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 polytyp-
ically 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 com-
binators (higher-order functions) like map and foldr capture very general pro-
gramming 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 [21] [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 interop-
erable 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:\footnote{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 \( f \in a \leftrightarrow b \) 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.}

\[
\begin{align*}
\textbf{data} \hspace{1em} & \text{List } a = \text{cons } a \hspace{1em} (\text{List } a) \hspace{1em} \mid \hspace{1em} \text{nil} \\
\textbf{data} \hspace{1em} & \text{Maybe } a = \text{one } a \hspace{1em} \mid \hspace{1em} \text{none} \\
\textbf{data} \hspace{1em} & \text{Bin } a = \text{join } (\text{Bin } a) \hspace{1em} (\text{Bin } a) \hspace{1em} \mid \hspace{1em} \text{tip } a \\
\textbf{data} \hspace{1em} & \text{Rose } a = \text{fork } a \hspace{1em} (\text{List } (\text{Rose } a))
\end{align*}
\]

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

\[
\begin{align*}
\textbf{map}_{\text{List}} & \in (\text{List } a \leftrightarrow \text{List } b) \leftrightarrow (a \leftrightarrow b) \\
\textbf{map}_{\text{Maybe}} & \in (\text{Maybe } a \leftrightarrow \text{Maybe } b) \leftrightarrow (a \leftrightarrow b) \\
\textbf{map}_{\text{Bin}} & \in (\text{Bin } a \leftrightarrow \text{Bin } b) \leftrightarrow (a \leftrightarrow b) \\
\textbf{map}_{\text{Rose}} & \in (\text{Rose } a \leftrightarrow \text{Rose } b) \leftrightarrow (a \leftrightarrow b)
\end{align*}
\]

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

\[
\begin{align*}
e & \in \text{List} \hspace{1em} \text{cons } u \hspace{1em} x & = & \text{eq } c \hspace{1em} u \hspace{1em} \lor \hspace{1em} e \in \text{List } x \\
e & \in \text{List} \hspace{1em} \text{nil} & = & \text{false} \\
e & \in \text{Maybe} \hspace{1em} \text{one } u & = & \text{eq } c \hspace{1em} u \\
e & \in \text{Maybe} \hspace{1em} \text{none} & = & \text{false} \\
e & \in \text{Bin} \hspace{1em} \text{join } x \hspace{1em} y & = & e \in \text{Bin } x \hspace{1em} \lor \hspace{1em} e \in \text{Bin } y \\
e & \in \text{Bin} \hspace{1em} \text{tip } u & = & \text{eq } c \hspace{1em} u \\
e & \in \text{Rose} \hspace{1em} \text{fork } u \hspace{1em} xs & = & \text{eq } c \hspace{1em} u \hspace{1em} \lor \hspace{1em} \text{any } (e \in \text{Rose } x) \hspace{1em} xs
\end{align*}
\]

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

\[
\begin{align*}
\text{sum}_{\text{List}} \hspace{1em} (\text{cons } u \hspace{1em} x) & = u \hspace{1em} + \hspace{1em} \text{sum}_{\text{List}} x \\
\text{sum}_{\text{List}} \hspace{1em} \text{nil} & = 0 \\
\text{sum}_{\text{Maybe}} \hspace{1em} \text{one } u & = u \\
\text{sum}_{\text{Maybe}} \hspace{1em} \text{none} & = 0
\end{align*}
\]
\[
\begin{align*}
\text{sum}_\text{Bin} \ (\text{join } x \ y) &= \text{sum}_\text{Bin} \ x + \text{sum}_\text{Bin} \ y \\
\text{sum}_\text{Bin} \ (\text{up } u) &= u \\
\text{sum}_\text{Rose} \ (\text{fork } u \ xs) &= u + \text{sum}_\text{List} \ (\text{map}_\text{List} \ \text{sum}_\text{Rose} \ xs)
\end{align*}
\]

Polytypy, now, allows us to replace all these definitions by a single definition for \(\text{map}_F\), a single definition for \(\varepsilon_F\) and a single definition for \(\text{sum}_F\), each of which can be specialised to any of the above datatype constructors and many more by taking \(F\) to be \(\text{List}\), \(\text{Maybe}\), \(\text{Bin}\), and so on.

