : \exists a \in x : aRb).
\end{array}
$$

Since $\in : \mathsf{id} \leftarrow \mathsf{E}$ is proper, we get that $\in$ is a natural transformation $\mathsf{id} \hookleftarrow \mathsf{P}$, though not a proper one.

For future use, we note that E can be expressed in terms of P:

$$\mathsf{E}R \quad = \quad \mathit{union} \cdot \mathsf{P}\Lambda R, \tag{1}$$

where $\mathit{union} : \mathsf{P}A \leftarrow \mathsf{PP}A$ is defined by $\mathit{union} = \mathsf{E}(\in)$. This function returns the union of a collection of sets. Since $\in : \mathsf{id} \leftarrow \mathsf{E}$ we have $\mathit{union} : \mathsf{E} \leftarrow \mathsf{EE}$. A more detailed discussion of P and its relation to E can be found in [8].

Preorders By definition, a *preorder* is a relation $R : A \leftarrow A$ which is both reflexive ($\mathit{id} \subseteq R$) and transitive ($R \cdot R \subseteq R$). For any relation R there exists a smallest preorder R^* containing R, defined as the least solution $X = R^*$ of

$$X \quad = \quad \mathit{id} \cup (X \cdot R).$$

The relation R^* is commonly known as the *reflexive transitive closure* of R. The following property will be useful in the sequel:

$$R \cdot S = S \cdot R \quad \Rightarrow \quad R^* \cdot S = S \cdot R^*. \tag{2}$$

Maximum Finally, we consider the *maximum* relation $\mathit{max}R : A \leftarrow \mathsf{P}A$ associated with a given preorder $R : A \leftarrow A$, defined by

$$\mathit{max}R \quad = \quad \in \cap (\in \backslash R)^{\circ}.$$

This definition corresponds to the usual definition of maximum elements in set theory: $a(\mathit{max}R)x$ 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. Note that, although the definition of $maxR$ does not depend on R being a preorder, it is useful only when R is one, so we shall assume without stating it explicitly that R is a preorder whenever the construction $maxR$ is considered.

There are two properties of max that we will need in Section 7. First of all,

$$\begin{array}{ll} & X \subseteq maxR \cdot \Lambda S \\ \equiv & \quad \{\text{definition of } maxR\} \\ & X \subseteq (\in \cap (\in \backslash R)^\circ) \cdot \Lambda S \\ \equiv & \quad \{\text{functions distribute backwards over } \cap\} \\ & X \subseteq (\in \cdot \Lambda S) \cap ((\in \backslash R)^\circ \cdot \Lambda S) \\ \equiv & \quad \{\Lambda \text{ cancellation and universal property of } \cap\} \\ & (X \subseteq S) \ \wedge \ (X \subseteq (\in \backslash R)^\circ \cdot \Lambda S). \end{array}$$

Continuing with the second term

$$\begin{array}{ll} & X \subseteq (\in \backslash R)^\circ \cdot \Lambda S \\ \equiv & \quad \{\text{shunting}\} \\ & X \cdot (\Lambda S)^\circ \subseteq (\in \backslash R)^\circ \\ \equiv & \quad \{\text{converse}\} \\ & \Lambda S \cdot X^\circ \subseteq \in \backslash R \\ \equiv & \quad \{\text{universal property of } \backslash\} \\ & \in \cdot \Lambda S \cdot X^\circ \subseteq R \\ \equiv & \quad \{\Lambda \text{ cancellation}\} \\ & S \cdot X^\circ \subseteq R \\ \equiv & \quad \{\text{converse}\} \\ & X \cdot S^\circ \subseteq R^\circ. \end{array}$$

Hence $X \subseteq maxR \cdot \Lambda S$ if and only if $X \subseteq S$ and $X \cdot S^\circ \subseteq R^\circ$. We call this property the universal property of max. We have given the proof in details because it is fairly typical of the kind of manipulations found in the relational calculus.

The second fact, which depends on R being a preorder, is that $maxR$ weakly distributes over $union$:

$$maxR \cdot union \quad \supseteq \quad maxR \cdot \mathsf{P}(maxR), \quad \text{provided } R \text{ is a preorder.} \tag{3}$$

To see that this inequation cannot be strengthened to an equality, consider some singleton set $A = \{a\}$ with the trivial preorder $R = id$. With $x = \{\{a\}, \{\}\}$ we have $a(maxR \cdot union)x$ but not $a(maxR \cdot \mathsf{P}(maxR))x$ since $(maxR)\{\,\}$ has empty range.

4 Types

Our approach to data types is based on the idea that every type constructor, such as *list* or *tree*, is associated with a functor, such as list or tree, that applies a function to all elements

of the type. Such functors correspond to the *map* operators in functional programming. This view of data types is also the basis of the work by Backhouse [1] to which the interested reader is referred for a more detailed discussion of the concepts introduced here. We begin by considering certain basic types.

Terminator The terminator 1 is a set with one element. It has the property that for every set A there exists precisely one function $1 \leftarrow A$, denoted by !. Note that, although 1 is indeed a final object in *Fun*, it is not a final object in *Rel* since $0 : 1 \leftarrow A$ and $0 \neq !$. The functor that comes with the terminator is the constant functor K_1 which maps all sets to 1, and all arrows to $id : 1 \leftarrow 1$. More generally, the constant functor K_A maps all sets to A and all arrows to $id : A \leftarrow A$.

