An Exercise in Polytypic Program Derivation: *repmin*

Oege de Moor

September 17, 1996

Abstract

A program derivation is said to be *polytypic* if some of its parameters are data types. The *repmin* problem is to replace all elements of a tree of numbers by the minimum element, making only a single pass over the original tree. Here we present a polytypic derivation for that problem. The derivation has an unusual feature: when interpreted in the category of relations, the resulting program is the well-known cyclic logic program, and when interpreted in the category of functions, it is the well-known higher-order functional solution.

1 Motivation

Suppose I were to show you a derivation of a shortest path algorithm, and my whole presentation was in terms of numbers, addition and minimum. Undoubtedly some of you would get up and point out that by abstracting over the operations and recording their algebraic properties, I could have derived a whole class of algorithms instead of one particular program. Indeed, such abstraction over operations is now commonly accepted as one of the hallmarks of good program derivation: generality is increased, and the presentation is not cluttered by irrelevant particularities. The purpose of this paper is to persuade you that similar benefits can be reaped by abstracting over type constructors.

For concreteness, consider the well-known *repmin* problem [3]: it is required to change a tree t into a tree of the same shape as t, but with all elements replaced by the minimum element. There is a simple program to do that: first find the minimum, and then do the replacement. That simple program does however consist of two passes over the original tree t. The challenge is, therefore, to give a program that achieves the same effect in a single pass.

Now observe that I have not told you what kind of tree I had in mind here, and yet the problem statement makes perfect sense for any reasonable data structure. What we shall do in this paper, therefore, is to formalise the above problem statement as a specification that is parametrised by the type constructor of trees. We shall then derive a program that carries the same parameter; solutions for a specific tree type are obtained by instantiating that parameter.

To achieve this objective, we shall make use of some category theory, and in fact our derivation can be interpreted in different categories; if you like, the category is a parameter as well. One way of thinking about categories is that each of them represents a paradigm of computation: so one might naively say that the category of functions represents functional programming, and that the category of relations represents logic programming. At least in this example, that correspondence holds up: when our derivation is read as an argument about functions, we obtain the well-known higher-order solution to the *repmin* exercise. When it is read as a manipulation of relations, the result is the well-known cyclic logic program. While it seems likely that any categorical approach to program derivation would yield such paradigm-parametrised results, this is the first non-trivial example I have seen.

The bulk of this paper consists of a summary of the results we need; it is in fact meant as an introduction to polytypic program derivation. An in-depth account can be found in the textbook [4]. These results are all very general and not particular to the problem in hand. In fact, most were known to category theorists long before any applications to program derivation were realised. After we have summarised the necessary theory, the derivation itself consists of five steps: polytypism need not be unwieldy.

2 Categories

For the purposes of this paper we shall think of a category as a universe of types and specifications. Each specification has a target type and a source type; we write $f : A \leftarrow B$ to indicate that f has target A and source B. In accordance with this notation, specifications are sometimes called *arrows*. Specifications can be composed whenever intermediate types match up: that is, if $f : A \leftarrow B$ and $g : B \leftarrow C$, we have $f \cdot g : A \leftarrow C$. Composition is associative. Furthermore, for each type A, we have an identity $id : A \leftarrow A$, so that

$$f \cdot id = f = id \cdot f.$$

The prime example of a category is *Fun*. Here the types are sets, and the specifications are total functions. Composition is given by ordinary composition of functions.

Another example of a category is *Rel*. The types of *Rel* are also sets, but the arrows are binary relations, composed through

$$a(R \cdot S)c \quad \equiv \quad \exists b : a\,R\,b \wedge b\,R\,c.$$

A third example of a category is Fun^2, which is like *Fun*, but now the types are *pairs* of sets, and the arrows are pairs of total functions. Composition is defined componentwise:

$$(f,g)\cdot(h,k) \quad = \quad (f\cdot h, g\cdot k).$$

We can similarly take arbitrary powers of a given category to obtain new categories.