Polytypy is orthogonal to polymorphism. The polytypic function \(\text{map}_F\) is truly polymorphic — that is, each of its instantiations is. The polytypic functions \(\varepsilon_F\) and \(\text{sum}_F\) are as polymorphic as \(\text{eq}\) and \(+\) are, which is, respectively, somewhat and hardly. However, \(\text{eq}\) 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:

\[
\begin{align*}
\oplus/\cdot f &= h \ \text{where} \\
h\ (x \# y) &= h \ x \oplus h \ y \\
h\ [u] &= f \ u \\
h\ [\ ] &= \nu_\oplus
\end{align*}
\]

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 unparser — rather like polytypic 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 unparser · 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 $e$, $e^a$ 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:

```
data 1 = blob
```

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

$$
F \ (f_0 \cdot g_0) \cdot \ldots \cdot (f_{n-1} \cdot g_{n-1}) = F f_0 \cdot \ldots \cdot F f_{n-1} \cdot F g_0 \cdot \ldots \cdot F g_{n-1}
$$

provided that $f_i \in a_i \leftarrow b_i$ and $g_i \in b_i \leftarrow c_i$

$$
F \ id \cdot \ldots \cdot id = id
$$

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

An example are the functions $mapF$, since they satisfy the functional identities $mapF(f \cdot g) = mapFf \cdot mapFg$ and $mapF id = id$. So they are unary functors. As is easily verified, $id_a^a$ is also a functor. It is $n$-ary for all $n$. Further, each extraction combinator

\footnote{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].}
\[ \text{Ex}_i^n f_0 \cdots f_{n-1} = f_i, \ i = 0, \ldots, n-1 \]

is an \( n \)-ary functor. We write \( \text{Id} \) for the unary functor \( \text{Ex}_0^1 \), and \( \text{Ex} \) and \( \text{Ex}_r \) for the binary functors \( \text{Ex}_0^2 \) and \( \text{Ex}_1^2 \).

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

\[ F a_0 \cdots a_{n-1} = A \]
\[ F b_0 \cdots b_{n-1} = B \]

So for unary functor \( F \) we have

\[ F f \in F a \leftarrow F b \iff f \in a \leftarrow b \]

Looking at the typing of \( \text{map}_F \):

\[ \text{map}_F f \in F a \leftarrow F b \iff f \in a \leftarrow b \]

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 \( \text{List} f \) rather than \( \text{map}_{\text{List}} f \). Likewise, we write \( a^k \) instead of \( \text{id}^k_a \).

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:

\[
\text{data} \ a + b = \text{inl} \ a \mid \text{inr} \ b
\]

\[
f + g = h \quad \text{where}
\]
\[
h(\text{inl} \ u) = \text{inl}(f \ u) \]
\[
h(\text{inr} \ v) = \text{inl}(g \ v)
\]

\[
f \circ g = h \quad \text{where}
\]
\[
h(\text{inl} \ u) = f \ u \]
\[
h(\text{inr} \ v) = g \ v
\]

The following typing rule will be used:

\[
f \circ g \in c \leftarrow a + b \iff f \in c \leftarrow a \land g \in c \leftarrow b
\]
The product functor. The binary product functor \( \times \) is given by:

\[
\text{data } a \times b = \text{pair } a \ b
\]

\[
f \times g = h \quad \text{where}
\]

\[
h(\text{pair } u \ v) = \text{pair } (f \ u) \ (g \ v)
\]

\[
\text{exl}(\text{pair } u \ v) = u
\]

\[
\text{exr}(\text{pair } u \ v) = v
\]

The following typing rules will be used:

\[
\text{exl} \in a \leftarrow a \times b
\]

\[
\text{exr} \in b \leftarrow a \times b
\]