Product Recall that in a category an object $A \times B$ with two arrows $outl : A \leftarrow A \times B$ and $outr : B \leftarrow A \times B$ is called a *product* if for all C and arrows $f : A \leftarrow C$, $g : B \leftarrow C$ there is a unique arrow $\langle f, g\rangle : A \times B \leftarrow C$ such that $outl \cdot \langle f, g\rangle = f$ and $outr \cdot \langle f, g\rangle = g$. The category *Fun* has products and the product functor $\times$ is defined as usual by

$$f \times g \;=\; \langle f \cdot outl, g \cdot outr\rangle.$$

This functor is a relator and we have $R \times S = \langle R \cdot outl, S \cdot outr\rangle$, where

$$\langle R, S\rangle \;=\; (outl^\circ \cdot R) \cap (outr^\circ \cdot S).$$

However, $outl$, $outr$ and $\langle -, -\rangle$ do not define a categorical product in *Rel* since, for example, $outl \cdot \langle R, 0\rangle = 0$ for all R. This is not a problem, for we said only that functional product has a unique extension in *Rel*, not that this extension should also be a product in *Rel*. We do have $outl \cdot \langle R, S\rangle = R$ and $outr \cdot \langle S, R\rangle = R$ whenever S is entire and so $outl \cdot (R \times S) = R \cdot outl$ and $outr \cdot (S \times R) = R \cdot outr$ whenever S is entire.

Coproducts Dually, in a category an object $A + B$ with two arrows $inl : A + B \leftarrow A$ and $inr : A + B \leftarrow B$ is a *coproduct* if for all C and arrows $f : C \leftarrow A$, $g : C \leftarrow B$ there is a unique arrow $[f, g] : C \leftarrow A + B$ such that $[f, g] \cdot inl = f$ and $[f, g] \cdot inr = g$. The category *Fun* has coproducts and the coproduct functor $+$ is defined as usual by

$$f + g \;=\; [inl \cdot f, inr \cdot g].$$

The coproduct is also a relator and $R + S = [inl \cdot R, inr \cdot S]$, where

$$[R, S] \;=\; (R \cdot inl^\circ) \cup (S \cdot inr^\circ).$$

Unlike the situation with products, inl, inr and $[-, -]$ do form a proper coproduct in *Rel*. For example, $[R, S] \cdot inl = R$ for all S, and from this it follows that

$$[R, S] \cdot [U, V]^\circ \;=\; R \cdot U^\circ \;\cup\; S \cdot V^\circ$$

which will be needed below.

Polynomial Relators Relators built up from constants, finite products and coproducts are said to be *polynomial*. More precisely, the class of polynomial relators is defined inductively by the following clauses:

1. The identity relator id and the constant relators K_A are polynomial;
2. if F and G are polynomial, then so are their composition $\mathsf{F} \cdot \mathsf{G}$, their sum $\mathsf{F} + \mathsf{G}$ and their product $\mathsf{F} \times \mathsf{G}$, where

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

Catamorphisms and promotion Let F be a relator. By definition, an F–*algebra* is a relation of type $A \leftarrow \mathsf{F}A$, the set A being called the *carrier* of the algebra. A F–*homomorphism* from an algebra $S : B \leftarrow \mathsf{F}B$ to an algebra $R : A \leftarrow \mathsf{F}A$ is a relation $X : A \leftarrow B$ such that

$$X \cdot S \;=\; R \cdot \mathsf{F}X.$$

Identity arrows are homomorphisms, and the composition of two homomorphisms is again a homomorphism, so F–algebras form the objects of a category whose arrows are homomorphisms. For many relators (in particular, the polynomial ones), this category has an initial object, which we shall denote by $\alpha : T \leftarrow \mathsf{F}T$. For any other F–algebra $R : A \leftarrow \mathsf{F}A$ the unique homomorphism from α to R will be denoted by $(\!|R|\!)$, so $(\!|R|\!) : A \leftarrow T$ is characterised by

$$(X \cdot \alpha = R \cdot \mathsf{F}X) \;\equiv\; (X = (\!|R|\!)).$$

Homomorphisms of the form $(\!|R|\!)$ are called *catamorphisms* [18]. The initial algebra α is, in fact, an isomorphism [16] so we can rewrite the above equivalence in the form

$$(X = R \cdot \mathsf{F}X \cdot \alpha^\circ) \;\equiv\; (X = (\!|R|\!)).$$

The well-known Knaster-Tarski Fixpoint Theorem says that the unique solution (if it exists) of $X = F(X)$ is also the least solution of $X \supseteq F(X)$ and the greatest solution of $X \subseteq F(X)$, so we get the following results, known collectively as *promotion*:

$$\begin{aligned}X = R \cdot \mathsf{F}X \cdot \alpha^\circ &\;\equiv\; X = (\!|R|\!) \\ X \subseteq R \cdot \mathsf{F}X \cdot \alpha^\circ &\;\Rightarrow\; X \subseteq (\!|R|\!) \\ X \supseteq R \cdot \mathsf{F}X \cdot \alpha^\circ &\;\Rightarrow\; X \supseteq (\!|R|\!).\end{aligned}$$

The typical use of promotion is when $X = S \cdot (\!|T|\!)$. In particular, the following calculation gives a useful condition for expressing $S \cdot (\!|T|\!)$ as a catamorphism:

$$\begin{aligned}& S \cdot (\!|T|\!) = (\!|R|\!) \\ \equiv\quad & \{\text{promotion}\} \\ & S \cdot (\!|T|\!) \cdot \alpha = R \cdot \mathsf{F}(S \cdot (\!|T|\!)) \\ \equiv\quad & \{\text{definition of } (\!|T|\!)\} \\ & S \cdot T \cdot \mathsf{F}(\!|T|\!) = R \cdot \mathsf{F}(S \cdot (\!|T|\!)) \\ \Leftarrow\quad & \{\mathsf{F} \text{ is a functor}\} \\ & S \cdot T = R \cdot \mathsf{F}S.\end{aligned}$$

Use of this, or similar, conditions in calculations will be signalled with the hint 'promotion'.