3 Functors

The types that are given as part of a category have no structure. For more interesting types we also need type constructors, for instance the type constructor *list* that, when applied to a set A, gives us the set $\mathit{list}\,A$ of all lists with elements drawn from A. Note that such a type constructor also gives rise, for each arrow $f : A \leftarrow B$, to another arrow $\mathit{list}\,f : \mathit{list}\,A \leftarrow \mathit{list}\,B$ that applies f to each element of a list.

All type constructors (such as *list*) are *functors*, which are homomorphisms of categories. To wit, let C and D be categories. A functor $F : C \leftarrow D$ is a mapping that consists of two components: one taking types to types, and another taking arrows to arrows. The requirement is that the structure of a category is preserved. That is, when $f : A \leftarrow B$ in D, we have $F f : F A \leftarrow F B$ in C. This states that typing information is preserved. Furthermore, a functor preserves composition and identities:

$$F\,id = id \quad \text{and} \quad F(h\cdot k) = F\,h \cdot F\,k.$$

For instance, consider the disjoint sum functor $(+) : \mathit{Fun} \leftarrow \mathit{Fun}^2$. For any two sets A and B, the disjoint sum $A + B$ is given by

$$A + B \quad = \quad \mathit{inl}\,A \mid \mathit{inr}\,B.$$

For two functions f and g, we define the sum by

$$\begin{aligned}(f+g)(\mathit{inl}\,a) &\quad = \quad \mathit{inl}(f\,a)\\ (f+g)(\mathit{inr}\,a) &\quad = \quad \mathit{inr}(g\,a).\end{aligned}$$

To say that disjoint sum preserves composition is to say that

$$(f+g)\cdot(h+k) \quad = \quad (f\cdot h)+(g\cdot k).$$

The proof that this identity is indeed satisfied is a simple matter of applying the definitions.

While discussing disjoint sums, we may as well take the opportunity to introduce the *case* operator $[f, g]$. It is defined by the equations

$$[f, g] \cdot inl = f \quad \text{and} \quad [f, g] \cdot inr = g.$$

As we shall see below, the case operator is extraordinarily useful in hiding irrelevant syntactic detail. One can set up quite a nice calculus for reasoning about the case construct, including its connection to conditional statements [4].

4 Products

Another example of a functor is the *product* functor $(\times) : Fun \leftarrow Fun^2$. On sets, we have that $A \times B$ is the cartesian product. On functions, product is defined by

$$(f \times g)(a, b) \quad = \quad (f\,a, g\,b).$$

Of course we can define a very similar functor $Rel \leftarrow Rel^2$ by setting

$$(a, b)(R \times S)(c, d) \quad = \quad a\,R\,c \wedge b\,S\,c$$

and this is clearly the generalisation of $\times$ from functions to relations. In fact, it can be shown under mild conditions that any functor $F : Fun^n \leftarrow Fun^m$ has precisely one generalisation $Rel^n \leftarrow Rel^m$ [4].

An important operator associated with products is the so-called *split*. It is defined by

$$\langle f, g \rangle\, a \quad = \quad (f\,a, g\,a).$$

We leave it to the reader to work out how this operator might be extended to relations. Note that, if we write *outl* and *outr* for the left and right projections respectively, we have for all functions f and g that

$$outl \cdot \langle f, g \rangle = f \quad \text{and} \quad outr \cdot \langle f, g \rangle = g.$$

Beware, however, that when f and g are generalised to arbitrary relations these equations do not hold.

The *product absorption* law shows how a product of arrows can be subsumed by a split:

$$(f \times g) \cdot \langle h, k \rangle \quad = \quad \langle f \cdot h, g \cdot k \rangle. \tag{1}$$

The product absorption law will play an important role in our derivation of the *repmin* problem. As an aside, we note that the product absorption law remains true when f, g, h and k are read as relations.

4.1 Cartesian closure

In the category of functions, there exists a one-to-one correspondence between arrows of type $A \leftarrow (B \times C)$ and arrows of type $(A \leftarrow C) \leftarrow B$. This correspondence is an old friend to functional programmers: it is what allows them to curry functions. Category theorists call this phenomenon *cartesian closure*.