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

From \( k \)-ary \( F \) we can make a unary functor \( F^* \) by defining \( F^* = F^\Delta \ 1d \cdots 1d \). So \( F^* z = F z \cdots z \), with \( k \) “\( z \)”s. When \( F \) is unary, \( F^* = F \). Furthermore we have a distributive property:

\[
(F^\Delta G_0 \cdots G_{k-1})^* = F^\Delta G_0^* \cdots G_{k-1}^*
\]

In the expression \( (a^k)^* \) 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:

\[
\text{data } \text{Nat} = \text{succ } \text{Nat} \mid \text{zero}
\]

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

\[
\text{data } \text{Nat} = \text{succ } \text{Nat} \mid \text{zero } 1
\]

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

\[
\text{data } \text{Nat} = \text{inl } \text{Nat} \mid \text{inr } 1
\]

The choice here is that afforded by \text{sum}, so we obtain, finally,

\[
\text{data } \text{Nat} = \text{in}(\text{Nat } + 1)
\]

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
\[ \|N \| = \mathrm{id} + \|1\|. \] 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 \( \text{Nat} \) to

\[
\textbf{data} \; \text{Nat} = \in (\|N\| \text{Nat})
\]

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

\[
\textbf{data} \; \mu F = \in F (\mu F)
\]

Now \( \in F \) is a polytypic function, with typing

\[ \in F : \mu F \leftarrow F \mu F \]

Each datatype \( \mu F \) has its \textit{cata} combinator, which we denote with Malcolm’s banana brackets:

\[
(f)_F \in a \leftarrow \mu F \iff f \in a \leftarrow F a
\]

It is defined by:

\[
(f)_F = h \text{ where } h (\in_F xs) = f ((F h) xs)
\]

In words, when catamorphism \( (f)_F \) is applied to a structure of type \( \mu F \), 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):

\[
h \cdot (f)_F = (g)_F \iff h \cdot f = g \cdot F h
\]

6 Type functors

Playing the same game on the definition of \textit{List} gives us:

\[
\textbf{data} \; \text{List} a = \in ((a \times \text{List} a) + 1)
\]

Replacing the datatype being defined, \( \text{List} a \), systematically by \( z \), we obtain the “equation”

\[
\textbf{data} \; z = \in ((a \times z) + 1)
\]
Thus, we see that the pattern functor here is \( \tau \cdot (a \times z) + 1 \). It has a parameter \( a \), which we make explicit by putting
\[
L\ a = (z \cdot (a \times z) + 1)
\]
Abstracting from \( a \) and \( z \), we can write: \( L = (\times) + 1^k \). 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 \( \tau F \) is an \( n \)-ary functor. The “map” part of a unary type functor can be expressed as a cata:
\[
\tau F\ f = \text{cata}_{F\ a \cdot F\ f\ id} f\ for\ f \in \ a \leftarrow b
\]
Repeating this game for \( Rose \), we find for its pattern functor \( R\ a\ z = a \times List\ z \), or \( R = \text{Exl} \times \text{List}^\omega \text{Exl} \). This is not a polynomial functor, because of the appearance of the type functor \( List \). Yet \( \tau R \) 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 \( \tau \) is called a regular functor. A formal grammar for the \( n \)-ary regular functors is:
\[
F^{(n)} ::= \begin{align*}
t &::= t^k & \text{n-ary constant functor, for each type } t \\
\text{Ex}_{i}^t &::= \text{n-ary extraction, } i = 0, \ldots, n - 1 \\
+ &::= \text{binary sum} & \text{only if } n = 2 \\
\times &::= \text{binary product} \\
F^{(k)} &::= F^{(n)}_0 \ldots F^{(n)}_{k-1} & \text{functor composition} \\
\tau F^{(n+1)} &::= \text{the type functor induced by } F^{(n+1)}
\end{align*}
\]
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:
\[
Rose = \tau(\times \text{Ex}_{0}^{2}((\tau(\times ((\times) 1^k))^\omega \text{Ex}_{1}^{2})))
\]
Daunting as this may look, it was obtained by purely mechanical unfolding of earlier definitions. The embedded \( \tau \) 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 \( a \leftarrow F^*a \) 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

\[
r[F] \in a \leftarrow F^*a
\]

for sufficiently simple \( F \). For the case \( rF \) we assume, for the sake of simplicity, that \( F \) is binary. We postpone the case \( 1^x \) to the last.