Tree types defined Let us now return to tree types. Tree types are initial algebras and can be named as such by type declarations. For example, the declaration

$$\mathsf{list}\,A \quad ::= \quad nil \,|\, snoc(\mathsf{list}\,A, A)$$

declares $[nil, snoc] : \mathsf{list}\,A \leftarrow \mathsf{H}_A(\mathsf{list}\,A)$ to be the initial H_A–algebra, where $\mathsf{H}_A(X) = 1 + (X \times A)$ and $\mathsf{H}_A(f) = id_1 + (f \times id_A)$. We can and will write $\mathsf{H}(A, X)$ instead of $\mathsf{H}_A(X)$, in which case we think of H as a *bifunctor*. By fixing the left or right argument of a bifunctor we get two functors and, since we shall need both functors below, it is useful to settle on a consistent convention for naming them. For a bifunctor H we define

$$\begin{array}{lcl} \mathsf{H}_0(f) & = & \mathsf{H}(f, id_X) \\ \mathsf{H}_1(f) & = & \mathsf{H}(id_A, f). \end{array}$$

Note that dependence on X and A has been suppressed in this notation. Moreover, we will always arrange the arguments of a given bifunctor H so that it is H_1 that describes the initial algebra. With this convention, $[nil, snoc]$ is the initial H_1–algebra corresponding to the bifunctor $\mathsf{H}(A, X) = 1 + (X \times A)$.

Above we wrote list in sans serif font, which is our convention for denoting functors. This was intended: with every tree type T is associated a certain functor, which we shall call a *tree* functor. For example, the tree functor list is just the familiar *map* operation of functional programming. The function $\mathsf{list}f$ is defined over snoc lists by

$$\mathsf{list}f \quad = \quad (\!|nil, snoc \cdot (id \times f)|\!).$$

Note that we write $(\!|e, f|\!)$ rather than the more clumsy $(\!|[e, f]|\!)$. Using the characterisation of catamorphisms, this definition expands to the familiar recursion equations

$$\begin{array}{rcl} \mathsf{list}f\,[\,] & = & [\,] \\ \mathsf{list}f\,(snoc(x, a)) & = & snoc(\mathsf{list}f\,x, f\,a). \end{array}$$

In fact, $(\!|e, f|\!)$ translates to the standard higher-order function $foldlf\ e$.

Tree functors are relators and the case of lists illustrates how they are defined in general: given an initial algebra $\alpha : \mathsf{T}A \leftarrow \mathsf{H}(A, \mathsf{T}A)$, the tree functor T is defined by

$$\mathsf{T}R \quad = \quad (\!|\alpha \cdot \mathsf{H}_0 R|\!).$$

We describe this situation by saying H is a binary relator with *tree type* (α, T). Elements of a tree type are called *trees*. Note that the definition of T gives that $\alpha : \mathsf{T} \leftarrow \mathsf{G}$, where $\mathsf{G}R = \mathsf{H}(R, \mathsf{T}R)$, is a proper natural transformation.

5 Membership

Data types record the presence of elements, so one would expect relator F to come equipped with a *membership* relation $\in_{\mathsf{F}}$ such that $a \in_{\mathsf{F}} x$ precisely when a is an element of x. Indeed, this notion of membership is so common that its definition is usually taken for granted.

Formally, a collection of arrows $\in_{\mathsf{F}}$ is a *membership* relation of F if for each R,

$$\mathsf{F}R \cdot \in_{\mathsf{F}} \backslash id = \in_{\mathsf{F}} \backslash R.$$

The pointwise interpretation of this equation is

$$\begin{array}{l} \forall a : a \in_{\mathsf{F}} x : aRb \\ \equiv \\ \exists y : x(\mathsf{F}R)y \wedge (\forall b' : b' \in_{\mathsf{F}} y : b' = b). \end{array}$$

Our first result about membership says that the above equation has at most one solution. For a proof see [9] (where proofs of most of the following facts about membership can be found).

Lemma 1 *If $\in_{\mathsf{F}}$ is a membership relation of F, then $\in_{\mathsf{F}}$ is the largest natural transformation of type $\mathsf{id} \leftarrow \mathsf{F}$.*

To illustrate, consider the membership relation of the powerset functor, writing $\in$ instead of $\in_{\mathsf{P}}$. To say that $\in$ is a natural transformation is to say that $f \cdot \in = \in \cdot \mathsf{P}f$ for all functions f (recall that every natural transformation is proper when restricted to functions). Equivalently, for all a and x we have $a \in \mathsf{P}f\, x$ if and only if there exists $b \in x$ such that $f\, b = a$, which is a well known property in set theory. There exist relators that do not have a membership relation, but all relators in programming do have one. Below we show how to construct the membership relation of polynomial relators, and of tree relators.

Lemma 2 *The membership relation of a polynomial relator is given by the following clauses:*

$$\begin{array}{rcl} \in_{\mathsf{id}} & = & id \\ \in_{\mathsf{K}\,A} & = & 0 \\ \in_{\mathsf{F}+\mathsf{G}} & = & [\in_{\mathsf{F}}, \in_{\mathsf{G}}] \\ \in_{\mathsf{F}\times\mathsf{G}} & = & \in_{\mathsf{F}} \cdot outl \cup \in_{\mathsf{G}} \cdot outr \\ \in_{\mathsf{F}\cdot\mathsf{G}} & = & \in_{\mathsf{G}} \cdot \in_{\mathsf{F}} \end{array}$$

Not every relator is polynomial; in particular, the tree relator T of a tree type is not polynomial. Since T is defined by a catamorphism, one might expect that $\in_{\mathsf{T}}$ can also be expressed as a catamorphism, but this is not possible for data types containing constants such as the empty list. The reason is simple: membership reduces to the empty relation on empty lists, and the empty relation would propagate through any catamorphic definition, rendering such a definition useless. Fortunately, there is another solution for the definition of $\in_{\mathsf{T}}$, one which makes use of three auxiliary relations, which we will call *root*, *branch* and *subtree*.

Let H be a binary relator, and let (α, T) be its tree type. We assume that both H_0 and H_1 have a membership relation, and we shall write $\in_i$ instead of $\in_{\mathsf{H}_i}$. The natural transformation $root : \mathsf{id} \leftarrow \mathsf{T}$ returns an element that occurs at the root of a tree. It is defined by the equation

$$root \;=\; \in_0 \cdot \alpha^{\circ}.$$

Naturality of *root* follows from the theory of membership, see [9]. Similarly, the natural transformation $branch : \mathsf{T} \leftarrow \mathsf{T}$ returns an immediate subtree of its argument

$$branch \quad = \quad \in_1 \cdot \alpha^\circ.$$

Finally, the natural transformation $subtree : \mathsf{T} \leftarrow \mathsf{T}$ is defined by $subtree = branch^*$ and returns an arbitrary subtree of a given tree.