One can define the following operators to capture the essence of cartesian closure:

$$\begin{array}{lcl} eval(f, a) & = & f\,a \\ curry f\, b\, c & = & f\,(b, c). \end{array}$$

The one-to-one correspondence is then stated in the equivalence

$$(eval \cdot (f \times id) = g) \quad \equiv \quad (f = curry\, g),$$

for all $f : (A \leftarrow C) \leftarrow B$ and $g : A \leftarrow (B \times C)$. In particular, we have the *curry cancellation* law

$$eval \cdot (curry\, g \times id) \quad = \quad g \tag{2}$$

which will be one of the key steps in our *repmin* derivation.

A little reflection will convince the reader that in the category of relations *Rel*, there is no one-to-one correspondence between arrows of type $(A \leftarrow C) \leftarrow B$ and $A \leftarrow (B \times C)$, at least not if we read $A \leftarrow C$ as the set of functions of type $A \leftarrow C$. However, we do have a very similar correspondence, namely between relations of type $(A \times C) \leftarrow B$ and relations of type $A \leftarrow (B \times C)$. In fact, if we define

$$\begin{array}{lcl} a\, eval((a', c'), c) & \equiv & a = a' \wedge c = c' \ \text{ and} \\ (a, c)(curry\, R)b & \equiv & aR(b, c), \end{array}$$

all equations quoted before about *eval* and *curry* still hold, with quantification over relations instead of total functions. (To the categorically wise I should mention that this does not mean that *Rel* is cartesian closed, for $(\times)$ is not a categorical product in *Rel*. The appropriate term for this situation is that *Rel* is *monoidal closed*.)

In our *repmin* derivation, we shall make use of *eval* and *curry*, and leave it open whether they should be interpreted functionally or relationally. As announced before, the functional interpretation will lead to a functional program, and the relational interpretation will give a logical solution.

5 Initial data types

The *repmin* exercise is about trees, so we need an appropriate way of modelling trees. For the purposes of this paper, we shall equate trees with arbitrary recursive data types. In category theory, recursive data types are modelled as follows.

Let $F : \mathit{Fun} \leftarrow \mathit{Fun}^2$ be a functor. We shall model trees as a type constructor $\mathit{fix}\,F$, called the *fixed point* of F (the text [4] speaks of *initial algebra* instead). To wit, the fixed point of F is an arrow $\mathit{in}\,F : \mathit{fix}\,F\,A \leftarrow F(A, \mathit{fix}\,F\,A)$ so that for each $h : A \leftarrow F(B, A)$, there exists an arrow $(\![h \mid F]\!) : A \leftarrow \mathit{fix}\,F\,B$. Furthermore, this arrow $(\![h \mid F]\!)$ should satisfy the equivalence

$$(g = (\![h \mid F]\!)) \quad \equiv \quad (g \cdot \mathit{in}\,F = h \cdot F(\mathit{id}, g)),$$

for all h of appropriate type. The arrow $(\![h \mid F]\!)$ is said to be the *catamorphism* of h and F, and F is called the *base functor* of $\mathit{fix}\,F$.

For concreteness, consider the following type of trees, defined in a style typical of functional programming languages:

$$\mathit{tree}\,A \quad = \quad \mathit{Tip}\,A \mid \mathit{Bin}(\mathit{tree}\,A, \mathit{tree}\,A).$$

The right hand side of this type definition corresponds to the base functor

$$F(A, T) \quad = \quad A + (T \times T).$$

Here $\mathit{fix}\,F$ is the type constructor *tree*, and $\mathit{in}\,F$ is the case expression $[\mathit{Tip}, \mathit{Bin}]$. The catamorphism $(\![[f, g] \mid F]\!)$ corresponds to the *fold* function

$$\begin{aligned} \mathit{fold}\,f\,g\,(\mathit{Tip}\,a) \quad &= \quad f\,a \\ \mathit{fold}\,f\,g\,(\mathit{Bin}(x, y)) \quad &= \quad f\,(\mathit{fold}\,f\,g\,x, \mathit{fold}\,f\,g\,y). \end{aligned}$$

