Calculate Polytypically!

Lambert Meertens
lambert@cwi.nl

Department of Algorithmics and Architecture, CWI, Amsterdam, and
Department of Computing Science, Utrecht University, The Netherlands

Abstract. A polytypic function definition is a function definition that is parametrised with a datatype. It embraces a *class* of algorithms. As an example we define a simple polytypic "crush" combinator that can be used to calculate polytypically. The ability to define functions polytypically adds another level of flexibility in the reusability of programming idioms and in the design of libraries of interoperable components.

1 Introduction

Which is more exciting: to find yet another algorithm, or to discover that two familiar algorithms are instances of one more abstract algorithm?

It is the latter that sparks new insight and opens the way for finding further connections, that makes it possible to organise and systematise our knowledge and eventually set as routine exercises problems that once were feats of scientific discovery. Mathematics likewise gets its leverage from abstraction, by going from the specific to the general. Essential to the expression of abstraction is the ability to parametrise.

John Hughes argues in [15] that the ability to name and reuse — i.e., to parametrise — is at the heart of the functional languages' power. Standard combinators (higher-order functions) like map and foldr capture very general programming idioms that are useful in almost any context. Polymorphic typing enables us to use the same programming idiom to manipulate data of different types.

The next step is the ability to *parametrise a function definition with a type*. A function thus parametrised is called *polytypic*. The "derived" functions of Haskell are all polytypic, as are catamorphisms and friends [24] [29] [31]. The standard foldr combinator is just the instantiation of the cata combinator for the datatype constructor *List*.

While a polymorphic function stands for one algorithm that happens to be insensitive to what type the values in some structure are, a polytypic function embraces a *class* of algorithms.

The ability to define functions polytypically adds another level of flexibility in the reusability of programming idioms and in the design of libraries of interoperable components. This, I claim, is of tremendous importance. Yet the greatest gain, I believe, is to come from the ability to *reason* polytypically in the process of deriving programs, in particular by calculational methods.

2 So what is polytypy?

Here are a few datatype constructor definitions[1]:

$$\textbf{data}\ \ List\ a\ =\ \mathsf{cons}\ a\ (List\ a)\ \mid\ \mathsf{nil}$$

$$\textbf{data}\ \ Maybe\ a\ =\ \mathsf{one}\ a\ \mid\ \mathsf{none}$$

$$\textbf{data}\ \ Bin\ a\ =\ \mathsf{join}\ (Bin\ a)\ (Bin\ a)\ \mid\ \mathsf{tip}\ a$$

$$\textbf{data}\ \ Rose\ a\ =\ \mathsf{fork}\ a\ (List(Rose\ a))$$

Each of these types has its own map combinator, for which we only give the typings:

$$\begin{array}{ll}
\mathsf{map}_{List} & \in\ (List\ a \leftarrow List\ b) \leftarrow (a \leftarrow b) \\
\mathsf{map}_{Maybe} & \in\ (Maybe\ a \leftarrow Maybe\ b) \leftarrow (a \leftarrow b) \\
\mathsf{map}_{Bin} & \in\ (Bin\ a \leftarrow Bin\ b) \leftarrow (a \leftarrow b) \\
\mathsf{map}_{Rose} & \in\ (Rose\ a \leftarrow Rose\ b) \leftarrow (a \leftarrow b)
\end{array}$$

Here are functions to test if a given value occurs in a data structure of one of these types.

$$\begin{array}{llcl}
e \in_{List} & \mathsf{cons}\ u\ x & = & \mathsf{eq}\ e\ u\ \ \vee\ \ e \in_{List} x \\
e \in_{List} & \mathsf{nil} & = & \mathsf{false} \\
\\
e \in_{Maybe} & \mathsf{one}\ u & = & \mathsf{eq}\ e\ u \\
e \in_{Maybe} & \mathsf{none} & = & \mathsf{false} \\
\\
e \in_{Bin} & \mathsf{join}\ x\ y & = & e \in_{Bin} x\ \ \vee\ \ e \in_{Bin} y \\
e \in_{Bin} & \mathsf{tip}\ u & = & \mathsf{eq}\ e\ u \\
\\
e \in_{Rose} & \mathsf{fork}\ u\ xs & = & \mathsf{eq}\ e\ u\ \ \vee\ \ \mathsf{any}\ (e \in_{Rose})\ xs
\end{array}$$

And here are functions to sum the elements in one of these structures — assuming they are numbers.

$$\begin{array}{llcl}
\mathsf{sum}_{List} & (\mathsf{cons}\ u\ x) & = & u\ +\ \mathsf{sum}_{List}\ x \\
\mathsf{sum}_{List} & \mathsf{nil} & = & 0 \\
\\
\mathsf{sum}_{Maybe} & (\mathsf{one}\ u) & = & u \\
\mathsf{sum}_{Maybe} & \mathsf{none} & = & 0
\end{array}$$

[1] Examples are in a pidgin based on functional languages like Haskell and Gofer. In particular the lexemic restrictions on constructor functions of these languages are not adhered to. To indicate the typing of a function, I write $f\ \in\ a \leftarrow b$ instead of $f:\ b \rightarrow a$. The advantage of this convention is that this matches the "backwardness" of composition, making it easier to assess the function typing of a composition.

$$
\begin{array}{lllll}
\mathsf{sum}_{Bin} & (\mathsf{join}\ x\ y) & = & \mathsf{sum}_{Bin}\ x\ +\ \mathsf{sum}_{Bin}\ y \\
\mathsf{sum}_{Bin} & (\mathsf{tip}\ u) & = & u \\
\\
\mathsf{sum}_{Rose} & (\mathsf{fork}\ u\ xs) & = & u\ +\ \mathsf{sum}_{List}\ (\mathsf{map}_{List}\ \mathsf{sum}_{Rose}\ xs)
\end{array}
$$

Polytypy, now, allows us to replace all these definitions by a single definition for map_F, a single definition for $\in_F$ and a single definition for sum_F, each of which can be specialised to any of the above datatype constructors and many more by taking F to be *List*, *Maybe*, *Bin*, and so on.