Below we shall list a number of examples to illustrate these definitions, but first we state the main result about membership of trees:

Lemma 3 *Let* H *be a binary relator with tree type* (α, T). *Then*

$$\in_{\mathsf{T}} \quad = \quad root \cdot subtree.$$

In words, this lemma says that a is an element of x if a occurs at the root of a subtree of x. This intuition is further explained in the following examples:

1. *Snoc lists.* With $\mathsf{H}(A, X) = 1 + (X \times A)$, we get the tree type $([empty, snoc], \mathsf{list})$ of snoc lists. Since

   $$\begin{aligned} \mathsf{H}_0 &= \mathsf{K}_1 + (\mathsf{K}_X \times \mathsf{id}) \\ \mathsf{H}_1 &= \mathsf{K}_1 + (\mathsf{id} \times \mathsf{K}_A), \end{aligned}$$

   we find, using Lemma 2, that

   $$\begin{aligned} \in_0 &= [0, outr] \\ \in_1 &= [0, outl]. \end{aligned}$$

   Hence

   $$root = \in_0 \cdot \alpha^\circ = [0, outr] \cdot [nil, snoc]^\circ = outr \cdot snoc^\circ$$

   and, similarly, $branch = outl \cdot snoc^\circ$. In other words, *root* is *last*, the partial function that returns the last element of a nonempty list, and *branch* is *init*, the partial function that removes the last element from a nonempty list. Lemma 3 says that a is an element of a list x iff a occurs as the last element of a prefix of x.

2. *Cons lists.* Dually, with the relator $\mathsf{H}(A, X) = 1 + (A \times X)$, we get the term type $([nil, cons], \mathsf{list})$ of cons lists. Here, *root* is *head* and returns the first element of a nonempty list, and *branch* is *tail*, returning the remainder.

3. *Binary trees.* With $\mathsf{H}(A, X) = 1 + (A \times (X \times X))$, we get the tree type $([nil, bin], \mathsf{bintree})$ described in Section 2. Here we have

   $$\begin{aligned} \mathsf{H}_0 &= \mathsf{K}_1 + (\mathsf{id} \times (\mathsf{K}_X \times \mathsf{K}_X)) \\ \mathsf{H}_1 &= \mathsf{K}_1 + (\mathsf{K}_A \times (\mathsf{id} \times \mathsf{id})). \end{aligned}$$

Hence

$$\begin{aligned} \in_0 &= [0, outl] \\ \in_1 &= [0, (outl \cup outr) \cdot outr], \end{aligned}$$

and so

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

The partial function *root* returns the label of a nonempty tree, and *branch* returns one of the subtrees. Lemma 3 says that a is an element of a tree x just in the case that a is the label of some subtree of x.

4. *Unlabelled binary trees.* Let us try another kind of tree, unlabelled binary trees with values at the tips. With $\mathsf{H}(A, X) = A + (X \times X)$ we get the tree type $([tip, bin], \mathsf{tree})$. This time we have

$$\begin{aligned} \mathsf{H}_0 &= \mathsf{id} + (\mathsf{K}_X \times \mathsf{K}_X) \\ \mathsf{H}_1 &= \mathsf{K}_A + (\mathsf{id} \times \mathsf{id}), \end{aligned}$$

and so

$$\begin{aligned} \in_0 &= [id, 0] \\ \in_1 &= [0, outl \cup outr]. \end{aligned}$$

Hence

$$\begin{aligned} root &= tip^\circ \\ branch &= (outl \cup outr) \cdot bin^\circ. \end{aligned}$$

Lemma 3 says that a is an element of a tree x just in the case that a occurs as a *tip* in x.

In the sequel, we shall use trees to represent sets. It will be necessary, therefore, to implement various operators on sets (such as the maximum relation) on trees. Formally, let H be a binary relator, and let (α, T) be its tree type. The function $setify : \mathsf{P} \leftarrow \mathsf{T}$ takes a tree and returns the set of its elements

$$setify = \Lambda \in_{\mathsf{T}}.$$

Note that *setify* is not a surjective function, because there are many sets that cannot be represented as a tree. For instance, no infinite set can be represented as a finite tree. The next lemma shows how one may implement the maximum operator on trees in terms of a simpler maximum operator, for instance on pairs.

Lemma 4 *Let* H *and* T *be as defined above, and let* R *be a preorder. Then*

$$maxR \cdot setify \supseteq (\!|maxR \cdot \Lambda X|\!),$$

where $X = \in_0 \cup \in_1$.

The containment cannot be strengthened to an equality because there may be constant trees that have no elements.

6 Subtrees and accumulations

The function $\Lambda subtree$ returns the set of subtrees of a given tree; in the case of snoc lists this gives the set of initial segments of a list. In functional programming there is an important and useful operation on initial segments, called *accumulation* and expressed by the higher-order function *scanl*. The key fact is the *accumulation lemma*, which says

$$map\,(foldl\,f\,e) \cdot inits \;=\; scanl\,f\,e.$$

Here, *inits* returns the list of initial segments of a list in ascending order of length:

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

The function *scanl* captures a common pattern of computation, the point of the accumulation lemma being that evaluation of

$$scanl\,(\oplus)\,e\,[a_1, a_2, \ldots, a_n] \;=\; [e, e \oplus a_1, (e \oplus a_1) \oplus a_2, \cdots ((e \oplus a_1) \oplus \cdots) \oplus a_n]$$

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

In an attempt to construct a generic version of the accumulation lemma, we might try and render the left-hand side as

$$\mathsf{list}\,([\![e, f]\!]) \cdot sort \cdot \Lambda subtree,$$

where *sort* orders a set of structures into ascending size. There are various problems with this idea, including the fact that it still involves lists in an essential way. Instead of a list of structures 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 subtree. Not every tree type allows the labelling of nodes (think of unlabelled binary trees), but there is a canonical way of introducing labels into a tree type. We consider this first, returning to accumulations at the end of the section.

