To appear in *Algebraic Methodology and Software Technology*, May 2000.

# Pointwise Relational Programming

Oege de Moor and Jeremy Gibbons

Programming Research Group, Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD, United Kingdom

**Abstract.** The point-free relational calculus has been very successful as a language for discussing general programming principles. However, when it comes to specific applications, the calculus can be rather awkward to use: some things are more clearly and simply expressed using variables. The combination of variables and relational combinators such as converse and choice yields a kind of nondeterministic functional programming language. We give a semantics for such a language, and illustrate with an example application.

## 1 Introduction

The following question appeared on a first year exam in algorithm design:

> The data type of trees is given by
>
> $$\text{data } \mathit{Tree}\ a \quad = \quad \mathit{Tip}\ a \mid \mathit{Bin}\,(\mathit{Tree}\ a)\ a\ (\mathit{Tree}\ a)\ .$$
>
> That is, a tree is a tip labelled with an element, or it consists of node that contains two subtrees and a label. A tree is said to be a *search tree* if its inorder traversal is strictly increasing. Design an algorithm that, given the preorder traversal of a search tree $t$, returns the tree $t$ itself.

The original statement of the question also asked candidates for efficient algorithms to perform inorder and preorder traversals. At the time the exam was set, this seemed a reasonable question. The solution had not been covered in the lectures, but students had seen similar algorithms for tree building. In the exam itself, however, no one got the answer right, so apparently this kind of question is too hard. That is discouraging, especially in view of the highly sophisticated problems that the same students can solve in mathematics exams. Why is programming so much harder?

In computer science, students are generally not provided with a coherent body of theorems that guide them to problem solutions. This contrasts with mathematics, where students do get a number of strategic theorems and subsidiary results that provide beacons in the quest for solutions. The program derivation community has for some time been aware of this difficulty, and worked on finding a body of theorems that aids the design of algorithms, and that encapsulates common problem solving techniques in precise mathematical facts [17, 19, 29]. In particular, the point free relational calculus has proved extraordinarily successful in the derivation of general programming principles [1, 4, 9, 10, 13, 14, 21, 20]. The calculus is said to be *point free* because it does not name the data values: instead, one manipulates programs and specifications through a set of combining forms, much in the style originally suggested by Backus [5] for functional programming. As the above references demonstrate, this style is appropriate for proving generic results. When applied to specific examples, however, it can be somewhat painful to use. Properties that are tacitly applied in a pointwise style become opaque when expressed using combinators. For example, when applying associativity of list concatenation

$$(x +\!\!+\, y) +\!\!+\, z \;=\; x +\!\!+\, (y +\!\!+\, z)$$

in a pointwise style, one simply leaves out the brackets. One cannot do the same in the point free statement (writing *cat* for $(+\!\!+)$ and *assoc* for the re-associating isomorphism)

$$cat \cdot (cat \times id) \;=\; cat \cdot (id \times cat) \cdot assoc \;.$$

As argued by Backhouse [3], silent application of ubiquitous properties such as associativity allows one to concentrate on the important mathematical facts, without being distracted by excess formality. It is our belief that the relational calculus (and the programming principles expressed in it) is unusable for program derivation in its present form, because it does distract from the essence of particular problems.

In this paper, we explore the suggestion that it may be preferable to conduct relational derivations of concrete programs in a pointwise calculus of nondeterministic mappings. The presentation is largely informal: we introduce concepts and notations through an example that is similar to the above exam question, but a little harder. Ideally, this slightly more complicated example would have been covered in lectures. By learning the general results in that derivation, students would find the above exam question easy, and we invite the reader to test our hypothesis by solving the exam question after reading this paper.

The semantics of nondeterministic mappings is a nettlesome topic, and it behoves us to outline the semantics we have in mind. It will come as no surprise that the basis is a translation from nondeterministic expressions to relational calculus, and the change from traditional relational calculus is therefore mostly notational.

## 2 A programming problem

Consider the problem of constructing a heap from a given sequence $x$, so that the inorder traversal of the heap is $x$ itself. We shall derive a linear time algorithm that was first suggested by Robert Floyd in the context of precedence parsing [18].

**Trees.** A *tree* is either empty, or it consists of two subtrees and an element. In the notation of Haskell [12], we can write

$$\text{data } Tree\ a \;=\; Null \;\mid\; Bin\ (Tree\ a)\ a\ (Tree\ a)\ .$$

**Inorder traversal.** The *inorder traversal* of a tree is returned by the function $\mathit{flatten} :: \mathit{Tree}\ a \to [a]$:

$$\begin{array}{lcl} \mathit{flatten}\ \mathit{Null} & = & [\,] \\ \mathit{flatten}\ (\mathit{Bin}\ s\ a\ t) & = & \mathit{flatten}\ s \mathbin{+\!\!+} [a] \mathbin{+\!\!+} \mathit{flatten}\ t\ . \end{array}$$

**Heaps.** A tree is said to be a *heap* if the label of each node is at most the elements below it. To wit, define the predicate $\mathit{heap} :: \mathit{Tree}\ a \to \mathit{Bool}$ by

$$\begin{array}{lcl} \mathit{heap}\ \mathit{Null} & = & \mathit{True} \\ \mathit{heap}\ (\mathit{Bin}\ s\ a\ t) & = & \mathit{heap}\ s \ \ \wedge\ \ \mathit{heap}\ t \ \ \wedge\ \ \mathit{above}\ a\ s \ \ \wedge\ \ \mathit{above}\ a\ t\ . \end{array}$$

The predicate $\mathit{above} :: a \to \mathit{Tree}\ a \to \mathit{Bool}$ checks that $a$ is at most the label of its descendant:

$$\begin{array}{lcl} \mathit{above}\ a\ \mathit{Null} & = & \mathit{True} \\ \mathit{above}\ a\ (\mathit{Bin}\ s\ b\ t) & = & a \leq b\ . \end{array}$$

Note that we do *not* require heaps to be balanced.

**Specification.** Our aim is to construct a total function *build* that is a right-inverse of *flatten*, and whose result is a heap. Formally, $\mathit{build} :: [a] \to \mathit{Tree}\ a$ should satisfy

$$\mathit{flatten}(\mathit{build}\ x) = x \quad \text{and} \quad \mathit{heap}\ (\mathit{build}\ x)\ .$$

We find it convenient to write the definition of *build* as a composition of *nondeterministic mappings*. For the purpose of this example, a nondeterministic mapping is merely another way of denoting a binary relation. The mapping $\mathit{flatten}^\circ :: [a] \to \mathit{Tree}\ a$ takes a list $x$, and returns some tree whose inorder traversal is $x$. Note that $\mathit{flatten}^\circ$ is nondeterministic, because it has the choice between many different trees. The mapping $\mathit{heap}? :: \mathit{Tree}\ a \to \mathit{Tree}\ a$ is a subset of the identity relation, only mapping a tree $t$ to itself if $t$ is a heap. Thus, $\mathit{heap}?$ is partial. Our problem is to construct a refinement *build* of the composition $\mathit{heap}? \cdot \mathit{flatten}^\circ$:

$$\mathit{build} \ \sqsupseteq\ \mathit{heap}? \cdot \mathit{flatten}^\circ\ .$$

This specification does not state the additional requirement that *build* is a total function.