Let us look at another example that illustrates the definition of $\mathit{fix}\,F$. A subtly different kind of tree is

$$\mathit{tree}'\,A \quad = \quad \mathit{Leaf}\,A \mid \mathit{Node}((\mathit{tree}'\,A, A), \mathit{tree}'\,A)$$

Here the base functor that encodes the right hand side is

$$F'(A, T) \quad = \quad A + ((T \times A) \times T), .$$

and we have that $\mathit{tree}' = \mathit{fix}\,F'$, and $[\mathit{Leaf}, \mathit{Node}] = \mathit{in}\,F'$. The catamorphism $(\![[f, g] \mid F']\!)$ corresponds to the fold' function

$$\begin{aligned} \mathit{fold}'\,f\,g\,(\mathit{Tip}\,a) \quad &= \quad f\,a \\ \mathit{fold}'\,f\,g\,(\mathit{Bin}((x, a), y) \quad &= \quad g\,((\mathit{fold}'\,f\,g\,x, a), \mathit{fold}'\,f\,g\,y) \end{aligned}$$

Note how the use of coproducts has enabled us to entirely hide the syntactical details of trees in the definition of catamorphisms. The ability to encapsulate unnecessary detail is one of the main advantages of a categorical style of program derivation.

One example of a technique that benefits by such encapsulation is *horizontal loop fusion*, also known as *tupling*. The key idea in that technique is that two iterators that both operate on the same structure can be merged into one. In terms of catamorphisms, this may be expressed by

$$\langle (\![h \mid F]\!), (\![k \mid F]\!) \rangle = (\![(h \times k) \cdot unzip\, F \mid F]\!) \tag{3}$$

$$unzip\, F = \langle F(id, outl), F(id, outr) \rangle. \tag{4}$$

On the left hand side we have a program that traverses its argument twice; by contrast, the right hand side achieves the same result with only one iteration. It is no surprise, therefore, that when we come to consider the *repmin* problem, this transformation (which is amusingly known as the *banana-split law*) will play a crucial role. A proof can be found in [4].

The recursion scheme provided by catamorphisms is, at first sight, rather limited. For example, consider the *rep* function on trees which takes a pair of a tree and an element and replaces all tips in the tree by that element:

$$\begin{aligned} rep(Tip\, a, b) &= Tip\, b \\ rep(Bin(x, y), b) &= Bin(rep(x, b), rep(y, b)). \end{aligned}$$

The definition of *rep* is very close to a catamorphism, but it does not quite fit the recursion pattern. Note, however, that if we define $crep = curry\, rep$, we have

$$\begin{aligned} crep(Tip\, a) &= Tip \\ crep(Bin(x, y)) &= \lambda b.Bin(crep\, x\, b, crep\, y\, b) \end{aligned}$$

so $crep = (\![Tip, g \mid F]\!)$, with $f(h, k) = \lambda b.Bin(h\, b, k\, b)$. This kind of juggling with λ-abstractions is however specific to the category of functions. If we also want a relational program, we have to give even more painful definitions. Fortunately, we can express the above idea about *rep* and *crep* in terms of *curry*, *uncurry* and *eval* alone:

Fact 1 *(Structural Recursion Theorem) Suppose that ϕ satisfies*

$$F(id, h \times id) \cdot \phi = \phi \cdot (F(id, h) \times id) \quad \text{, for all } h.$$

Then the equation

$$x \cdot (in\, F \times id) = g \cdot F(id, x) \cdot \phi$$

has a unique solution in x, given by

$$x = uncurry (\![curry(g \cdot F(id, eval) \cdot \phi) \mid F]\!).$$

A proof of this fact can be found in [4]. Informally, it says that the recursion scheme of catamorphisms can be augmented by an extra parameter. The advantage of expressing it in terms of *curry*, *uncurry* and *eval* is that it works in any category where these operations are defined, in particular in both *Fun* and *Rel*. Below we shall use the above fact to give a general definition of *rep* that does not rely on a specific datatype.