Labelled tree types Let H be a binary relator, and (α, T) its tree type. Define another bifunctor H' by

$$\mathsf{H}'(A, X) \;=\; \mathsf{H}(1, X) \times A$$

and let (α', T') be its tree type. We call H' the *labelled* variant of H, and (α', T') the *labelled* tree type associated with (α, T). Let us consider some examples to clarify the idea.

1. *Snoc lists.* With $\mathsf{H}(A, X) = 1 + (X \times A)$ we get

   $$\mathsf{H}'(A, X) \;=\; (1 + (X \times 1)) \times A \;\cong\; A + (X \times A).$$

   So the labelled tree type is isomorphic to the type $([one, snoc], \mathsf{nelist})$ of nonempty snoc lists.

2. *Nonempty snoc lists.* What happens when we try and label nonempty snoc lists? Here $\mathsf{H}(A, X) = A + (X \times A)$ and so

$$\mathsf{H}'(A, X) \;=\; (1 + (X \times 1)) \times A \;\cong\; A + (X \times A).$$

Therefore labelling does not change nonempty snoc lists.

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

$$\mathsf{H}'(A, X) \;=\; (1 + (1 \times (X \times X))) \times A \;\cong\; A + ((X \times X) \times A).$$

The tree type of H' is thus isomorphic to $([\mathit{tip}, \mathit{node}], \mathsf{netree})$, the type of nonempty labelled binary trees.

4. *Unlabelled binary trees.* With $\mathsf{H}(A, X) = A + (X \times X)$, giving the tree type of unlabelled binary trees with values at the tips, we find

$$\mathsf{H}'(A, X) \;=\; (1 + (X \times X)) \times A \;\cong\; A + ((X \times X) \times A),$$

so the tree type of H' is isomorphic to that of the previous example.

Subtrees Now we can define *subtrees* as a structure of structures. Given a binary relator H with tree type (α, T) and labelled tree type (α', T'), the natural transformation $\mathit{subtrees} : \mathsf{T}' \cdot \mathsf{T} \hookleftarrow \mathsf{T}$ is defined by the equation

$$\mathit{subtrees} \;=\; (\!|\alpha' \cdot \langle \mathsf{H}_0!, \alpha \cdot \mathsf{H}_1 \mathit{root} \rangle|\!). \tag{4}$$

We will show below that *subtrees* is a function; it takes a tree and turns it into a tree of the same shape in which every node is labelled by its subtree. The somewhat complicated definition is explained with the help of the following examples.

1. *Snoc lists.* We have just seen that for snoc lists, (α', T) is isomorphic to the tree type $([\mathit{one}, \mathit{snoc}], \mathsf{nelist})$ of non-empty lists. Let σ denote the obvious isomorphism from $(1 + (1 \times X)) \times A$ to $A + (X \times A)$, so $\alpha' = [\mathit{one}, \mathit{snoc}] \cdot \sigma$. Now we have

$$\begin{array}{ll} & \alpha' \cdot \langle \mathsf{H}_0!, \alpha \cdot \mathsf{H}_1 \mathit{root} \rangle \\ = & \quad \{\text{definitions, and } \mathsf{H}(A, X) = 1 + (X \times A)\} \\ & [\mathit{one}, \mathit{snoc}] \cdot \sigma \cdot \langle \mathit{id} + (\mathit{id} \times !), [\mathit{nil}, \mathit{snoc}] \cdot (\mathit{id} + \mathit{root} \times \mathit{id}) \rangle \\ = & \quad \{\text{coproduct}\} \\ & [\mathit{one}, \mathit{snoc}] \cdot \sigma \cdot \langle \mathit{id} + (\mathit{id} \times !), [\mathit{nil}, \mathit{snoc} \cdot (\mathit{root} \times \mathit{id})] \rangle \\ = & \quad \{\text{property of } \sigma\} \\ & [\mathit{one}, \mathit{snoc}] \cdot (\mathit{nil} + \langle \mathit{id}, \mathit{snoc} \cdot (\mathit{root} \times \mathit{id}) \rangle) \\ = & \quad \{\text{coproduct}\} \\ & [\mathit{one} \cdot \mathit{nil}, \mathit{snoc} \cdot \langle \mathit{id}, \mathit{snoc} \cdot (\mathit{root} \times \mathit{id}) \rangle]. \end{array}$$

Recalling that the root of a snoc list is its last element, and writing the first and second terms above as $[[\,]]$ and $\otimes$, respectively, we can translate the definition of *subtrees* into more familiar programming terms:

$$\begin{aligned} subtrees &= foldl\,(\otimes)\,[[\,]] \\ x \otimes a &= x +\!\!+ [last\,x +\!\!+ [a]]. \end{aligned}$$

This is precisely the definition of the function *inits* found in the theory of lists.

2. *Binary trees.* Here the labelled tree type (α', T') is isomorphic to the tree type $([tip, node], \mathsf{netree})$ of nonempty binary trees. As before one may calculate that

$$\begin{aligned} &\alpha' \cdot \langle \mathsf{H}_0!, \alpha \cdot \mathsf{H}_1 root\rangle = \\ &[tip \cdot nil, node \cdot \langle id \times id, bin \cdot (id \times (root \times root))\rangle]. \end{aligned}$$

A functional programmer would define *subtrees* by the following equations, writing $bin\,a\,u\,v$ instead of $bin\,(a,(u,v))$ and $node\,a\,u\,v$ instead of $node\,((u,v),a)$:

$$\begin{aligned} subtrees\,nil &= tip\,nil \\ subtrees\,(bin\,a\,u\,v) &= node\,(bin\,a\,(root\,x)\,(root\,y))\,x\,y \\ &\quad \text{where } x = subtrees\,u \\ &\qquad\quad\;\; y = subtrees\,v \\ root\,(node\,a\,u\,v) &= a \end{aligned}$$

Properties Let us now look at some algebraic properties of *subtrees*. To repeat the earlier definition, we are given a binary relator H with tree type (α, T) and labelled tree type (α', T'). Moreover, (α', T') is the tree type of H', defined by $\mathsf{H}'(A,B) = \mathsf{H}(1,B) \times A$.