Polytypy is orthogonal to polymorphism. The polytypic function map_F is truly polymorphic — that is, each of its instantiations is. The polytypic functions $\in_F$ and sum_F are as polymorphic as eq and $+$ are, which is, respectively, somewhat and hardly. However, eq is — or can be defined as — a polytypic function; see e.g. Sheard [34].

Other terms that have been used for the same concept are "structural polymorphism" (Ruehr [33]), "generic programming" (de Moor [5], Bird, de Moor and Hoogendijk [4]) and "type parametric programming" (Sheard [34]).

3 Some historical remarks

In what I'll refer to as "classic BMF" [28] [2], a.k.a. "Squiggol", the focus was on lists, with particular emphasis on a symmetric view in which lists are built up from the empty-list constructor $[\,]$, the singleton-list constructor $[\_]$, and an associative constructor $+\!\!+$. Catamorphisms on these symmetric lists were written, in the most general case, in the form $\oplus/\cdot f*$ (a "reduce" after a "map"), which requires $\oplus$ to be an associative operator with some neutral element $\nu_\oplus$. In other words, $(\oplus, \nu_\oplus)$ constitutes a monoid, just like $(+\!\!+, [\,])$ does. The meaning is then inductively defined by:

$$
\begin{array}{ll}
\oplus/\cdot f* = h\ \textbf{where} & \\
\quad h\ (x +\!\!+ y) & = h\ x\ \oplus\ h\ y \\
\quad h\ [u] & = f\ u \\
\quad h\ [\,] & = \nu_\oplus
\end{array}
$$

It is possible to leave $\nu_\oplus$ implicit since neutral elements — if they exist — are unique.

These notations were devised with one purpose only: to facilitate the derivation of programs by *calculation*. In spite of the focus on lists, the intention, from the start, has been to contribute to the development of "constructive algorithmics" as a discipline for calculational program construction encompassing much more than the theory of lists, however fertile by itself.

Malcolm [24] [25] [26] showed how to generalise essential parts of the theory to other initial datatypes, based on a categorical approach (Manes and Arbib [27], Hagino [13]). Fokkinga [7] [8] [11] honed the categorically-inspired calculational techniques to a fine edge.

While the theory developed by Malcolm and Fokkinga gave the basic tools needed for polytypic definitions, its application to deriving actual programs by calculation was initially largely confined to instantiations for, each time, one specific datatype.

The first calculational derivation of an actual polytypic algorithm that I saw, and an elegant one at that, was the one in Bird, de Moor and Hoogendijk [4]. Earlier work by Bird and de Moor on solving a variety of optimisation problems by calculation was polytypically unified by de Moor in [5]. Several further examples of polytypic calculations can be found in Bird and de Moor [3].

The most impressive polytypic algorithms today are those developed by Jeuring and his group, such as Jeuring's polytypic pattern-matching algorithm [21]. Jansson [17] presents a polytypic unification algorithm (see also Jansson and Jeuring [19]). Although not derived calculationally, these algorithms provide strong evidence of the potential of polytypic definitions.

Huisman [16] defines a polytypic function $\mathsf{unparser}$ — rather like polytypic $\mathsf{flatten}$ but with extra "hooks" for plugging in concrete syntax — and calculates a polytypic parser from it by function inversion. By defining a suitable intermediate abstract data type, the textual representation of a structured document can be changed by a composition $\mathsf{unparser} \cdot \mathsf{parser}$.

4 Notation and terminology

The notation $(x :: e)$, in which the expression e may depend on the dummy x, denotes the same as the lambda form $(\lambda x \mapsto e)$. For any e, e^{K} denotes the constant function that maps all arguments to e. Function id_a is the identity function restricted to type a. The datatype 1 stands for some one-element type, like that defined by:

$$\textbf{data}\ 1 \ = \ \mathsf{blob}$$

Functor. An n-ary *functor*[2] F is a combinator that maps an n-tuple of functions $f_0, \ldots, f_{n-1}$ to a function $F\ f_0\ \cdots\ f_{n-1}$ in such a way that composition and identities are respected:

$$F\ (f_0 \cdot g_0)\ \cdots\ (f_{n-1} \cdot g_{n-1}) \ = \ F\ f_0 \cdots f_{n-1} \ \cdot \ F\ g_0 \cdots g_{n-1}$$
provided that $f_i \in a_i \leftarrow b_i$ and $g_i \in b_i \leftarrow c_i$

$$F\ \mathsf{id}\ \cdots\ \mathsf{id} \ = \ \mathsf{id}$$

The clause concerning the typing serves to ensure the definedness of the compositions.

An example are the functions map_F, since they satisfy the functional identities $\mathsf{map}_F(f \cdot g) \ = \ \mathsf{map}_F\, f \ \cdot \ \mathsf{map}_F\ g$ and $\mathsf{map}_F\ \mathsf{id} \ = \ \mathsf{id}$. So they are unary functors. As is easily verified, ${\mathsf{id}_a}^{\mathsf{K}}$ is also a functor. It is n-ary for all n. Further, each extraction combinator

[2] The terminology is borrowed from category theory, but no knowledge of category theory is needed to follow the exposition here. Gentle introductions to category theory that are inspired by its use for program calculation can be found in [9] and [30].

$$\mathsf{Ex}_i^n\ f_0\ \cdots\ f_{n-1} \;=\; f_i,\ i \;=\; 0,\ldots,n-1$$

is an n-ary functor. We write Id for the unary functor Ex_0^1, and Exl and Exr for the binary functors Ex_0^2 and Ex_1^2.