We close our discussion of tree types by noting that the fixed point construction gives rise to a functor. For let $h : A \leftarrow B$; we can then define $\mathit{fix}\,F\,h : \mathit{fix}\,F\,A \leftarrow \mathit{fix}\,F\,B$ by

$$\mathit{fix}\,F\,h \quad = \quad (\!| \mathit{in}\,F \cdot F(h, \mathit{id}) \mid F |\!).$$

Again, we omit the proof that $\mathit{fix}\,F$ does indeed preserve identities and composition. For any function h, the function $\mathit{fix}\,F\,h$ applies h to all elements of a tree. For example, consider the function *maptree*

$$\begin{aligned} \mathit{maptree}\,h\,(\mathit{Tip}\,a) &\quad = \quad \mathit{Tip}\,(h\,a) \\ \mathit{maptree}\,h\,(\mathit{Bin}\,(x, y)) &\quad = \quad \mathit{Bin}\,(\mathit{maptree}\,h\,x, \mathit{maptree}\,h\,y). \end{aligned}$$

For the base functor F that was defined above, we have $\mathit{maptree} = \mathit{fix}\,F$.

6 Distribute and replace

The last result of the previous section assumed the existence of a collection of arrows $\phi : F(A, B \times C) \leftarrow (F(A, B) \times C)$ satisfying the equation

$$F(\mathit{id}, h \times \mathit{id}) \cdot \phi \quad = \quad \phi \cdot (F(\mathit{id}, h) \times \mathit{id}),$$

for all h. Elsewhere, Paul Hoogendijk and I have shown that there exists a canonical construction of such a ϕ for any data type [7]. More precisely, for any functor $H : \mathit{Fun} \leftarrow \mathit{Fun}$ that models a data type, we have a collection of arrows $\mathit{dist}\,H : H\,(A \times B) \leftarrow (H\,A \times B)$ that satisfies

$$H(f \times \mathit{id}) \cdot \mathit{dist}\,H \quad = \quad \mathit{dist}\,H \cdot (H f \times \mathit{id}).$$

Informally, $\mathit{dist}\,H$ takes some data structure x and an element a, and pairs up every element of x with a. In categorical jargon $\mathit{dist}\,H$ is called *strength* of H, but in the present context that terminology seems inappropriate; I prefer to speak of a *distributor*. The distributor of a fixed point can be described via the general recursion scheme of Fact 1.

Fact 2 *We have* $h = \mathit{dist}\,(\mathit{fix}\,F)$ *iff*

$$h \cdot (\mathit{in}\,F \times \mathit{id}) \quad = \quad \mathit{in}\,F \cdot F(\mathit{id}, h) \cdot \mathit{dist}\,(F \cdot \mathit{fork}).$$

Here *fork* is the obvious functor $C^2 \leftarrow C$ that doubles its arguments. A proof can be found in [6].

With the introduction of distributors, we are getting closer to the *repmin* problem. For consider the operation $rep : H\,A \leftarrow H\,B \times A$, that takes a data structure x and an element a, and returns a data structure y that has the same shape as x, but every element of x has been replaced by a. We can program *rep* in terms of *dist*:

$$rep\,H \quad = \quad H\,outr \cdot dist\,H\,.$$

Unlike our earlier definition of *rep*, which only worked for a specific kind of tree, this definition makes sense for any tree type whatsoever. The remarks we made for that specific instance about currying and catamorphisms generalise to:

$$\begin{array}{l} curry\,(rep\,(\mathit{fix}\,F)) \\ = \\ (\!|\, curry(in\,F \cdot F(outr, eval) \cdot dist\,(F \cdot fork)) \mid F \,|\!). \end{array} \tag{5}$$