To start with, we show that *subtrees* is a function. Recalling the definition (4) of *subtrees*, we need only show that $root : \mathsf{id} \leftarrow \mathsf{T}'$ is a function. Observe that

$$root = \in_0 \cdot \alpha'^{\circ} = outr \cdot \alpha'^{\circ}.$$

Since α' is an isomorphism, this particular instance of *root* is a function, and therefore *subtrees* is a function as well.

Next, we have

$$root \cdot subtrees = id \quad \text{and} \quad branch \cdot subtrees = subtrees \cdot branch. \tag{5}$$

The first equation says that the label at the root of $subtrees\,x$ is x itself, and the second equation says that the branches of $subtrees\,x$ are the subtrees of the branches of x. The first equation may be proved using promotion, and the second equation follows from the naturality of membership, but we omit details. Using these two equations, we will now show that *subtrees* is an implementation of $\Lambda subtree$. Recall that $setify : \mathsf{P} \leftarrow \mathsf{T}'$ is defined by $setify = \Lambda \in_{\mathsf{T}'}$.

Lemma 5 $setify \cdot subtrees = \Lambda subtree.$

Proof. First note that because *subtrees* is a function, we have

$$setify \cdot subtrees = \Lambda \in_{\mathsf{T}'} \cdot subtrees = \Lambda(\in_{\mathsf{T}'} \cdot subtrees).$$

Using this fact, the proof proceeds as follows:

$$\begin{array}{ll} & setify \cdot subtrees = \Lambda subtree \\ \equiv & \quad \{\text{above, characterisation of } \Lambda\} \\ & \in_{\mathsf{T}'} \cdot subtrees = subtree \\ \equiv & \quad \{\text{Lemma 3, definition of } subtree\} \\ & root \cdot branch^* \cdot subtrees = branch^* \\ \equiv & \quad \{\text{Implication (2), Equation (5)}\} \\ & root \cdot subtrees \cdot branch^* = branch^* \\ \equiv & \quad \{\text{Equation (5)}\} \\ & true \end{array}$$

□

Accumulations Now let us return to accumulations and the accumulation lemma. In fact, definition (4) of *subtrees* is a special case of an accumulation. Formally, for a given H_1–algebra R the accumulation $\lfloor R \rfloor$ of R is defined by

$$\lfloor R \rfloor \quad = \quad (\!|\alpha' \cdot \langle \mathsf{H}_0!, R \cdot \mathsf{H}_1 root \rangle|\!).$$

Definition (4) is the special case $subtrees = \lfloor \alpha \rfloor$. As we said before, accumulations are very popular in functional programming, where they are known as *scans*. Gibbons [11] has made a study of scans on a particular species of binary tree.

Intuitively, the accumulation $\lfloor R \rfloor$ implements the evaluation of the catamorphism $(\!|R|\!)$ on all subtrees of its argument. This is the content of

Lemma 6 *(Accumulation) For any H_1–algebra R, we have*

$$\mathsf{T}'(\!|R|\!) \cdot subtrees \quad \supseteq \quad \lfloor R \rfloor.$$

The proof is a straightforward application of promotion.

Deforestation The structure built up by an accumulation is usually not the final result of a computation; it is only an intermediate stage. The labelled tree that was constructed with an accumulation is often evaluated by a catamorphism. In such cases, the labelled tree need never be built up as a whole: 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 [21], and Swierstra [20] speaks of *virtual data structures*. The next lemma shows how deforestation can be used in the present context:

Lemma 7 *(Deforestation) Let S be a H'_1–algebra, and R a H_1–algebra. Then*

$$(\!|S|\!) \cdot \lfloor R \rfloor \quad \supseteq \quad outl \cdot (\!|X|\!)$$

where $X = \langle S, outr \rangle \cdot \langle \mathsf{H}(!, outl), R \cdot \mathsf{H}_1\, outr) \rangle$.

Again the proof is a simple application of promotion, and we shall 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 $(\!|S|\!) \cdot \lfloor R \rfloor$ never exists in its entirety anyway. It was for this reason that the above result was not stated in the theory of lists.

7 Pruning

We now proceed to formalise the notion of *pruning* introduced in Section 2. In order to do so, it is necessary to assume that H is of a particular form, namely

$$\mathsf{H}(A, X) \quad = \quad 1 + \mathsf{G}(A, X)$$

for some binary relator G which is not further specified. We follow Backhouse [1] in calling such relators *pointed*. Since $\mathsf{H}(A, X)$ is a coproduct, the tree type (α, T) can be written in the form $[\nu, \rho]$: there exist $\nu : \mathsf{T} \leftarrow 1$ and $\rho : \mathsf{T} \leftarrow \mathsf{G}_1\mathsf{T}$ such that

$$[\nu, \rho] = \alpha.$$

One could think of ν as a constant, representing the empty structure. In the sequel, we shall abuse notation and write ν instead of $\nu \cdot !$.

To prune a tree x means to substitute ν for some (zero or more) subtrees of x. The relation $prune : \mathsf{T} \hookleftarrow \mathsf{T}$ takes a tree and prunes it in some arbitrary way:

$$prune \quad = \quad (\!|\alpha \cup \nu|\!).$$

Note that for snoc lists, *prune* is precisely the suffix relation. The function *prunings* in Section 2 could be defined by $prunings = \Lambda prune$. These examples suggest that *prune* is a preorder; that this is indeed so can be verified using promotion.

Horner's rule Now we turn to the second result in the theory of lists. This result is known as Horner's rule because of its similarities with Horner's method of evaluating polynomials. In [5] the rule was given in the form

$$foldl1(\oplus) \cdot map\,(foldl\,(\otimes)\,e) \cdot tails \quad = \quad foldl\,(\odot)\,e,$$

where $a \odot b = (a \otimes b) \oplus e$. For this identity to be valid, it is required that $\otimes$ distribute backward over $\oplus$. Furthermore, *tails* returns the list of tail segments of a list in descending order of length:

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

For example, taking $\oplus = +$ and $\otimes = \times$ Horner's rule says that

$$(e \times a_1 \times a_2 \times a_3) + (e \times a_2 \times a_3) + (e \times a_3) + e$$

can be evaluated in the form

$$(((e \times a_1 + e) \times a_2) + e) \times a_1 + e.$$

Moreover, the alternative form does not exploit the associativity of $+$ or $\times$.