An n-ary functor induces a mapping on n-tuples of types. Let, for $f_i \in a_i \leftarrow b_i,\ i \;=\; 0,\ldots,n-1$, the (most general) typing of $F\ f_0\ \cdots\ f_{n-1}$ be given by

$$F\ f_0\ \cdots\ f_{n-1} \in A \leftarrow B$$

Then we denote these types A and B by

$$\begin{aligned} F\ a_0 \cdots a_{n-1} &= A \\ F\ b_0 \cdots b_{n-1} &= B \end{aligned}$$

So for unary functor F we have

$$F\ f \in F\ a \leftarrow F\ b \ \Leftarrow\ f \in a \leftarrow b$$

Looking at the typing of map_F:

$$\mathsf{map}_F\ f \in F\ a \leftarrow F\ b \ \Leftarrow\ f \in a \leftarrow b$$

we see that the type mapping induced is F, i.e., $(a :: F\ a)$. We shall from here on use the *same* notation for the combinator and for its induced type mapping. Moreover, when applicable, we use the name of the type mapping for that. So, from here on, for function f, we write $\mathit{List}\ f$ rather than $\mathsf{map}_{\mathit{List}}\ f$. Likewise, we write a^{K} instead of $\mathsf{id}_a{}^{\mathsf{K}}$.

To introduce polytypic definitions, we need to abstract from the constructor function names. Here are some basic functors that will be helpful, together with some auxiliary functions.

The sum functor. The binary sum functor $+$ is given by:

$$\mathbf{data}\ \ a+b \;=\; \mathsf{inl}\ a \ \mid\ \mathsf{inr}\ b$$

$$\begin{aligned} f+g \;&=\; h\ \ \mathbf{where} \\ &\qquad h(\mathsf{inl}\ u) \;=\; \mathsf{inl}(f\ u) \\ &\qquad h(\mathsf{inr}\ v) \;=\; \mathsf{inl}(g\ v) \\ f \triangledown g \;&=\; h\ \ \mathbf{where} \\ &\qquad h(\mathsf{inl}\ u) \;=\; f\ u \\ &\qquad h(\mathsf{inr}\ v) \;=\; g\ v \end{aligned}$$

The following typing rule will be used:

$$f \triangledown g \in c \leftarrow a+b \quad \Leftarrow \quad f \in c \leftarrow a \ \wedge\ g \in c \leftarrow b$$

The product functor. The binary product functor $\times$ is given by:

$$\mathbf{data}\ \ a \times b \ = \ \mathsf{pair}\ a\ b$$

$$\begin{aligned} f \times g \ &= \ h \ \ \mathbf{where} \\ &\qquad h(\mathsf{pair}\ u\ v) \ = \ \mathsf{pair}\ (f\ u)\ (g\ v) \\ \mathsf{exl}(\mathsf{pair}\ u\ v) \ &= \ u \\ \mathsf{exr}(\mathsf{pair}\ u\ v) \ &= \ v \end{aligned}$$

The following typing rules will be used:

$$\begin{aligned} &\mathsf{exl} \in \ a \leftarrow a \times b \\ &\mathsf{exr} \in \ b \leftarrow a \times b \end{aligned}$$

Functor composition. If F is a k-ary functor, and $G_0, \ldots, G_{k-1}$ are all n-ary functors, their composition $F^{\triangle}\ G_0\ \cdots\ G_{k-1}$ is an n-ary functor that maps an n-tuple z to $F\ (G_0\ z)\ \cdots\ (G_{k-1}\ z)$. Instead of $+^{\triangle}\ F\ G$ we write $F + G$, and likewise for $\times$.

From k-ary F we can make a unary functor $F^{\star}$ by defining $F^{\star} \ = \ F^{\triangle}\ \mathsf{Id}\ \cdots\ \mathsf{Id}$. So $F^{\star}z \ = \ F\ z\ \cdots\ z$, with k "z"s. When F is unary, $F^{\star} \ = \ F$. Furthermore we have a distribution property:

$$(F^{\triangle}\ G_0\ \cdots\ G_{k-1})^{\star} \ = \ F^{\triangle}\ {G_0}^{\star}\ \cdots\ {G_{k-1}}^{\star}$$

In the expression $(a^{\mathsf{K}})^{\star}$ the value of k is not determined, but since it is immaterial to the result this shouldn't be a problem.

5 Catamorphisms

We first look at a simple inductively defined datatype, that of the Peano naturals:

$$\mathbf{data}\ \ Nat \ = \ \mathsf{succ}\ Nat \ \mid \ \mathsf{zero}$$

There is only one number zero, which we can make explicit by:

$$\mathbf{data}\ \ Nat \ = \ \mathsf{succ}\ Nat \ \mid \ \mathsf{zero}\ 1$$

Instead of fancy constructor function names like succ and zero we now employ boring standard ones:

$$\mathbf{data}\ \ Nat \ = \ \mathsf{inl}\ Nat \ \mid \ \mathsf{inr}\ 1$$

The choice here is that afforded by sum, so we obtain, finally,

$$\mathbf{data}\ \ Nat \ = \ \mathsf{in}(Nat \ + \ 1)$$

in which there is one explicit constructor function left.