Our specification makes use of two operators that are generally useful in the sequel. The *converse* of a mapping $f :: A \to B$ simply swaps the pairs in the binary relation that $f$ denotes, and we write $f^\circ :: B \to A$. Given a total function $p :: A \to \mathit{Bool}$, the *test* of $p$ is a subset $p?$ of the identity relation consisting of precisely those elements that satisfy $p$.

### 2.1 Representing Trees by Spines

One could try and derive a solution directly from the above specification, but this will lead to a quadratic time algorithm. We therefore seek a different representation of trees. The choice of such a representation is the main inventive step in our derivation.

Both in algorithm design and in the study of programming language interpreters, it is very common to represent a tree by its *spine*. The type of spines is given by the synonym

$$\text{type } \mathit{Spine}\ a \ = \ [(a, \mathit{Tree}\ a)] \ .$$

A spine is turned into a tree by rolling it up from the left. That is, the function $\mathit{roll} :: \mathit{Spine}\ a \to \mathit{Tree}\ a$ is defined by

$$\begin{aligned} \mathit{roll} \ &= \ \mathit{foldl}\ \mathit{bin}\ \mathit{Null} \\ &\quad \text{where } \mathit{bin}\ s\ (a, t) \ = \ \mathit{Bin}\ s\ a\ t \ . \end{aligned}$$

The function *roll* is in fact a bijection. Here $\mathit{foldl} :: (a \to b \to a) \to a \to [b] \to a$ is the function that takes a binary operator, a seed value and a list, and sums the list from left to right, starting with the given seed value.

Before proceeding, it is useful to introduce a dual of *foldl*, named *foldr*. Its type is $\mathit{foldr} :: (a \to b \to b) \to b \to [a] \to b$, and *foldr step seed* sums a list from right to left, using the operator *step*, and starting with the given value *seed*.

Having made the decision to represent trees by spines, we can sketch a possible derivation, making two wishful steps that will need further elaboration in the sequel:

$$\begin{array}{ll} & \mathit{heap}? \cdot \mathit{flatten}^{\circ} \\ = & \quad \{\text{since } \mathit{roll} \text{ is bijective}\} \\ & \mathit{heap}? \cdot \mathit{roll} \cdot \mathit{roll}^{\circ} \cdot \mathit{flatten}^{\circ} \\ = & \quad \{\text{since converse is contravariant}\} \\ & \mathit{heap}? \cdot \mathit{roll} \cdot (\mathit{flatten} \cdot \mathit{roll})^{\circ} \\ = & \quad \{\text{WISH: inverting } \mathit{flatten} \cdot \mathit{roll}\} \\ & \mathit{heap}? \cdot \mathit{roll} \cdot \mathit{foldr}\ \mathit{add}\ [\,] \\ = & \quad \{\text{define } \mathit{hp} = \mathit{heap} \cdot \mathit{roll}\} \\ & \mathit{roll} \cdot \mathit{hp}? \cdot \mathit{foldr}\ \mathit{add}\ [\,] \\ \sqsubseteq & \quad \{\text{WISH: fusion}\} \\ & \mathit{roll} \cdot \mathit{foldr}\ \mathit{join}\ [\,] \end{array}$$

Note that all steps are equalities except for the last one, which is a refinement. We only reduce the amount of nondeterminism by choosing a specific tree that has the heap property. If we can fulfill the wishes in the above calculation, and *join* is a total function, we can set

$$\mathit{build} \ = \ \mathit{roll} \cdot \mathit{foldr}\ \mathit{join}\ [\,] \ ,$$

and our problem is solved.

Let us now consider the two unproven steps in some more detail. In the first wishful step, we assumed the existence of a mapping *add* such that

$$(\mathit{flatten} \cdot \mathit{roll})^{\circ} \;=\; \mathit{foldr}\;\mathit{add}\;[\,]\;.$$

The second wishful step aims to promote the test for heapiness of a spine *hp*?, so that

$$\mathit{hp}? \cdot \mathit{foldr}\;\mathit{add}\;[\,] \;\sqsubseteq\; \mathit{foldr}\;\mathit{join}\;[\,]\;.$$

To discharge this requirement, we shall have to synthesise a total function *join* that satisfies the above containment.

### 2.2 First wishful step: Inverting $\mathit{flatten} \cdot \mathit{roll}$

Abbreviate $\mathit{flar} = \mathit{flatten} \cdot \mathit{roll}$. Before inverting *flar*, it is worthwhile to examine it a bit further. To start with, note the type:

$$\mathit{flar} \;::\; \mathit{Spine}\;a \to [a]\;.$$

To flatten a spine, we could first flatten its components, and then concatenate the result. That is,

$$\mathit{flar} \;=\; \mathit{concat} \cdot \mathit{map}\;\mathit{flat}\;,$$

where $\mathit{flat} :: (a, \mathit{Tree}\;a) \to [a]$ flattens a single spine component:

$$\mathit{flat}\;(a, t) \;=\; a : \mathit{flatten}\;t\;.$$

It now follows that *flar* distributes over $[\,]$ and $(+\!\!+)$ as follows:

$$\begin{array}{lcl}\mathit{flar}\;[\,] & = & [\,]\\ \mathit{flar}\;(x +\!\!+ y) & = & \mathit{flar}\;x +\!\!+ \mathit{flar}\;y\;.\end{array}$$

This concludes our little exploration of *flar* itself; we now turn to its inversion.

**Converse Function Theorem [13]** *Let $f :: A \to [B]$ be a total function. Let $e :: A$ and $(\oplus) :: B \to A \to A$ be mappings that are jointly surjective onto $A$. Furthermore assume that*

$$f\;e \sqsupseteq [\,] \;\; \textit{and} \;\; f\;(a \oplus x) \sqsupseteq a : f\;x\;.$$

*Then we have $f^{\circ} = \mathit{foldr}\;(\oplus)\;e$.*

To say that two mappings are *jointly surjective* onto $A$ is to say that the union of their ranges is $A$. To apply the above result to $f = \mathit{flar}$, we start with the right hand side of the last condition, that is $a : \mathit{flar}\;x$. Without assuming anything about the structure of $x$, it is difficult to make progress, so let us say that