In many cases in practice, the operation $\oplus$ is either binary minimum or maximum, so $foldl1(\oplus)$ takes the minimum or maximum of essentially a set of values. This leads to a version of Horner' rule in which *tails* is not required to return a list:

$$max \cdot map\,(foldl\,(\otimes)\,e) \cdot tails \quad = \quad foldl\,(\odot)\,e,$$

where $a \odot b = (a \otimes b)\,\mathbf{max}\,e$. The necessary condition here is that $\otimes$ should be monotonic with respect to $\leq$. It is this version we shall generalise to arbitrary tree types. The reason is that, in general, *prunings* cannot be turned into a function returning a structure of structures in the same way as *subtrees*; there is an arbitrary number of prunings we could attach to each node, so we would have to label each node with a set of prunings.

Before we state the generalised version of Horner's rule, we need to make precise what we mean by monotonicity.

Monotonicity A functional algebra $f : A \leftarrow \mathsf{F}A$ is said to be *monotonic* on a relation $R : A \leftarrow A$ if

$$f \cdot \mathsf{F}R \quad \subseteq R \cdot f.$$

Here are two examples to explain this definition.

1. *Addition* Let $R = leq$, where *leq* denotes the relation $\leq$ on number s, and let $f = plus$, where *plus* denotes binary addition. With $\mathsf{F}X = X \times X$ the monotonicity condition translates to

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

   and says that $x = y + z$ and $y \leq y'$ and $z \leq z'$ implies $x \leq y' + z'$. This is just the (true) statement that addition is monotonic in both arguments.

2. *Cons lists.* Let $R = lex$, where *lex* is the lexicographic ordering on lists, and $f = cons$. With $\mathsf{F}X = id \times X$, the monotonicity condition translates to

   $$cons \cdot (id \times lex) \quad \subseteq \quad lex \cdot cons$$

   and says that $x = [a] +\!\!+ y$ and $y \leq y'$ implies $x \leq [a] +\!\!+ y'$. This is just the true statment that *cons* is monotonic with respect to the lexicographic ordering.

Lemma 8 *(Horner's rule) Let* H *be a pointed binary relator and* $g = [g_0, g_1]$ *be a functional* H_1*–algebra. If* g_1 *is monotonic with respect to the preorder* R*, then*

$$maxR \cdot \mathsf{P}(\![g]\!) \cdot \Lambda prune \quad \supseteq \quad (\![maxR \cdot \Lambda\, Y]\!),$$

where $Y = g_0 \cup g$.

The proof of Horner's Rule is a straightforward application of promotion, plus the universal property of $maxR$. Below we consider two examples:

1. *Snoc lists.* The maximum suffix sum can be specified as

   $$mss \;=\; max\,leq \cdot \mathsf{P}sum \cdot \Lambda prune,$$

   where $sum = (\!|zero, plus|\!)$. It is easy to check that $plus$ is monotonic on leq, so Horner's rule is applicable. Since

   $$max\,leq \cdot \Lambda(zero \cup plus) \;\supseteq\; [0, f]$$

   where $f(s, a) = (s + a)\ \mathbf{max}\ 0$, we can implement mss by $(\!|0, f|\!)$.

   Similarly, the maximum sum suffix problem can be specified

   $$mss \;=\; max\,R \cdot \Lambda prune,$$

   where $xRy = sum\,x \leq sum\,y$. Since here $g = \alpha = [nil, snoc]$ the requirement is that $snoc$ should be monotonic on R:

   $$sum\,x \leq sum\,y \;\Rightarrow\; sum\,(x +\!\!+ [a]) \leq sum\,(y +\!\!+ [a]).$$

   This condition is satisfied because addition is monotonic under $\leq$. We can implement this version of mss by $(\!|nil, f|\!)$, where

   $$f(x, a) \;=\; \begin{cases} x +\!\!+ [a], & \text{if } sum\,(x +\!\!+ [a]) > 0 \\ [\,], & \text{otherwise} \end{cases}$$

   We can make this program more efficient by representing each tree x by the pair $(x, sum\,x)$, thereby avoiding computing sum from scratch each time.

2. *Binary trees.* Taking $\mathsf{H}(A, X) = 1 + (A \times (X \times X))$ gives the tree type $([nil, bin], \mathsf{tree})$. The sum of a tree is defined by

   $$sum \;=\; (\!|zero, plus|\!),$$

   where $plus(a, (m, n)) = a + m + n$. For this type mss gives the maximum pruning sum. The monotonicity condition reads

   $$m_0 \leq m_1 \;\wedge\; n_0 \leq n_1 \;\Rightarrow\; plus(a, (m_0, n_0)) \leq plus(a, (m_1, n_1))$$

   and is easily seen to be true. This time we can implement mss by $(\!|zero, f|\!)$, where $f(a, (m, n)) = (a + m + n)\ \mathbf{max}\ 0$.

8 Segments and segment decomposition

Having covered subtrees and prunings, we can now return to arbitrary segments. A tree y is a segment of x if there exists a subtree z of x such that y is a pruning of z. Formally, the natural transformation $segment : \mathsf{T} \hookleftarrow \mathsf{T}$ is defined by

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

This definition was introduced informally in Section 2 for lists and trees of various kinds. The *segment decomposition* theorem stated below puts the previous results together.

Theorem 1 *(Segment Decomposition) Let* H *be a pointed binary relator with tree type* (α, T). *Suppose* $g = [g_0, g_1]$ *is a functional* H_1*-algebra, and that* g_1 *is monotonic on the preorder* R. *Then*

$$maxR \cdot \mathsf{P}(\!|g|\!) \cdot \Lambda segment \quad \supseteq \quad (\!|maxR \cdot \Lambda X|\!) \cdot \lfloor maxR \cdot \Lambda Y \rfloor,$$

where $X = \in_0 \cup \in_1$, *and* $Y = g_0 \cup g$.