Now define the unary functor N by

$$N\ z \ = \ z + 1$$

Using the notations introduced earlier, this functor can also be expressed as $N = \mathsf{Id} + 1^{\mathsf{K}}$. The functor N captures the pattern of the inductive formation of the Peano naturals. The point is that we can use this to rewrite the definition of Nat to

$$\mathbf{data}\ \ Nat \ = \ \mathsf{in}(N\ Nat)$$

Apparently, the pattern functor N uniquely determines the datatype Nat. A functor built only from constants, extractions, sums, products and composition is called a *polynomial* functor. Whenever F is a unary polynomial functor, a definition of the form $\mathbf{data}\ \ Z \ = \ \mathsf{in}(F\,Z)$ uniquely determines Z. We need a notation to denote the datatype Z that is obtained, and write $Z \ = \ \mu F$. So $Nat \ = \ \mu N$. Replacing Z by μF in the datatype definition, and adding a subscript to the single constructor function in in order to disambiguate it, we obtain:

$$\mathbf{data}\ \ \mu F \ = \ \mathsf{in}_F(F\ \mu F)$$

Now in_F is a polytypic function, with typing

$$\mathsf{in}_F \in \mu F \leftarrow F\ \mu F$$

Each datatype μF has its cata combinator, which we denote with Malcolm's banana brackets:

$$(\![f]\!)_F \in a \leftarrow \mu F \quad \Leftarrow \quad f \in a \leftarrow F\ a$$

It is defined by:

$$\begin{aligned} (\![f]\!)_F \ &= \ h\ \mathbf{where} \\ &\quad\ h\,(\mathsf{in}_F\ xs) \ = \ f\,((F\ h)\ xs) \end{aligned}$$

In words, when catamorphism $(\![f]\!)_F$ is applied to a structure of type μF, this means it is applied recursively to the components of the structure, and the results are combined by applying its "body" f. The importance of catamorphisms is that they embody a closed expression for a familiar inductive definition technique ("canned induction") and thereby allow the polytypic expression of important program calculation rules, among which this fusion law (Malcolm):

$$h \cdot (\![f]\!)_F \ = \ (\![g]\!)_F \quad \Leftarrow \quad h \cdot f \ = \ g \cdot F\ h$$

6 Type functors

Playing the same game on the definition of $List$ gives us:

$$\mathbf{data}\ \ List\ a \ = \ \mathsf{in}((a \times List\ a)\ +\ 1)$$

Replacing the datatype being defined, $List\ a$, systematically by z, we obtain the "equation"

$$\mathbf{data}\ \ z \ = \ \mathsf{in}((a \times z)\ +\ 1)$$

Thus, we see that the pattern functor here is $(z :: (a \times z) + 1)$. It has a parameter a, which we make explicit by putting

$$L\ a = (z :: (a \times z) + 1)$$

Abstracting from a and z, we can write: $L = (\times) + 1^{\mathsf{K}}$. Now $List\ a = \mu(L\ a)$, or, abstracting from a:

$$List = (a :: \mu(L\ a))$$

In general, a parametrised functor $F\,a$ gives rise to a new functor, like here $List$. Such functors are called *type functors*. We introduce a notation:

$$\tau F = (a :: \mu(F\ a))$$

so $List = \tau L$, with L as above. The parameter a may actually be an n-tuple if functor F is $(n+1)$-ary, and then τF is an n-ary functor. The "map" part of a unary type functor can be expressed as a cata:

$$\tau F\ f = (\![\mathsf{in}_{F\,a} \cdot F\ f\ \mathsf{id}]\!)_{F\,b} \ \text{ for } f \in a \leftarrow b$$

Repeating this game for $Rose$, we find for its pattern functor $R\ a\ z = a \times List\ z$, or $R = \mathsf{Exl} \times List^{\triangle}\ \mathsf{Exr}$. This is not a polynomial functor, because of the appearance of the type functor $List$. Yet τR is well defined. Incorporating type functors into the ways of constructing functors extends the class of polynomial functors to the class of regular functors.

7 Regular functors

The definition of Fokkinga [10] will be followed, with one minor modification. A functor built only from constants, extractions, sums, products, composition and τ is called a *regular* functor. A formal grammar for the n-ary regular functors is:

$\mathrm{F}^{(n)}$ ::=	t^{K}	n-ary constant functor, for each type t
$\mid$	Ex_i^n	n-ary extraction, $i = 0, \ldots, n-1$
$\mid$	$+ \mid \times$ (only if $n = 2$)	binary sum and product functor
$\mid$	$\mathrm{F}^{(k)\triangle}\ \mathrm{F}_0^{(n)} \cdots \mathrm{F}_{k-1}^{(n)}$	functor composition
$\mid$	$\tau\mathrm{F}^{(n+1)}$	the type functor induced by $\mathrm{F}^{(n+1)}$

The minor modification, now, is that in the constant functors we do not allow any type t, but consider only the constant functor 1^{K}. This has a technical background that we cannot go into for space limitations.

Here is how the functor $Rose$ is produced by this grammar:

$$Rose = \tau(\times^{\triangle}\ \mathsf{Ex}_0^2\ ((\tau(+^{\triangle}\ (\times)\ 1^{\mathsf{K}}))^{\triangle}\ \mathsf{Ex}_1^2))$$

Daunting as this may look, it was obtained by purely mechanical unfolding of earlier definitions. The embedded τ corresponds to the type functor $List$.

8 Polytypic crush

The key to polytypic type definitions (given the present state of the art — no *Polyps From Outer Space* yet but see Freyd [12], Meijer and Hutton [32], Sheard and Fegaras [35] and Fegaras and Sheard [6] for possible extensions) is the formal grammar for regular functions. The class of regular functors is itself like (and can be modelled by) an inductive datatype, and so polytypic functions can be defined by induction on the formation of a regular functor.