$x = y \mathbin{+\!\!+} z$, for some $y$ and $z$. This is no restriction, because concatenation is surjective. We now calculate:

$$\begin{array}{cl}
 & a : \mathit{flar}\,(y \mathbin{+\!\!+} z) \\
= & \quad \{\mathit{flar} \text{ distributes over } (\mathbin{+\!\!+})\} \\
 & a : \mathit{flar}\, y \mathbin{+\!\!+} \mathit{flar}\, z \\
= & \quad \{\text{definition of } \mathit{flar}\} \\
 & a : \mathit{flatten}(\mathit{roll}\; y) \mathbin{+\!\!+} \mathit{flar}\, z \\
= & \quad \{\text{definition of } \mathit{flat} \text{ (in reverse)}\} \\
 & \mathit{flat}(a, \mathit{roll}\; y) \mathbin{+\!\!+} \mathit{flar}\, z \\
= & \quad \{\text{since } \mathit{flar} = \mathit{concat} \cdot \mathit{map}\, \mathit{flat}\} \\
 & \mathit{flar}((a, \mathit{roll}\; y) : z)
\end{array}$$

It follows that for $a \oplus (y \mathbin{+\!\!+} z) = (a, \mathit{roll}\; y) : z$, the condition $\mathit{flar}\,(a \oplus x) \sqsupseteq a : \mathit{flar}\; x$ is satisfied. The other conditions of the converse function theorem are trivially satisfied, and so we conclude that

$$\begin{array}{lcl}
\mathit{flar}^{\circ} & = & \mathit{foldr}\; \mathit{add}\; [\,] \\
\mathit{add}\; a\,(y \mathbin{+\!\!+} z) & = & (a, \mathit{roll}\; y) : z\ .
\end{array}$$

Some readers will raise their eyebrows at our loose use of non-injective patterns in definitions, such as $(y \mathbin{+\!\!+} z)$. Indeed, we could have written the following instead, using more conventional notation:

$$\begin{array}{lcl}
\mathit{add}\; a\; \mathit{yz} & = & (a, t) : z \\
 & & \quad \text{where } (t, z) = \mathit{roll}1\,(\mathit{split}\; \mathit{yz})\ .
\end{array}$$

Here $\mathit{roll}1(y, z) = (\mathit{roll}\; y, z)$, and $\mathit{split}$ is the converse of concatenation, that is $(\mathit{uncurry}(\mathbin{+\!\!+}))^{\circ}$.

The remainder of the derivation is completely routine for those who are familiar with an algebraic style of programming, and we have only spelt out the details for the benefit of those who wish to learn about this style. Readers may wish to skip to the next section at this point, and only return to the full details of the derivation later.

### 2.3 Second wishful step: Promoting *hp*?

Our next task is to promote the test for heapiness into the above expression for $\mathit{flar}^{\circ}$. Recall the exact statement of that proof obligation:

$$\mathit{hp}? \cdot \mathit{foldr}\; \mathit{add}\; [\,] \ \sqsubseteq\ \mathit{foldr}\; \mathit{join}\; [\,]\ ,$$

where $\mathit{join}$ is a total function. Instead of directly deriving a total function, we shall first find a total mapping $\mathit{step}$ such that

$$\mathit{hp}? \cdot \mathit{foldr}\; \mathit{add}\; [\,] \ \sqsubseteq\ \mathit{foldr}\; \mathit{step}\; [\,]\,.$$

Then, in a later subsection, we shall refine $\mathit{step}$ to a total function $\mathit{join}$.

**Synthesis of *step*.** Our starting point is the well-known

**Fusion Theorem** [1] *Let* $f : B \to A$, $(\oplus) :: C \to B \to B$, $e :: B$, $(\otimes) :: C \to A \to A$, *and* $d :: A$ *be mappings. Then*

$$f \cdot foldr\,(\oplus)\,e \;\sqsubseteq\; foldr\,(\otimes)\,d \ ,$$

*provided* $f(a \oplus x) \sqsubseteq a \otimes f\,x$, *and* $f\,e \sqsubseteq d$.

To prove the above containment, it therefore suffices to find a mapping *step* so that

$$hp?(add\ a\ yz) \;\sqsubseteq\; step\ a\,(hp?\ yz) \ .$$

Clearly, in order to do so, we shall need a number of properties of *hp*. Recall the definition $hp = heap \cdot roll$. It is easy to verify that

$$\begin{array}{lcl} hp\,[\,] & = & True \\ hp\,((a,t):z) & = & heap\ t \ \wedge\ hp\ z \ \wedge\ bet\ a\,(t,z) \ . \end{array}$$

Here $bet\ a\,(t,z)$ checks whether $a$ lies between $t$ and $z$. That is:

$$\begin{array}{lcl} bet\ a\,(t,z) & = & above\ a\ t \ \wedge\ fit\ a\ z \\ & & \text{where } fit\ a\,[\,] = True \\ & & \qquad\ \ fit\ a\,((b,t):u) = b \le a \ . \end{array}$$

Armed with these facts, a straightforward calculation shows that we have the desired containment

$$hp?(add\ a\ yz) \;\sqsubseteq\; step\ a\,(hp?\ yz) \ .$$

for the following definition of *step*:

$$\begin{array}{lcl} step\ a\ yz & = & (a,t):x \\ & & \text{where } (t,x) = (bet\ a)?\,(roll\ y, z) \\ & & \qquad\qquad\qquad \text{where } \ (y +\!\!+\, z) = yz \ . \end{array}$$

Again the use of a non-injective pattern could have been avoided at the expense of a little perspicuity:

$$\begin{array}{lcl} step\ a\ yz & = & (a,t):z \\ & & \text{where } (t,z) = (bet\ a)?\,(roll1\,(split\ yz)) \end{array}$$

with the auxiliary definitions

$$\begin{array}{lcl} roll1\,(y,z) & = & (roll\ y, z) \\ split & = & (\lambda(x,y).x +\!\!+\, y)^{\circ} \ . \end{array}$$

**Refining** *step* **to** *join***.** To complete our derivation, we need to refine the above definition of *step* to a total function *join*. We first concentrate on finding an effective definition of $roll1 \cdot split$. Write $f^*$ for the reflexive transitive closure of a mapping $f$.

> **Leap Frog Lemma for Closure** [**13**] *Let* $f :: A \to B$ *and* $g :: A \to A$ *and* $h : B \to B$ *be mappings. If* $f \cdot g = h \cdot f$*, then*
>
> $$f \cdot g^* \;=\; h^* \cdot f \;.$$

We have

$$split\ yz \;=\; shift^*([\,], yz)\;,$$

where $shift(y,\ [a] +\!\!+\ z) = (y +\!\!+\ [a],\ z)$. Note that

$$roll1 \cdot shift \;=\; shunt \cdot roll1$$

where $shunt(s,\ [(a,t)] +\!\!+\ z) = (Bin\ s\ a\ t,\ z)$. By the above leap frog property, we conclude

$$roll1(split\ yz) \;=\; shunt^*(roll1\,([\,], yz)) \;=\; shunt^*(Null,\ yz)\;.$$

In words, one can generate all possible results of $roll1\,(split\ yz)$ by repeated shunting.