Let us prove the above equation to illustrate the preceding definitions and results. First, observe that

$$\begin{array}{cl}
 & rep\,(\mathit{fix}\,F) \cdot (in\,F \times id) \\
= & \quad \{\text{definition of } rep\} \\
 & \mathit{fix}\,F\,outr \cdot dist\,(\mathit{fix}\,F) \cdot (in\,F \times id) \\
= & \quad \{dist \text{ of fixed point}\} \\
 & \mathit{fix}\,F\,outr \cdot in\,F \cdot F(id, dist\,F) \cdot dist\,(F \cdot fork) \\
= & \quad \{\text{since } \mathit{fix}\,F\,h = (\!|\, in\,F \cdot F(h, id) \mid F \,|\!)\} \\
 & \quad in\,F \cdot F(outr, id) \cdot F(id, \mathit{fix}\,F\,outr) \cdot \\
 & \quad F(id, dist\,(\mathit{fix}\,F)) \cdot dist\,(F \cdot fork) \\
= & \quad \{F \text{ is a functor}\} \\
 & in\,F \cdot F(outr, \mathit{fix}\,F\,outr \cdot dist\,(\mathit{fix}\,F)) \cdot dist\,(F \cdot fork) \\
= & \quad \{\text{definition of } rep\} \\
 & in\,F \cdot F(outr, rep\,(\mathit{fix}\,F)) \cdot dist\,(F \cdot fork)
\end{array}$$

Equation (5) is now immediate from Fact 1. It is quite conceivable that this fact about *rep* has other uses beside the *repmin* exercise, so the above calculation is not counted towards the cost of a polytypic solution to *repmin*.

7 Minimum

The only ingredient of the *repmin* exercise that is still missing is the notion of minimum. We could define this with absolute rigour in the relational calculus

[4], but that would amount to formal overkill for our present purposes, so we take a little shortcut.

First, it can be shown [7] that for any functor F that models a data type, there exists a function $\mathit{setify} : P\,A \leftarrow F\,A$ that turns a data structure into its set of elements. Now define $\mathit{min}\,F : N \leftarrow FN$ as the partial function that returns the minimum element from that set of elements. Here N stands for the set of natural numbers, but in fact any well-ordered set would do.

One can prove that minimums of trees can be found in a tree traversal

$$\mathit{min}\,(\mathit{fix}\,F) \quad \supseteq \quad (\!|\mathit{min}\,(F \cdot \mathit{fork}) \,|\, F|\!). \tag{6}$$

Note, however, that this is an inequation, and not an identity. The reason is that some subtrees may contain no elements at all, and then the minimum function is not defined. However, if we can guarantee that all substructures have an element, the partial function min can be proved total:

Fact 3 *If $F(0,0) = 0$ then $\mathit{min}\,(\mathit{fix}\,F)$ and $\mathit{min}\,(F \cdot \mathit{fork})$ are total functions.*

Here 0 is the empty set, or in categorical vernacular: the initial object. If $\mathit{min}\,(\mathit{fix}\,F)$ is a total function, the inclusion of (6) can be strengthened to an equality, for two functions that are included in each other are necessarily equal.

8 Derivation

For any kind of tree $\mathit{fix}\,F$, we can now define the *repmin* exercise by

$$\mathit{repmin} \quad : \quad \mathit{fix}\,F\,N \leftarrow \mathit{fix}\,F\,N \tag{7}$$

$$\mathit{repmin}\,(\mathit{fix}\,F) \quad = \quad \mathit{rep}\,(\mathit{fix}\,F) \cdot \langle \mathit{id}, \mathit{min}\,(\mathit{fix}\,F)\rangle. \tag{8}$$

The split $\langle \mathit{id}, \mathit{min}\,(\mathit{fix}\,F)\rangle$ returns a pair of the original tree and the minimum element. As we have seen, that minimum element can only be guaranteed to exist if $F(0,0) = 0$, so we shall assume that this condition is satisfied.