Let us see how we can define a polytypic crush combinator that, applied to a suitable "body", results in a function $r[F]$ with typing $a \leftarrow F^\star a$ for all regular F. We write $r[F]$ here rather than r_F because, in this definition, F is the main parameter. In the process we shall see what ingredients are needed for its "body". We shall make a concerted effort to minimise the number of ingredients that need to be supplied to the combinator, and — also to stay as polymorphic as possible — we let ourselves be guided by typing considerations to take whatever will do when available "for free".

So we consider all cases corresponding to the production rules of the grammar. The inductive hypothesis is that we already have

$$r[F] \in a \leftarrow F^\star a$$

for sufficiently simple F. For the case τF we assume, for the sake of simplicity, that F is binary. We postpone the case 1^{K} to the last.

Case Ex_i^n: the requirement is $r[\mathsf{Ex}_i^n] \in a \leftarrow a$.
(Recall that $\mathsf{Ex}_i^{n\star} a = \mathsf{Ex}_i^n \; a \cdots a = a$). The choice is obvious: $r[\mathsf{Ex}_i^n] = \mathsf{id}$. So this need not be supplied.

Case $+$: the requirement is $r[+] \in a \leftarrow a + a$.
Here there is one (and only one) polymorphic function that will do, namely $\mathsf{id} \triangledown \mathsf{id}$.

Case $\times$: the requirement is $r[\times] \in a \leftarrow a \times a$.
There are polymorphic possibilities, namely exl and exr, but fixing any choice from these here would constitute an unacceptable discrimination against either the Left or the Right. So some ingredient $\oplus \in a \leftarrow a \times a$ will have to be supplied.

Case $F^\triangle \; G_0 \cdots G_{k-1}$: the requirement is

$$r[F^\triangle \; G_0 \cdots G_{k-1}] \in a \leftarrow F \; (G_0{}^\star \, a) \cdots (G_{k-1}{}^\star \, a) \, .$$

(The typing uses $(F^\triangle \; G_0 \cdots G_{k-1})^\star = F^\triangle \; G_0{}^\star \cdots G_{k-1}{}^\star$.) By the inductive hypothesis we have

$$r[F] \in a \leftarrow F^\star a$$

as well as $r[G_i] \in a \leftarrow G_i{}^\star \, a$, so that, using the typing of functors,

$$F \; r[G_0] \cdots r[G_{k-1}] \in F^\star a \leftarrow F \; (G_0{}^\star \, a) \cdots (G_{k-1}{}^\star \, a)$$

By composing these two we obtain for free

$$r[F] \cdot F\ r[G_0] \cdots r[G_{k-1}]$$

as having the required typing.

Case τF: the requirement is $r[\tau F] \in a \leftarrow \tau F\ a$.
Using $\tau F\ a = \mu(F\ a)$, and pattern matching against

$$(\![f]\!)_G \in a \leftarrow \mu G \quad \Leftarrow \quad f \in a \leftarrow G\,a$$

(replace here G by $F\ a$) we see that we can use a catamorphism

$$(\![f]\!)_{F\,a} \in a \leftarrow \mu(F\ a)$$

which has the required typing if

$$f \in a \leftarrow F^{\star} a$$

The latter requirement is solved by $f = r[F]$. The free solution is therefore $r[\tau F] = (\![r[F]]\!)_{F\,a}$.

Case 1^{K}: the requirement is $r[1^{\mathsf{K}}] \in a \leftarrow 1$.
We need some value of type a. We solve this by imposing the requirement on the ingredient $\oplus$ (needed for the case $\times$) that it have a neutral element $\nu_{\oplus}$, and take that.
□

So, in summary, we only need to supply one ingredient: a binary operation $\oplus \in a \leftarrow a \times a$ that has a neutral element. We introduce the notation

$$\langle\!\langle \oplus \rangle\!\rangle_F \in a \leftarrow F^{\star} a$$

for this polytypic crush.

More flexibility. We make our crush more flexible by allowing an optional second parameter $f \in a \leftarrow b$ and defining

$$\begin{aligned} \langle\!\langle \oplus, f \rangle\!\rangle_F &\in a \leftarrow F^{\star} b \\ \langle\!\langle \oplus, f \rangle\!\rangle_F &= \langle\!\langle \oplus \rangle\!\rangle_F \cdot F^{\star} f \end{aligned}$$

which generalises the one-parameter form since $\langle\!\langle \oplus \rangle\!\rangle = \langle\!\langle \oplus, \mathsf{id} \rangle\!\rangle$.

We also define a variant crush, actually just a useful abbreviation, designed for duty under bad weather conditions. What if $\oplus$ has no neutral element, like, for example, the operation $\downarrow$ selecting the lesser of two naturals? This was dealt with in classic BMF by introducing so-called "fictitious values". Here is a precise way of handling this. Given $\oplus \in a \leftarrow a \times a$ we construct a new operator $\oplus^{\mathsf{M}} \in \mathit{Maybe}\ a \leftarrow \mathit{Maybe}\ a \times \mathit{Maybe}\ a$ which behaves like $\oplus$ on the range of one, preserves associativity and symmetry, if any, also on the extended domain and has none as a neutral element:

$$
\begin{array}{rcl}
\mathsf{one}\ u \oplus^{\mathsf{M}} \mathsf{one}\ v & = & \mathsf{one}(u \oplus v) \\
\mathsf{one}\ u \oplus^{\mathsf{M}} \mathsf{none} & = & \mathsf{one}\ u \\
\mathsf{none}\ \oplus^{\mathsf{M}} \mathsf{one}\ v & = & \mathsf{one}\ v \\
\mathsf{none}\ \oplus^{\mathsf{M}} \mathsf{none} & = & \mathsf{none}
\end{array}
$$