In the definition of *step*, we wished to have a result of $roll1\,(split\ yz)$ that satisfies the predicate $bet\ a$. Arbitrarily, we chose the first such result encountered during the shunting, thus obtaining the function $join \sqsupseteq step$:

$$\begin{array}{lll} join\ a\ yz & = & (a,t) : x \\ & & \text{where}\ \ \begin{array}[t]{lcl} (t,x) & = & search\,(Null, yz) \\ search\,(t,x) & = & \text{if}\ bet\ a\ (t,x) \\ & & \text{then}\ (a,t) : x \\ & & \text{else}\ search\,(shunt\,(t,x))\;. \end{array} \end{array}$$

Now consider whether this definition of *join* is indeed a total function. It is certainly single-valued. To see that it is also total, note that if there are no elements in the spine where $a$ fits in between, then either $a$ is strictly smaller than all elements in the spine, or it is strictly greater. If $a$ is strictly smaller, it can be inserted at the end of the spine, and if it is strictly greater, it can be inserted at the head.

The full program that we derived is included in an appendix. To see that it is linear time, note that after a node has been shunted, it is never considered again. It follows that if we sum the cost of the *join* operation over the whole duration of the algorithm, the total cost is proportional to the number of nodes, and hence linear.

### 2.4 Afterthoughts

Looking back at the derivation, an appeal to the Converse Function Theorem was not really necessary here. In the inversion of *flar*, we could simply have proved an inclusion using the Fusion Theorem in lieu of the Converse Function Theorem. That would however have been unsatisfactory. With hardly any additional proof effort, the Converse Function Theorem shows exactly where the reduction of indeterminacy occurs, namely in the choice of a particular heap, and not in the inversion of *flar*.

Although the derivation *per se* is not the main concern of this paper, the reader may wish to compare our argument with those given in [2, 11, 15, 22, 28], all of whom consider the same (or a similar) problem. We believe that our derivation proceeds in greater strides, all of which can be applied to other inversion problems. In particular, they apply to the exam question quoted at the beginning of this paper. That problem is easier than the one we just covered, because it is not necessary to consider the change of representation from trees to spines.

## 3 Semantics

So far we have been entirely informal about the meaning of our expressions, relying on the reader's intuition about nondeterministic mappings. It is well known that there are difficult semantical problems in dealing with nondeterminism and higher order functions, and we ought to address these at least to the extent that the above derivation can be fully justified. Our starting point is the standard semantics of $\lambda$-expressions in cartesian closed categories. It will be useful if the reader has some knowledge of elementary category theory [6].

For the purpose of this paper, one can think of a category as a universe of typed specifications: it consists of objects (types) and arrows (specifications or programs). Each arrow $h$ has a source type $A$ and a target type $B$: we write $h : A \to B$ to indicate this type information. For each object $A$ there is a distinguished arrow $id : A \to A$. Arrows can be composed, subject to some typing rules. That is, when $f : A \to B$ and $g : B \to C$, their composite is $g \cdot f : A \to C$. Composition is associative and has identity element $id$.

The canonical example of a category is *Fun*, where the objects are sets and the arrows are total functions. Another example is *Rel*, where the objects are also sets, but where the arrows are binary relations, *i.e.* sets of pairs, composed in the usual manner.

### 3.1 Cartesian closed categories

A *binary functor* on a category is a binary operation $\oplus$ on objects and on arrows, which respects typing ($f \oplus g : A \oplus C \to B \oplus D$ when $f : A \to B$ and $g : C \to D$), identities ($id \oplus id = id$) and composition ($(f \cdot g) \oplus (h \cdot j) = (f \oplus h) \cdot (g \oplus j)$).

A category *has finite products* if it has a binary functor $\times$, with arrows $fst : A \times B \to A$ and $snd : A \times B \to B$, where for arrows $f$ and $g$ with a common source there is a unique arrow $h$ such that $fst \cdot h = f$ and $snd \cdot h = g$. This unique $h$ is written $\langle f, g \rangle$. In the category *Fun*, cartesian product of sets with parallel composition of functions is a categorical product.

A *cartesian closed* category is a category with finite products and a binary functor $\Rightarrow$ where there exists an isomorphism between arrows of type $A \times B \to C$ and arrows of type $A \to (B \Rightarrow C)$. In the category *Fun*, $B \Rightarrow C$ is the set of functions from $B$ to $C$. The two sides of the isomorphism are named *curry* and *uncurry* respectively.

It is useful to define *evaluation* by

$$\begin{array}{lcl} eval & : & (A \Rightarrow B) \times A \to B \\ eval & = & uncurry\, id\ . \end{array}$$

Note that we have $uncurry\, f = eval \cdot (f \times id)$.

**Semantics of λ-calculus.** Each cartesian closed category determines a semantics for typed λ-calculi. The way this works is by mapping primitive types to the objects of the category, and then defining mappings from composite types to objects, and from expressions to the arrows of the category. Thorough introductions to the subject may be found in [16, 23].

To be precise, if we have a mapping *obj* from primitive types to objects, we extend the translation to composite types as follows (1 stands for the nullary product):

$$\begin{array}{lcl} funobj\ t & = & obj\ t \\ funobj\ 1 & = & 1 \\ funobj\ (S \times T) & = & funobj\ S \times funobj\ T \\ funobj\ (S \to T) & = & funobj\ S \Rightarrow funobj\ T \end{array}$$

We stress once more that *funobj* maps type expressions to objects of the category *Fun*. Hence, although we use the same symbol $\times$ on both sides of the equation, their meanings are quite different.

To give a meaning to closed λ-expressions, we map expressions of type $T$ to arrows of type $1 \to funobj\ T$. The translation is in terms of a function *fun* that takes the expression to be translated, as well as a list of free variables as its arguments:

$$\begin{array}{lcl} funtrans\, E & = & fun\, E\, [\,] \\ fun\, x\, xs & = & snd \cdot fst^i, \quad \text{where } xs_i \text{ is the first occurrence of } x \text{ in } xs \\ fun\, c\, xs & = & funcon\, c \cdot fst^n, \quad \text{where } n \text{ is the length of } xs \\ fun\, (F\, E)\, xs & = & eval \cdot \langle fun\, F\, xs, fun\, E\, xs \rangle \\ fun\, (\lambda x.E)\, xs & = & curry\, (fun\, E\, (x : xs)) \end{array}$$

Given a list $xs$ of variables $[x_0 : T_0, x_1 : T_1, \ldots, x_{n-1} : T_{n-1}]$, and $E : T$, the type of $fun\, E\, xs$ is

$$(((1 \times funobj\ T_{n-1}) \times funobj\ T_{n-2}) \ldots \times funobj\ T_0) \quad \to \quad funobj\ T$$

Note that in specifying the translation of constant expressions, we have assumed the existence of an operator *funcon* that maps each constant $(c : T)$ to its interpretation as an arrow $1 \to \mathit{funobj}\ T$. When the above translation is applied in the category of sets and total functions, it gives the standard semantics where $\lambda$-expressions are read as total functions.