Proof. As before, let H' be the labelled variant of H, with tree type (α', T'). We argue

$$\begin{array}{ll}
 & maxR \cdot \mathsf{P}(\!|g|\!) \cdot \Lambda segment \\
= & \quad \{\text{definition of } segment\} \\
 & maxR \cdot \mathsf{P}(\!|g|\!) \cdot \Lambda(prune \cdot subtree) \\
= & \quad \{\Lambda \text{ of composition}\} \\
 & maxR \cdot \mathsf{P}(\!|g|\!) \cdot \mathsf{E}prune \cdot \Lambda subtree \\
= & \quad \{\mathsf{E} \text{ in terms of } \mathsf{P} \text{ (Eq. 1)}\} \\
 & maxR \cdot \mathsf{P}(\!|g|\!) \cdot union \cdot \mathsf{P}\Lambda prune \cdot \Lambda subtree \\
= & \quad \{\text{since } \mathsf{P} = \mathsf{E} \text{ on functions and } union : \mathsf{E} \leftarrow \mathsf{E} \cdot \mathsf{E}\} \\
 & maxR \cdot union \cdot \mathsf{P}(\mathsf{P}(\!|g|\!) \cdot \Lambda prune) \cdot \Lambda subtree \\
\supseteq & \quad \{\text{distributing } maxR \text{ over } union, \text{ since } R \text{ is a preorder}\} \\
 & maxR \cdot \mathsf{P}(maxR \cdot \mathsf{P}(\!|g|\!) \cdot \Lambda prune) \cdot \Lambda subtree \\
\supseteq & \quad \{\text{Horner's rule, Lemma 8}\} \\
 & maxR \cdot \mathsf{P}(\!|maxR \cdot \Lambda Y|\!) \cdot \Lambda subtree \\
= & \quad \{\text{Lemma 5}\} \\
 & maxR \cdot \mathsf{P}(\!|maxR \cdot \Lambda Y|\!) \cdot setify \cdot subtrees \\
\supseteq & \quad \{\text{since } setify : \mathsf{P} \hookleftarrow \mathsf{T}'\} \\
 & maxR \cdot setify \cdot \mathsf{T}'(\!|maxR \cdot \Lambda Y|\!) \cdot subtrees \\
\supseteq & \quad \{\text{Accumulation Lemma 6}\} \\
 & maxR \cdot setify \cdot \lfloor maxR \cdot \Lambda Y \rfloor \\
\supseteq & \quad \{\text{Lemma 4}\} \\
 & (\!|maxR \cdot \Lambda X|\!) \cdot \lfloor maxR \cdot \Lambda Y \rfloor.
\end{array}$$

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

9 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 [1], who has shown how many theorems about regular algebra can be generalised to data types. Backhouse and his team have also developed a generic theory of *zips* [2].

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 [15] 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. Johan Jeuring and Jaap van der Woude scrutinized drafts of this paper, and suggested many improvements.

References

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

[2] Roland Backhouse, Henk Doornbos, and Paul Hoogendijk. Commuting relators. Technical Report. Available by anonymous ftp from `ftp.win.tue.nl`, directory `pub/math.prog.construction`., 1992.

[3] M. Barr and C. Wells. *Toposes, Triples and Theories*, volume 278 of *Grundlehren der Mathematischen Wissenschaften*. Springer–Verlag, 1985.

[4] 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.

[5] R. S. Bird. Lectures on constructive functional programming. In M. Broy, editor, *Constructive Methods in Computing Science*, volume 55 of *NATO ASI Series F*, pages 151–216. Springer–Verlag, 1989.

[6] R. S. Bird. A calculus of functions for program derivation. In D.A. Turner, editor, *Research Topics in Functional Programming*, University of Texas at Austin Year of Programming Series, pages 287–308. Addison–Wesley, 1990.

[7] A. Carboni, G.M. Kelly, and R.J. Wood. A 2–categorical approach to geometric morphisms I. *Cahiers de Topologie et Geometrie Differentielle Categoriques*, 32(1):47–95, 1991.

[8] O. de Moor. Categories, relations and dynamic programming. D.Phil. thesis. Technical Monograph PRG-98, Computing Laboratory, Oxford, 1992.

[9] O. de Moor. Working notes on membership of data types. Unpublished manuscript, 1993.

[10] P. J. Freyd and A. Ščedrov. *Categories, Allegories*, volume 39 of *Mathematical Library*. North–Holland, 1990.

[11] J. Gibbons. Algebras for tree algorithms. D.Phil. thesis. Programming Research Group, Computing Laboratory, 11 Keble Road, Oxford OX1 3QD, September 1991.

[12] R. Goldblatt. *Topoi — The Categorial Analysis of Logic*, volume 98 of *Studies in Logic and the Foundations of Mathematics*. North–Holland, 1986.

[13] J. Jeuring. Deriving algorithms on binary labelled trees. In P.M.G. Apers, D. Bosman, and J. Van Leeuwen, editors, *Proceedings SION Computing Science in the Netherlands*, pages 229–249, 1989.

[14] P. T. Johnstone. *Topos Theory*. Academic Press, 1977.

[15] M. P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. *Journal of Functional Programming*, 5(1):1–35, 1995.

[16] J. Lambek. A fixpoint theorem for complete categories. *Mathematische Zeitschrift*, 103:151–161, 1968.

[17] D. J. Lehmann and M. B. Smyth. Algebraic specification of data types: A synthetic approach. *Mathematical Systems Theory*, 14:97–139, 1981.

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

[19] E. G. Manes and M. A. Arbib. *Algebraic Approaches to Program Semantics*. Texts and Monographs in Computer Science. Springer–Verlag, 1986.

[20] S. D. Swierstra and O. de Moor. Virtual data structures. In B. Möller, H. Partsch, and S. Schuman, editors, *Formal Program Development*, volume 755 of *Lecture Notes in Computer Science*, pages 355–371, 1993.

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