We use this now to define the variant. To distinguish it from the normal one we prepend a superscript M. With $\oplus$ and f typed as before,

$$
\begin{array}{l}
{}^{\mathsf{M}}\langle\!\langle \oplus, f \rangle\!\rangle_F \in \mathit{Maybe}\ a \leftarrow F^{\star} b \\
{}^{\mathsf{M}}\langle\!\langle \oplus, f \rangle\!\rangle = \langle\!\langle \oplus^{\mathsf{M}}, \mathsf{one} \cdot f \rangle\!\rangle
\end{array}
$$

As for the normal crush we may omit the f-parameter when it is id.

9 Crush compared to cata

So isn't this crush a cata? No, it is not. For one thing, we saw that every type functor can be written as a catamorphism. Simple typing considerations show that in general type functors can not be expressed in the form of a crush. In that sense the crush combinator is less general. It is more general in the polytypic sense that crushes apply to source type $F^{\star}a$ for any functor F, while catamorphisms are only defined on source types of the form μG. (However, if $G = F\ a$, then μG is $\tau F\ a$, and the crush for τF is indeed a catamorphism.)

An interesting connection to classic BMF is

$$
\langle\!\langle \oplus, f \rangle\!\rangle_{List} = \oplus/ \cdot f*
$$

when $\oplus$ is the operator of a monoid. So we see that the catamorphism combinator $(\![\_]\!)$ introduced by Malcolm [24] [25] [26] and the present $\langle\!\langle \_ \rangle\!\rangle$ are different, incomparable, generalisations of Classic Cata™.

The most telling difference is the following. While $(\![\_]\!)$ *itself* is a polytypic combinator, its *application* to a body does in general not result in a polytypic function. In contrast, the application of $\langle\!\langle \_ \rangle\!\rangle$ always gives a polytypic function.

10 Some examples of polytypic crush

Function sum from Section 2 can be defined polytypically as a crush:

$$
\mathsf{sum} = \langle\!\langle + \rangle\!\rangle
$$

in which "+" is addition on numbers. Using the flexibility afforded by the optional parameter, we can modify this to define polytypic size, a function for counting the number of elements in a structure:

$$
\mathsf{size} = \langle\!\langle +, 1^{\mathsf{K}} \rangle\!\rangle
$$

Polytypic membership is obtained by

$$
e \in\ = \langle\!\langle \vee, \mathsf{eq}\ e \rangle\!\rangle
$$

Here is polytypic $\mathsf{flatten}$:

$$\begin{array}{lcl} \mathsf{flatten}_F & \in & \mathit{List}\ a \leftarrow F^\star a \\ \mathsf{flatten} & = & \langle\!\langle +\!\!+, [\_] \rangle\!\rangle \end{array}$$

Polytypic first returns the first element of its argument (first in in-order depth-first traversal). Since there may be no first element, we use the weatherproof variant:

$$\begin{array}{lcl} \mathsf{first}_F & \in & \mathit{Maybe}\ a \leftarrow F^\star a \\ \mathsf{first} & = & {}^{\mathsf{M}}\langle\!\langle \ll \rangle\!\rangle \ \ \mathbf{where}\ u \ll v \ = \ u \end{array}$$

In all these examples the crush has the form $\langle\!\langle \oplus, f \rangle\!\rangle$ in which $\oplus$ is associative. This is not a coincidence. Although not required for the well-definedness, the associativity of the operation is suggested by the fact that modelling n-tuples with pairs can be done from the left or from the right, corresponding to the isomorphy of types $(a \times b) \times c$ and $a \times (b \times c)$. Since the choice is arbitrary, it makes sense to require $\oplus$ to be associative.

Why, then, not *require* it to be associative? Well, here are some interesting applications with a non-associative operator.

Polytypic depth (or height, if you prefer), returns the depth of the deepest element, if any:

$$\mathsf{depth} \ = \ {}^{\mathsf{M}}\langle\!\langle \odot, 0^{\mathsf{K}} \rangle\!\rangle \ \ \mathbf{where}\ m \odot n \ = \ (m \uparrow n) + 1$$

Function binned returns a $\mathit{Maybe}^{\triangle}\mathit{Bin}$ value preserving the tree shape (if any) while converting type $F^\star a$ to $\mathit{Bin}\ a$:

$$\begin{array}{lcl} \mathsf{binned}_F & \in & \mathit{Maybe}(\mathit{Bin}\ a) \leftarrow F^\star a \\ \mathsf{binned} & = & {}^{\mathsf{M}}\langle\!\langle \mathsf{join}, \mathsf{tip} \rangle\!\rangle \end{array}$$

11 Calculating with polytypic functions

Polytypic crush captures one particular — although rather common — pattern of polytypic definition. For instantiations to specific datatypes, the calculation rules are well known. For example, if $h \ = \ \langle\!\langle \oplus, f \rangle\!\rangle_{\mathit{Bin}}$,

$$\begin{array}{lcl} h \cdot \mathsf{join} & = & \oplus \cdot h \times h \\ h \cdot \mathsf{tip} & = & f \end{array}$$

But we can go further. Not only can "canned" polytypy be put to good use to save a lot of work in writing polytypic programs, it can also be used to "calculate polytypically", giving identities that are polytypically valid.

As an illustration, we give, without proof, a polytypic fusion law for crushes, analogous to the fusion law for catamorphisms.