### 3.2 Relations

Our aim, however, is to read $\lambda$-expressions as arrows in *Rel*, not *Fun*. Furthermore, we would like to extend the above translation to expressions that involve nondeterministic choice, intersection and arbitrary patterns in definitions. An obvious strategy presents itself: find appropriate new definitions of *curry* and *uncurry* (and hence of *eval*), and use these instead of the usual functional definitions.

**A failed attempt to define exponents.** The first problem we encounter in such an attempt is that pairing is not a product in *Rel*. There is an obvious definition taking a pair of relations $R : A \to B$ and $S : A \to C$ to $\langle R, S\rangle : A \to B \times C$:

$$(b, c)\langle R, S\rangle a \;\equiv\; bRa \;\wedge\; cRa \;.$$

But it is not the case that

$$\mathit{fst} \cdot \langle R, S\rangle \;=\; R \;.$$

Take, for example, $R = \mathit{id}$ and $S = \emptyset$.

We could attempt to carry on nevertheless, and look for an appropriate equivalent of $B \Rightarrow C$ in *Rel*. That is, we seek a definition of $(\sim)$ so that relations of type $A \times B \to C$ are in one-to-one correspondence with relations of type $A \to (B \sim C)$. The obvious choice is

$$B \sim C \;=\; B \times C \;.$$

It follows that we have an isomorphism $\mathit{curry}_r$ and $\mathit{uncurry}_r$, and this isomorphism mimics the situation in a cartesian closed category.

We could now apply the above translation of $\lambda$-terms, using $\mathit{curry}_r$ for *curry* and $\mathit{uncurry}_r$ for *uncurry*. But that leads to rather unpleasant surprises. For instance, if we translate the expression

$$(\lambda f.\lambda x.f(f\ x))\ \mathit{succ}$$

(where *succ* is the successor function) the result is the empty relation. We conclude that this is not our intended semantics.

**A successful definition of exponents.** A *topos* is a category where relational composition is well-defined and associative (in the jargon: a *regular* category), and furthermore there exists an isomorphism between relations of type $A \to B$ and functions of type $A \to P\,B$. The category of sets and total functions is a topos, and there $P\,B$ is the set of all subsets of $B$. The transition from relations to set-valued functions is called *power transpose* and written

$$\Lambda R\; a \;=\; \{\, b \mid b\; R\; a \,\}\ .$$

Using this additional structure, we define:

$$\begin{array}{lcl} B \rightsquigarrow C & = & P(B \sim C) \\ \mathit{curry}_p\; R & = & \Lambda(\mathit{curry}_r\; R) \\ \mathit{uncurry}_p\; S & = & \mathit{uncurry}_r(\in \cdot\, S)\ . \end{array}$$

This does not define an isomorphism, for we have $\mathit{uncurry}_p(\mathit{curry}_p\; R) \;=\; R$, but not in general $\mathit{curry}_p(\mathit{uncurry}_p\; S) \;=\; S$. It is however the case that these definitions coincide with the functional ones in a precise sense. Note that we can define a function

$$\begin{array}{lcl} \rho & : & (A \Rightarrow B) \to (A \rightsquigarrow B) \\ \rho & = & \mathit{curry}_p\; \mathit{eval}\ . \end{array}$$

Now for any functions $f$ and $g$ of appropriate type we have:

$$\rho \cdot \mathit{curry}\, f = \mathit{curry}_p f \ \text{ and } \ \mathit{uncurry}_p(\rho \cdot g) \;=\; \mathit{uncurry}\, g\ .$$

Furthermore, we note that the conversion $\rho$ is injective.

In analogy with the definition of *eval*, we define

$$\mathit{eval}_p \;=\; \mathit{uncurry}_p\, \mathit{id}\ .$$

It is easily checked that $\mathit{eval}_p \cdot (\rho \times \mathit{id}) = \mathit{eval}$, so again this definition coincides with the functional one on functional arguments.

### 3.3 Relational Semantics

Our first task is to define a mapping from types to objects of *Rel*, interpreting arrows as our newly defined exponent:

$$\begin{array}{lcl} \mathit{relobj}\; A & = & \mathit{obj}\; A \\ \mathit{relobj}\; 1 & = & 1 \\ \mathit{relobj}\; (S \times T) & = & \mathit{relobj}\; S \times \mathit{relobj}\; T \\ \mathit{relobj}\; (S \to T) & = & \mathit{relobj}\; S \rightsquigarrow \mathit{relobj}\; T\ . \end{array}$$

As before *obj* is the interpretation of primitive types.

The relational translation is the same as that for functions, except that we use $\mathit{curry}_p$ and $\mathit{uncurry}_p$ in lieu of *curry* and *uncurry*:

$$\begin{array}{lcl} \mathit{reltrans}\; E & = & \mathit{rel}\; E\; [\,] \\ \mathit{rel}\; x\; \mathit{xs} & = & \mathit{snd} \cdot \mathit{fst}^i, \quad \text{where } \mathit{xs}_i \text{ is the first occurrence of } x \text{ in } \mathit{xs} \\ \mathit{rel}\; c\; \mathit{xs} & = & \mathit{relcon}\; c \cdot \mathit{fst}^n, \quad \text{where } n \text{ is the length of } \mathit{xs} \\ \mathit{rel}\; (F\; E)\; \mathit{xs} & = & \mathit{eval}_p \cdot \langle \mathit{rel}\; F\; \mathit{xs}, \mathit{rel}\; E\; \mathit{xs} \rangle \\ \mathit{rel}\; (\lambda x.E)\; \mathit{xs} & = & \mathit{curry}_p\; (\mathit{rel}\; E\; (x : \mathit{xs}))\ . \end{array}$$

The function *relcon* maps each constant $c : T$ to the corresponding relation of type $1 \to relobj\ T$.

We can now extend the translation to expressions that did not have a semantics before. Below we discuss three examples of new expression forms: many more are possible, and the reader may wish to try and define some of her own favourites for combining specifications.

**Choice.** Consider $E_1 \sqcup E_2$, the nondeterministic choice between $E_1$ and $E_2$. We define this via union of relations:

$$rel\,(E_1 \sqcup E_2)\,xs \;=\; (rel\,E_1\,xs) \cup (rel\,E_2\,xs)\ .$$

It is easy to check that with this definition, function application distributes through choice. In particular, we have that

$$(\lambda x.x - x)\,(0 \sqcup 1) \;=\; 0\ ,$$

so one could say that we have a call-by-value semantics for choice.