The specification of *repmin* embodies two tree traversals: one for the computation of *rep*, and another one for the computation of *min*. Our aim is, therefore, to write *repmin* in terms of a single catamorphism:

$$\begin{array}{ll}
 & \mathit{repmin}\,(\mathit{fix}\,F) \\
= & \quad \{ \text{ definition (8) } \} \\
 & \mathit{rep}\,(\mathit{fix}\,F) \cdot \langle \mathit{id}, \mathit{min}\,(\mathit{fix}\,F)\rangle \\
= & \quad \{ \text{ exponentials (2) } \} \\
 & \mathit{eval} \cdot (\mathit{curry}(\mathit{rep}\,(\mathit{fix}\,F)) \times \mathit{id}) \cdot \langle \mathit{id}, \mathit{min}\,(\mathit{fix}\,F)\rangle \\
= & \quad \{ \text{ product absorption (1) } \}
\end{array}$$

$$
\begin{array}{cl}
 & eval \cdot \langle curry\,(rep\,(fix\,F)), min\,(fix\,F)\rangle \\
= & \quad \{ \text{ see below: } curry(rep(fix\,F)) = (\![f \mid F]\!), \text{ and } min(fix\,F) = (\![g \mid F]\!) \} \\
 & eval \cdot \langle (\![f \mid F]\!), (\![g \mid F]\!)\rangle \\
= & \quad \{ \text{ banana split (3) } \} \\
 & eval \cdot (\![(f \times g) \cdot unzip\,F \mid F]\!)
\end{array}
$$

In the penultimate step, the definitions of f and g can be copied from Equations (5) and (6). Note that the resulting expression does indeed specify only a single traversal of the argument.

9 Instantiation

For the benefit of those who do not do their daily programming in terms of categorical combinators, we now instantiate the result of our derivation in a particular case. We shall consider trees whose base functor is given by

$$F(A,B) \quad = \quad A + (B \times B).$$

9.1 Haskell

We first consider the functional program, and we express it in the functional language *Haskell*. We should note that *Fun* is not an appropriate model of Haskell, but any model of functional programming is likely to contain all total functions between datatypes, so in that sense all programs derived in *Fun* can be written in Haskell. The relevant type of trees is introduced by

```
data Tree a = Tip a | Bin (Tree a) (Tree a)
```

The program itself consists of two stages: first a pair of a function f and the minimum m is constructed, and then f is applied to m:

```
repmin x = f m
           where (f,m) = rm x
```

It is the function `rm` that implements the catamorphism that does the single pass over the input tree `x`:

```
rm (Tip a) = (Tip,a)
rm (Bin x y) =  (\ m -> Bin (fx m) (fy m),min mx my)
                where (fx,mx) = rm x
                      (fy,my) = rm y
```

This is essentially the program obtained by Pettorossi in [12]. Another derivation is [13], and that paper also shows by experiments that (if we discount the cost of creating closures) the above program is better than the original specification.

9.2 Prolog

Whereas the correspondence between categorical combinators and Haskell is fairly obvious, a little more work is needed for a translation into a logic language like Prolog. The reason is that logic languages tend to lack all forms of compositionality (see, however, [10] for a logic language that overcomes this problem). The formal correspondence between Prolog and *Rel* is therefore even more dubious then the correspondence between Haskell and *Fun*. Recall the definition of *eval* and *curry* in *Rel*:

$$\begin{array}{rcl} a\,eval((a',c'),c) & \equiv & a = a' \wedge c = c' \ \text{ and} \\ (a,c)(curry\,R)b & \equiv & aR(b,c). \end{array}$$

Using this, the Prolog program becomes

```
repmin(X,Y) :- rm(X,A,A,Y) .

rm(tip(A),A,B,tip(B)) .
rm(bin(X,Y),A,C,bin(U,V)) :- rm(X,A,B0,U), rm(Y,A,B1,V), min(C,B0,B1) .
```

This program is sometimes said to be cyclic, because the parameter `A` in the first call of `rm` is used both for input (its first occurrence) and for output (the second occurrence).