Crush fusion. *If the following three equations are satisfied:*

$$\begin{aligned} h \cdot \oplus &= \otimes \cdot h \times h \\ h\ \ \nu_{\oplus} &= \nu_{\otimes} \\ h \cdot f &= g \end{aligned}$$

then

$$h \cdot \langle\!\langle \oplus, f \rangle\!\rangle \ = \ \langle\!\langle \otimes, g \rangle\!\rangle$$

□

This is basically the "free theorem" (Wadler [37]) for polytypic crush, but a bit of fudging with the type is needed to handle the neutral elements. Jeuring and Jansson [22] show how to derive these for polytypic functions in general.

We can use this fusion law to find a condition under which

$$\langle\!\langle \oplus, f \rangle\!\rangle_{List} \cdot \mathsf{flatten} \ = \ \langle\!\langle \oplus, f \rangle\!\rangle$$

Using $\mathsf{flatten} \ = \ \langle\!\langle +\!\!\!+, [\_] \rangle\!\rangle$ and putting $h \ = \ \langle\!\langle \oplus, f \rangle\!\rangle_{List}$, crush fusion gives the conditions:

$$\begin{aligned} h \cdot +\!\!\!+ &= \oplus \cdot h \times h \\ h\ \ [\,] &= \nu_{\oplus} \\ h \cdot [\_] &= f \end{aligned}$$

From the theory of lists [2] we know that these are satisfied when $h \ = \ \oplus / \cdot f*$, that is, when $\oplus$ is associative. This shows that for associative $\oplus$ the crush $\langle\!\langle \oplus \rangle\!\rangle$ disregards any tree structure of the argument; it might as well have been a linear list.

For bad weather we have:

Corollary. *If the following two equations are satisfied:*

$$\begin{aligned} h \cdot \oplus &= \otimes \cdot h \times h \\ h \cdot f &= g \end{aligned}$$

then

$$\mathit{Maybe}\ h \ \cdot \ {}^{\mathsf{M}}\langle\!\langle \oplus, f \rangle\!\rangle \ = \ {}^{\mathsf{M}}\langle\!\langle \otimes, g \rangle\!\rangle$$

□

An application is:

$$\mathit{Maybe}\ \langle\!\langle \oplus, f \rangle\!\rangle_{Bin} \ \cdot \ \mathsf{binned}_F \ = \ {}^{\mathsf{M}}\langle\!\langle \oplus, f \rangle\!\rangle_F$$

12 Some futuristic remarks

Suppose we need a function to swap two naturals, with the typing $\mathsf{swap} \in Nat \times Nat \leftarrow Nat \times Nat$. That is not a hard task, but somehow it is in the nature of programming that it consists of easy tasks, only there are so many of them. The hard thing is to combine all the easy solutions to the little easy tasks in the right way, and anything helpful in that is helpful in programming. A good typing discipline is helpful. No decent functional programmer would define swap specialised to the naturals, but instead use a polymorphic function

$$\mathsf{swap} \in a \times b \leftarrow b \times a$$

In fact, giving this typing, you just can't get it wrong or else the type checking will tell you.

Similarly, even when — for all we know — a function may be needed for only one specific datatype, it may be helpful to define it polytypically. The possibilities to get it wrong but type correct are, if not crushed, then at least definitely reduced. Hindley-Milner style type inference for polytypic functions is described by Jansson and Jeuring [20]. Also, the polytypic version may be genuinely simpler. Just compare the polytypic definitions of $e\in$ and sum with the versions specialised for *Rose* from Section 2.

I started the Introduction with a question. Finding a new algorithm may be exciting, but coding yet another specialisation of a generic algorithm is not. Polytypy may prove to be the key to the level of flexibility needed to achieve interoperability by structural (as opposed to *ad hoc*) techniques. To facilitate polymorphic definition, we need elementary polytypic building blocks. Backhouse, Doornbos and Hoogendijk define, in a relational setting, a doubly polytypic and polymorphic zip. Jeuring [21] and Jeuring and Jansson [22] give many examples of further building blocks. More research is needed on "canned" polytypy, obviating the need of explicit induction on the formation of a regular functor. The crush combinator defined above is just a start.

References

1. Roland Backhouse, Henk Doornbos and Paul Hoogendijk. A Class of Commuting Relators. Unpublished, Eindhoven University of Technology, 1992. WWW `ftp://ftp.win.tue.nl/pub/math.prog.construction/zip.dvi.Z` .
2. Richard S. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, volume F36 of *NATO ASI Series*, pages 5–42. Springer–Verlag, 1987.
3. Richard Bird and Oege de Moor. *Algebra of Programming*. To appear, Prentice Hall, 1996.
4. Richard Bird, Oege de Moor and Paul Hoogendijk. Generic functional programming with types and relations. *J. of Functional Programming*, 6(1):1–28, 1996.
5. Oege de Moor. A Generic Program for Sequential Decision Processes. In Manuel Hermenegildo and S. Doaitse Swierstra, editors, *PLILP'95: Programming Languages: Implementations, Logics and Programs*, volume 982 of *LNCS*, pages 1–23. Springer Verlag, 1995.