**General patterns.** As another example, consider the expression form for general pattern matching, $\lambda x : P \to E$. The idea is that $P$ can be any expression whatsoever: patterns are not restricted to injective constructor functions. The variable $x$ is the bound variable in the abstraction. In our example derivation, we used this device to split a list in an arbitrary way. The translation of these generalised abstractions is defined as follows:

$$rel\,(\lambda\,x : P \to E)\,xs \;=\; comp \cdot \langle rel\,(\lambda x.E)\,xs, conv \cdot rel\,(\lambda x.P)\,xs\rangle\ .$$

This definition uses two auxiliary functions, namely

$$\begin{array}{lcl} conv & : & (A \leadsto B) \to (B \leadsto A) \\ conv & = & \Lambda(swap \cdot \in) \\ comp & : & ((A \leadsto B) \times (C \leadsto A)) \to (C \leadsto B) \\ comp & = & curry_p(eval_p \cdot (id \times eval_p) \cdot assoc)\ . \end{array}$$

The function $swap : X \times Y \to Y \times X$ swaps the components of a pair. In the definition of *comp*, the function *assoc* is the obvious rebracketing isomorphism $(X \times Y) \times Z \to X \times (Y \times Z)$.

To illustrate, consider

$$rotate \;=\; (\lambda x : (fst\,x \mathbin{+\!\!+} snd\,x) \to snd\,x \mathbin{+\!\!+} fst\,x)\ .$$

This mapping takes a list, and returns any rotation of its argument.

**Fixpoints.** Because *all* terms are interpreted as relations, including the higher order ones, we can take fixpoints by introducing a new constant for each object $A$

$$\mathit{fix} \;:\; (A \to A) \to A \ ,$$

which is interpreted as

$$\begin{array}{lcl} \mathit{relcon}\,\mathit{fix} & : & 1 \to (A \rightsquigarrow A) \rightsquigarrow A \\ \mathit{relcon}\,\mathit{fix} & = & \mathit{curry}_p(\mathit{dom} \cdot \mathit{cap} \cdot (\mathit{relid} \times \mathit{id})) \ . \end{array}$$

That is, we intersect the argument relation with the identity, and then take the domain of that intersection to find its fixpoints; any fixpoint of $F$ is a possible result of $\mathit{fix}\,F$. The auxiliary operators in this definition are given by

$$\begin{array}{lcl} \mathit{cap} & = & \Lambda(\in \cdot\, \mathit{fst} \cap \in \cdot\, \mathit{snd}) \\ \mathit{dom} & = & \Lambda(\mathit{fst} \cdot \in) \\ \mathit{relid} & = & \mathit{curry}_p\; \mathit{snd} \ . \end{array}$$

Admittedly this treatment of fixpoints is a little unusual, because if there are multiple fixpoints, we simply select one of them nondeterministically. Furthermore, if there is no fixpoint of $F$, then $\mathit{fix}\,F$ is the empty relation. Of course it is also possible to define a least fixpoint operator of type

$$((A \to B) \to (A \to B)) \to (A \to B)$$

that returns a minimal fixpoint with respect to relation inclusion; the definitions are only slightly more involved than those given here.

### 3.4 Refinement

The possibility of nondeterminism means that we can consider refinement of relational expressions. We propose a definition modelling reduction of nondeterminism, but not enforcing preservation of domains; other definitions are possible.

Naturally, the definition involves inclusion, and the relational translation *reltrans*. For expressions $E_1, E_2 : T$ with free variables $xs$, we define

$$E_1 \sqsubseteq E_2 \;\equiv\; \mathit{reltrans}\; E_2\; \mathit{xs} \subseteq \mathit{ref}\; T \cdot \mathit{reltrans}\; E_1\; \mathit{xs} \ .$$

The 'mediation' $\mathit{ref}\; T$ is needed to cover higher-order cases. For example, we would like to have

$$\lambda x.x \sqcup x{+}1 \;\sqsubseteq\; \lambda x.x{+}1$$

but the relational translations of both sides (arrows $1 \to (\mathit{Int} \rightsquigarrow \mathit{Int})$ in $\mathit{Rel}$) are not related simply by inclusion. So we define $\mathit{ref}\; T : \mathit{relobj}\; T \to \mathit{relobj}\; T$ by induction over $T$ as follows:

$$\begin{array}{lcl} \mathit{ref}\; A & = & \mathit{id} \\ \mathit{ref}\; (S \times T) & = & \mathit{ref}\; S \times \mathit{ref}\; T \\ \mathit{ref}\; (S \to T) & = & \in \backslash\, ((\mathit{id} \times \mathit{ref}\; T) \cdot \in) \end{array}$$

where the operator '$\backslash$' (called 'weakest prespecification' in [20]) is defined by $(R \subseteq S\backslash T) \equiv (S \cdot R \subseteq T)$, or in points, $x\; (R\backslash S)\; y \equiv \forall z : z\; R\; x \Rightarrow z\; S\; y$.

Note that $\mathit{id} \subseteq \mathit{ref}\; T$ and $\mathit{ref}\; T \cdot \mathit{ref}\; T \subseteq \mathit{ref}\; T$, so $\sqsubseteq$ is a preorder. However, it is not a partial order; for example, the two expressions $(\lambda x.x \sqcup x{+}1)$ and $(\lambda x.x \sqcup x{+}1) \sqcup (\lambda x.x{+}1)$ refine to each other, but have different semantics.

### 3.5 Consistency of functional and relational semantics

Our relational semantics harbours a number of surprises for the unwary. For example, recall the illustration of the choice operator given above; it is not the case that $(\lambda x.x - x)\,(0 \sqcup 1)$ is equal to $(0 \sqcup 1) - (0 \sqcup 1)$ — the former always evaluates to 0, whereas the latter may evaluate to $-1$, 0 or 1. It follows that $\beta$-conversion is no longer a valid equality; however, we do have

$$(0 \sqcup 1) - (0 \sqcup 1) \;\sqsubseteq\; (\lambda x.x - x)\,(0 \sqcup 1)$$

reflecting the fact that the first expression is more nondeterministic than the second.

The validity of $\eta$-conversion

$$(\lambda x.E\;x) \;=\; E$$

is also compromised. Let $\mathit{twice} = \lambda f.f \cdot f$ and consider, for example,

$$\mathit{twice}\,(\mathit{id} \sqcup \mathit{succ}) \quad \text{and} \quad \mathit{twice}\,(\lambda x.(\mathit{id} \sqcup \mathit{succ})x)\;.$$

The expression on the left can evaluate to $\mathit{id}$ or $\mathit{succ} \cdot \mathit{succ}$. The expression on the right, however, offers the choice between $\mathit{id}$, $\mathit{succ}$ and $\mathit{succ} \cdot \mathit{succ}$. Like $\beta$-reduction, $\eta$-conversion can be weakened to an inequality.

Given these mildly surprising facts, and all the additional power that a relational semantics brings, it is necessary to pin down in what sense the relational semantics coincides with the functional semantics. Intuitively, we would like to express the fact that if we feed functional inputs into the relational semantics, the result is the same as feeding those inputs into the functional semantics, and converting the result to a relation. To make this precise, we need some further terminology.