10 Discussion

We have given an example of a program derivation that is parametrised by the type constructor of trees. Why is such an example interesting? Perhaps the most obvious advantage is the generality obtained, but I believe there is another, more important implication: the derivation is entirely guided by the mathematical properties of the components of the specification. In fact, I have a confession to make here: when I started to write this paper, I first wrote out the relational polytypic solution. Although equivalent to the formulation derived here, it was a big mess of combinators, and I very nearly abandoned the idea of deriving it before even having started. It was a delight, therefore, to find that by the 5-step calculation of Section 8, one arrives at a very compact program that makes sense for functions and relations alike. This experience gives good hope that at least some of the polytypic programs that have recently appeared in the literature [8, 9] can be significantly improved and shortened by deriving them in a rigorous manner.

The first non-trivial examples of polytypic program derivation were [2, 5], and several more can be found in [4]. There has been a lot of work that provides us with the plumbing for writing polytypic programs, *e.g.* [1, 11]. Such plumbing plays the same role in polytypic programming that more elementary control structures (*e.g.* *while* loops) do in the *wp*-calculus. Now that the foundations seem to have stabilised, I hope that others will join the effort and exploit those foundations in concrete programming exercises. Suggestions for such exercises can be found in [4].

There are many more little programs where the functional and logical solutions can be derived simultaneously: a particularly rich source of examples are programs that use *accumulation parameters* (in the functional vernacular) and programs that use *difference lists* (in the logical vernacular). The sense in which accumulation parameters and difference lists are actually the same is explained by the monoidal closure properties of *Fun* and *Rel*.

Acknowledgements

Thanks to Richard Bird for countless discussions on polytypic programming, and for his efforts to improve my style. Graham Hutton, Doaitse Swierstra and Bernard Sufrin also suggested improvements over earlier drafts of this paper. I am grateful to Paul Hoogendijk for collaboration on the finer points of polytypism. Finally, Johan Jeuring invented the stomach-turning term *polytypism*, and spread it so widely that I was forced to adopt it in this paper. What I meant throughout was, of course, higher-order polymorphism.

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. Available from URL `http://www.win.tue.nl/win/cs/wp/papers/papers.html`, September 1992.

[2] R. C. Backhouse, H. Doornbos, and P. Hoogendijk. A class of commuting relators. Available from URL `http://www.win.tue.nl/win/cs/wp/papers/papers.html`.

[3] R. S Bird. The promotion and accumulation strategies in functional programming. *ACM Transactions on Programming Languages and Systems*, 6(4):487–504, 1984.

[4] R. S. Bird and O. De Moor. *Algebra of Programming*. International Series in Computer Science. Prentice Hall, 1996.

[5] R. S. Bird, P. F. Hoogendijk, and O. De Moor. Generic programming with relations and functors. *Journal of Functional Programming*, 6(1):1–28, 1996.

[6] J. R. B. Cockett and D. Spencer. Strong categorical datatypes I. In R. A. G. Seely, editor, *Category Theory 1991*, volume 13 of *CMS Conference Proceedings*, pages 141–169. Canadian Mathematical Society, 1992.

[7] P. F. Hoogendijk and O. De Moor. What is a data type? Submitted for publication, 1996.

[8] J. Jeuring and P. Jansson. Polytypic programming. In *Second International Summer school on Advanced Functional Programming*, volume To appear of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[9] J. T. Jeuring. Polytypic pattern matching. In S. Peyton-Jones, editor, *Functional Programming Languages and Computer Architecture*, pages 238–248. Association for Computing Machinery, 1995.

[10] R. M. McPhee and O. De Moor. Compositional logic programming. Submitted for publication, 1996.

[11] L. Meertens. Calculate polytypically. In *Proceedings of the Eighth International Symposium on Programming Languages, Implementations, Logics and Programs*, Lecture Notes in Computer Science. Springer-Verlag, 1996.

[12] A. Pettorossi. Derivation of programs which traverse their input data only once. In G. Cioni and A. Salwicki, editors, *School on Programming Methodologies*, pages 165–184. Academic Press, 1987.

[13] M. Takeichi. Partial parametrization eliminates multiple traversals of data structures. *Acta Informatica*, 24:57–77, 1987.