6. Leonidas Fegaras and Tim Sheard. Revisiting catamorphisms over datatypes with embedded functions. In *Proceedings Principles of Programming Languages, POPL '96*, 1996.
7. Maarten M. Fokkinga. *Law and Order in Algorithmics.* PhD thesis, University of Twente, Dept INF, Enschede, The Netherlands, 1992.
8. Maarten M. Fokkinga. Calculate categorically! *Formal Aspects of Computing*, 4(4):673–692, 1992.
9. Maarten M. Fokkinga. A gentle introduction to category theory — the calculational approach. In *Lecture Notes of the STOP 1992 Summerschool on Constructive Algorithmics*, pages 1–72 of Part 1. Utrecht University, 1992.
10. Maarten M. Fokkinga. Monadic maps and folds for arbitrary datatypes. Memoranda Informatica 94-28, University of Twente, 1994.
11. Maarten M. Fokkinga. Datatype laws without signatures. *Mathematical Structures in Computer Science*, 6:1–32, 1996.
12. Peter Freyd. Recursive types reduced to inductive types. In *Proceedings Logic in Computer Science, LICS '90*, pages 498–507, 1990.
13. Tatsuya Hagino. *Category Theoretic Approach to Data Types.* PhD thesis, University of Edinburgh, 1987.
14. Paul F. Hoogendijk. Generators, Destructors and Natural Transformations. Unpublished, Eindhoven University of Technology, 1993. WWW `ftp://ftp.win.tue.nl/pub/math.prog.construction/gendes.dvi.Z` .
15. John Hughes. The Design of a Pretty-printing Library. In Johan Jeuring and Erik Meijer, editors, *Advanced Functional Programming*, LNCS 925, pages 53–96. Springer Verlag, 1995.
16. Marieke Huisman. *The Calculation of a Polytypic Parser.* Master's thesis, Utrecht University, Dept. of Computing Science, 1996.
17. Patrik Jansson. *Polytypism and Polytypic Unification.* Master's thesis, Chalmers University of Technology and University of Göteborg, 1995. WWW `file://ftp.cs.chalmers.se/pub/users/patrikj/papers/masters/thesis.ps.Z` .
18. Patrik Jansson and Johan Jeuring. Polyp — a polytypic programming language. Submitted for publication, 1996. WWW `http://www.cs.chalmers.se/~johanj/polytypism/polyp.ps` .
19. Patrik Jansson and Johan Jeuring. Polytypic unification — implementing polytypic functions with constructor classes. Submitted for publication, 1996. WWW `http://www.cs.chalmers.se/ johanj/polytypism/unify.ps` .
20. Patrik Jansson and Johan Jeuring. Type inference for polytypic functions. In preparation, 1996.
21. Johan Jeuring. Polytypic pattern matching. In S. Peyton Jones, editor, *Conference Record of FPCA '95, SIGPLAN-SIGARCH-WG2.8 Conference on Functional Programming Languages and Computer Architecture*, pages 238–248, 1995. WWW `http://www.cs.chalmers.se/~johanj/ppm.dvi` .
22. Johan Jeuring and Patrik Jansson. Polytypic programming. To appear in *Proceedings of the Second International Summer School on Advanced Functional Programming Techniques*, LNCS, Springer Verlag, 1996. WWW `http://www.cs.chalmers.se/~johanj/polytypism/notes.ps` .
23. Mark P. Jones. Functional Programming with Overloading and Higher-Order Polymorphism. In Johan Jeuring and Erik Meijer, editors, *Advanced Functional Programming*, LNCS 925, pages 97–136. Springer Verlag, 1995.
24. Grant Malcolm. Homomorphisms and promotability. In J.L.A. van de Snepscheut, editor, *Mathematics of Program Construction*, LNCS 375, pages 335–347. Springer

Verlag, 1989.
25. Grant Malcolm. *Algebraic Data Types and Program Transformation.* PhD thesis, University of Groningen, 1990.
26. Grant Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14(2–3):255–280, 1990.
27. Ernest G. Manes and Michael A. Arbib. Algebraic Approaches to Program Semantics. Text and Monographs in Computer Science. Springer Verlag, 1986.
28. Lambert Meertens. Algorithmics—towards programming as a mathematical activity. In J.W. de Bakker, M. Hazewinkel, and J.K. Lenstra, editors, *Proceedings of the CWI Symposium on Mathematics and Computer Science*, volume 1 of *CWI Monographs*, pages 289–334. North–Holland, 1986.
29. Lambert Meertens. Paramorphisms. *Formal Aspects of Computing*, 4(5):413–425, 1992.
30. Lambert Meertens. Category Theory for Program Construction by Calculation. *Lecture Notes for ESSLLI'95*, 1995. WWW `http://www.cwi.nl/~lambert/e95.ps.Z` .
31. Erik Meijer, Maarten M. Fokkinga and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *FPCA91: Functional Programming Languages and Computer Architecture*, LNCS 523, pages 124–144. Springer Verlag, 1991.
32. Erik Meijer and Graham Hutton. Bananas in space: extending fold and unfold to exponential types. In S. Peyton Jones, editor, *Conference Record of FPCA '95, SIGPLAN-SIGARCH-WG2.8 Conference on Functional Programming Languages and Computer Architecture*, pages 324–333, 1995.
33. Fritz Ruehr. *Analytical and Structural Polymorphism Expressed Using Patterns Over Types.* PhD thesis, University of Michigan, 1992.
34. Tim Sheard. Type parametric programming with compile-time reflection. Oregon Graduate Institute of Science and Technology, 1993.
35. Tim Sheard and Leonidas Fegaras. A Fold for All Seasons. In *FPCA'93, Conference on Functional Programming Languages and Computer Architecture*, pages 233–242. ACM Press, 1993. WWW `ftp://cse.ogi.edu/pub/crml/fpca93.ps.Z` .
36. Daniël Tuijnman. *A Categorical Approach to Functional Programming.* PhD thesis, Universität Ulm, Fakultät für Informatik, Abteilung Programmiermethodik und Compilerbau, 1995.
37. Phil Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture, FPCA '89*, pages 347–359. ACM Press, 1989.