**Positive and negative interpretation of types.** The *positive* interpretation of a type replaces every positive arrow with a relation exponent, and every negative arrow with a function exponent. Dually, the *negative* interpretation turns positive arrows into function exponents, and negative arrows into relation exponents. These two interpretations of types are defined by mutual recursion:

$$\begin{array}{lcl}
\mathit{obj}^+\,(S \to T) & = & \mathit{obj}^-\,S \leadsto \mathit{obj}^+\,T \\
\mathit{obj}^+\,(S \times T) & = & \mathit{obj}^+\,S \times \mathit{obj}^+\,T \\
\mathit{obj}^+\,A & = & \mathit{obj}\,A \\
\\
\mathit{obj}^-\,(S \to T) & = & \mathit{obj}^+\,S \Rightarrow \mathit{obj}^-\,T \\
\mathit{obj}^-\,(S \times T) & = & \mathit{obj}^-\,S \times \mathit{obj}^-\,T \\
\mathit{obj}^-\,A & = & \mathit{obj}\,A
\end{array}$$

We can now define four representation mappings to convert between the different interpretations of types that we have defined:

$$
\begin{tikzcd}
 & \mathit{funobj}\ T \arrow[dl, "\mathit{funrep}^{+}\ T"'] & \\
\mathit{obj}^{+}\ T & & \mathit{obj}^{-}\ T \arrow[ul, "\mathit{funrep}^{-}\ T"'] \arrow[dl, "\mathit{relrep}^{-}\ T"] \\
 & \mathit{relobj}\ T \arrow[ul, "\mathit{relrep}^{+}\ T"] &
\end{tikzcd}
$$

Intuitively, the negative occurrences of arrows are 'inputs' whereas the positive occurrences are 'outputs'. One could thus think of $\mathit{funrep}^{+}$ as converting all the output functions into relations. The other mappings can be given similar interpretations. To define these four mappings, we shall need to use the arrow $\rho : (A \Rightarrow B) \to (A \rightsquigarrow B)$ that was defined earlier.

$$
\begin{array}{lcl}
\mathit{funrep}^{+}\,(S \to T) & = & \rho \cdot (\mathit{funrep}^{-}\ S \Rightarrow \mathit{funrep}^{+}\ T) \\
\mathit{funrep}^{+}\,(S \times T) & = & \mathit{funrep}^{+}\ S \times \mathit{funrep}^{+}\ T \\
\mathit{funrep}^{+}\ A & = & \mathit{id} \\[1em]
\mathit{funrep}^{-}\,(S \to T) & = & \mathit{funrep}^{+}\ S \Rightarrow \mathit{funrep}^{-}\ T \\
\mathit{funrep}^{-}\,(S \times T) & = & \mathit{funrep}^{-}\ S \times \mathit{funrep}^{-}\ T \\
\mathit{funrep}^{-}\ A & = & \mathit{id} \\[1em]
\mathit{relrep}^{+}\,(S \to T) & = & \mathit{relrep}^{-}\ S \rightsquigarrow \mathit{relrep}^{+}\ T \\
\mathit{relrep}^{+}\,(S \times T) & = & \mathit{relrep}^{+}\ S \times \mathit{relrep}^{+}\ T \\
\mathit{relrep}^{+}\ A & = & \mathit{id} \\[1em]
\mathit{relrep}^{-}\,(S \to T) & = & (\mathit{relrep}^{+}\ S \rightsquigarrow \mathit{relrep}^{-}\ T) \cdot \rho \\
\mathit{relrep}^{-}\,(S \times T) & = & \mathit{relrep}^{-}\ S \times \mathit{relrep}^{-}\ T \\
\mathit{relrep}^{-}\ A & = & \mathit{id}
\end{array}
$$

Finally, we define for a list of typed variables

$$\mathit{list}\,[x_0 : T_0, \ldots, x_{n-1} : T_{n-1}]\,f \;=\; (((\mathit{id} \times f\ T_{n-1}) \times f\ T_{n-2}) \ldots f\ T_0)\,.$$

We can now formulate the following

**Theorem.** Suppose that for each constant $c : T$, we have

$$\mathit{relrep}^{+}\ T \cdot \mathit{relcon}\ c \;=\; \mathit{funrep}^{+}\ T \cdot \mathit{funcon}\ c\ ,$$

and furthermore there exists an arrow $\mathit{mincon}\ c : 1 \to \mathit{obj}^{-}\,T$ so that

$$
\begin{array}{lcl}
\mathit{funrep}^{-}\ T \cdot \mathit{mincon}\ c & = & \mathit{funcon}\ c \\
\mathit{relrep}^{-}\ T \cdot \mathit{mincon}\ c & = & \mathit{relcon}\ c\ .
\end{array}
$$

Let $E : T$ be an expression with free variables $xs$ that is built from variables, constants, $\lambda$-abstraction and application. Furthermore, assume that $E$ does not contain any $\beta$-redexes. Then we have

$$\begin{array}{l} \quad \mathit{funrep}^{+}\ T \cdot \mathit{fun}\ T\ \mathit{xs} \cdot \mathit{list}\ \mathit{xs}\ \mathit{funrep}^{-} \\ = \\ \quad \mathit{relrep}^{+}\ T \cdot \mathit{rel}\ T\ \mathit{xs} \cdot \mathit{list}\ \mathit{xs}\ \mathit{relrep}^{-}\ . \end{array}$$

This statement formalises our requirement that the relational semantics be consistent with the functional semantics. We believe it also holds true without the restriction to $\beta$-normal form, but we were unable to complete a proof before this paper had to go to press.

## 4 Epilogue

We do not advocate that the point free relational calculus be abandoned. Indeed, results such as the Converse Function Theorem are most conveniently proved, for all datatypes at once, in a point free style. Specific instantiations of generic results (in a point free style) tend to be painful however. The technical reason is that properties such as associativity, which we silently apply in a pointwise style, become exceedingly verbose. The suggestion to use nondeterministic mappings for program calculation was made earlier in a pioneering paper by Meertens [25]. Similar ideas, but in a slightly different spirit, can also be found in the CIP specification language [7, 8] and in refinement of expression languages [26, 30].

The semantics that we have sketched leaves many questions unanswered. In particular, there is an equivalence of categories between the category of typed lambda calculi with surjective pairing and the category of cartesian closed categories. To obtain a similar result for pointwise relational calculi, their logical properties have to be axiomatised. We have not even started to do this. It is hoped, however, that this paper might inspire others to join us in answering such questions.

We have benefited much from the pioneering work of Naumann and Martin on defining appropriate exponents for relations and predicate transformers [24, 27]. Here we have chosen a very simple notion of exponent, so that functions get mapped to their representation as binary relations, which in turn are represented as sets of pairs. We believe that for the purpose of deriving functional programs, the semantics is adequate and preferable to the more complicated constructions of [27]. Naumann aims at a semantics for higher-order imperative programs, which is a much more difficult problem.

## Acknowledgements

We would like to thank Richard Bird, Luke Ong and David Naumann for many enjoyable discussions on the topic of this paper. Lambert Meertens first kindled our interest in nondeterminacy in the late eighties, but at that time we did not have the courage to see it through.

## References


1. C. J. Aarts, R. C. Backhouse, P. F. Hoogendijk, E. Voermans, and J. C. S. P. Van der Woude. A relational theory of datatypes. September 1992.
2. A. Augusteijn. An alternative derivation of a binary heap construction function. In R. S. Bird, C. C. Morgan, and J. C. P. Woodcock, editors, *Mathematics of Program Construction*, volume 669 of *Lecture Notes in Computer Science*, pages 368–374. Springer-Verlag, 1992.
3. R. C. Backhouse. Making formality work for us. *EATCS Bulletin*, 38:219–249, 1989.
4. R. C. Backhouse, P. De Bruin, P. F. Hoogendijk, G. Malcolm, T. S. Voermans, and J. C. S. P. Van der Woude. Polynomial relators. In M. Nivat, C. S. Rattray, T. Rus, and G. Scollo, editors, *Algebraic Methodology and Software Technology*, Workshops in Computing, pages 303–362. Springer-Verlag, 1992.
5. J. Backus. Can programming be liberated from the Von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21:613–641, 1978.
6. M. Barr and C. Wells. *Category Theory for Computing Science*. International Series in Computer Science. Prentice Hall, 1990.
7. F. L. Bauer, R. Berghammer, M. Broy, W. Dosch, F. Geiselbrechtinger, R. Gnatz, E. Hangel, W. Hesse, B. Krieg-Brückner, A. Laut, T. Matzner, B. Möller, F. Nickl, H. Partsch, P. Pepper, K. Samelson, M. Wirsing, and H. Wössner. *The Munich Project CIP. Volume I: The Wide Spectrum Language CIP-L*, volume 183 of *Lecture Notes in Computer Science*. Springer-Verlag, 1985.
8. F. L. Bauer and H. Wössner. *Algorithmic Language and Program Development*. Texts and Monographs in Computer Science. Springer-Verlag, 1982.
9. R. Berghammer, P. Kempf, G. Schmidt, and T. Ströhlein. Relation algebra and logic of programs. In H. Andreka and J. D. Monk, editors, *Algebraic Logic*, volume 54 of *Colloquia Mathematica Societatis Janos Bolyai*, pages 37–58. North-Holland, 1991.
10. R. Berghammer and B. Von Karger. Formal derivation of CSP programs from temporal specifications. In *Mathematics of Program Construction*, volume 947 of *Lecture Notes in Computer Science*, pages 180–196. Springer-Verlag, 1995.
11. 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.
12. R. S Bird. *Introduction to Functional Programming in Haskell*. International Series in Computer Science. Prentice Hall, 1998.
13. R. S. Bird and O. De Moor. *Algebra of Programming*. International Series in Computer Science. Prentice Hall, 1996.
14. C. Brink and G. Schmidt, editors. *Relational Methods in Computer Science*. Springer-Verlag, 1996. Supplemental Volume of the Journal *Computing*, to appear.
15. W. Chen and J. T. Udding. Program inversion: More than fun! *Science of Computer Programming*, 15(1):1–13, 1990.
16. P. L. Curien. *Categorical Combinators, Sequential Algorithms and Functional Programming*. Research Notes in Theoretical Computer Science. Pitman, 1986.
17. E. W. Dijkstra. *A Discipline of Programming*. Series in Automatic Computation. Prentice Hall, 1976.

18. R. W. Floyd. Syntactic analysis and operator precedence. *Journal of the ACM*, 10(3):316–333, 1963.
19. D. Gries. *The Science of Programming*. Texts and Monographs in Computer Science. Springer-Verlag, 1981.
20. C. A. R. Hoare and J. He. The weakest prespecification, I. *Fundamenta Informaticae*, 9(1):51–84, 1986.
21. G. Jones and M. Sheeran. Circuit design in Ruby. In J. Staunstrup, editor, *Formal Methods for VLSI Design*, pages 13–70. Elsevier Science Publications, 1990.
22. E. Knapen. Relational programming, program inversion and the derivation of parsing algorithms. Computing science notes, Department of Mathematics and Computing Science, Eindhoven University of Technology, 1993.
23. J. Lambek and P. J. Scott. *Introduction to Higher Order Categorical Logic*, volume 7 of *Cambridge Studies in Advanced Mathematics*. Cambridge University Press, 1986.
24. C. E. Martin. *Preordered categories and predicate transformers*. D.Phil. thesis, Computing Laboratory, Oxford, UK, 1991.
25. L. Meertens. Algorithmics – towards programming as a mathematical activity. In J. W. De Bakker, M. Hazewinkel, and J. K. Lenstra, editors, *Mathematics and Computer Science*, volume 1 of *CWI Monographs*, pages 3–42. North-Holland, 1987.
26. J. M. Morris. Programming by expression refinement: The KMP algorithm. Chapter 37 of W. H. J. Feijen, A. J. M. van Gasteren, D. Gries, and J. Misra, editors, *Beauty is our Business*. Springer-Verlag, 1990.
27. D. A. Naumann. Towards squiggly refinement algebra. In D. Gries and W. P. De Roever, editors, *Programming Concepts and Methods*, pages 346–365. Chapman and Hall, 1998.
28. B. Schoenmakers. Inorder traversal of a binary heap and its inversion in optimal time and space. In R. S. Bird, C. C. Morgan, and J. C. P. Woodcock, editors, *Mathematics of Program Construction*, volume 669 of *Lecture Notes in Computer Science*, pages 291–301. Springer-Verlag, 1992.
29. D. R. Smith. Top-down synthesis of divide-and-conquer algorithms. *Artificial Intelligence*, 27(1):43–96, 1985.
30. N. T. E. Ward. *A refinement calculus for nondeterministic expressions*. Ph.D. thesis, University of Queensland, 1994.

## 5 Appendix: Haskell program

This is the full text of the program that we derived in Section 2.

```
data Tree a = Null | Bin (Tree a) a (Tree a)
  deriving Show

build :: Ord a => [a] -> Tree a
build = roll . foldr join []
  where
    join a t = search (Null,t)
    search (t,x) = if   bet a (t,x)
                   then (a,t):x
                   else search (shunt (t,x))

roll = foldl bin Null
  where
    bin s (a,t) = Bin s a t

shunt (s,(a,t):ts) = (Bin s a t,ts)

bet a (t,x) = above a t && fit a x

above a Null        = True
above a (Bin s b t) = a <= b

fit a []         = True
fit a ((b,t):u)  =  b <= a
```