Case \( \text{Ex}^n \): the requirement is \( r[\text{Ex}^n] \in a \leftarrow a \).

(Recall that \( \text{Ex}^n a = \text{Ex}^0 a \cdots a = a \)). The choice is obvious: \( r[\text{Ex}^n] = \text{id} \).

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 \( \text{id} \circ \text{id} \).

Case \( \times \): the requirement is \( r[\times] \in a \leftarrow a \times a \).

There are polymorphic possibilities, namely \( \text{exl} \) and \( \text{exr} \), 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 \( F^\Delta G_0 \cdots G_{k-1} \): the requirement is

\[
r[F^\Delta G_0 \cdots G_{k-1}] \in a \leftarrow F (G_0^* a) \cdots (G_{k-1}^* a)
\]

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

\[
r[F] \in a \leftarrow F^*a
\]

as well as \( r[G_i] \in a \leftarrow G_i^* a \), so that, using the typing of functors,

\[
F \ r[G_0] \cdots r[G_{k-1}] \in F^*a \leftarrow F (G_0^* a) \cdots (G_{k-1}^* a)
\]

By composing these two we obtain for free
\[ r[F] \cdot F \cdot r[G_0] \cdots r[G_{k-1}] \]
as having the required typing.

**Case \( \tau F \):** the requirement is \( r[\tau F] \in a \leftarrow \tau F \ a \).

Using \( \tau F \ a = \mu(F \ a) \), and pattern matching against
\[
(f)_{G} \in a \leftarrow \mu G \quad \iff \quad f \in a \leftarrow G \ a
\]
(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
\[
f \in a \leftarrow F^{*} a
\]
The latter requirement is solved by \( f = r[F] \). The free solution is therefore \( r[\tau F] = \{ r[F] \}_{F_a} \).

**Case \( 1^{*} \):** the requirement is \( r[1^{*}] \in a \leftarrow 1 \).

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.

\[ \square \]

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
\[
\llbracket \oplus \rrbracket_F \in a \leftarrow F^{*} a
\]
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
\[
\llbracket \oplus, f \rrbracket_F \in a \leftarrow F^{*} b
\]
\[
\llbracket \oplus, f \rrbracket_F = \llbracket \oplus \rrbracket_F \cdot F^{*} f
\]
which generalises the one-parameter form since \( \llbracket \oplus \rrbracket = \llbracket \oplus, \text{id} \rrbracket \).

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 \( \oplus^M \in \text{Maybe} \ a \leftarrow \text{Maybe} \ a \times \text{Maybe} \ a \) which behaves like \( \oplus \) on the range of \text{one}, preserves associativity and symmetry, if any, also on the extended domain and has \text{none} as a neutral element:
\[
\text{one } u \oplus^M \text{ one } v = \text{ one}(u \oplus v) \\
\text{one } u \oplus^M \text{ none } = \text{ one } u \\
\text{none } \oplus^M \text{ one } v = \text{ one } v \\
\text{none } \oplus^M \text{ none } = \text{ none}
\]

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,

\[
M(\oplus, f)_F \in \text{Maybe } a \leftarrow F^* b \\
M(\oplus, f)_F = \langle \langle \oplus^M, \text{ one } \cdot f \rangle \rangle
\]

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 \( F^* a \) for any functor \( F \), while catamorphisms are only defined on source types of the form \( \mu G \). (However, if \( G = F a \), then \( \mu G \) is \( \tau F a \), and the crush for \( \tau F \) is indeed a catamorphism.)

An interesting connection to classic BMF is

\[
\langle \langle \oplus, f \rangle \rangle_{\text{List}} = \oplus/ \cdot f^*
\]

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

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

10 Some examples of polytypic crush

Function \text{sum} from Section 2 can be defined polytypically as a crush:

\[
\text{sum} = \langle \langle \_ \rangle \rangle
\]

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

\[
\text{size} = \langle \langle +, 1^* \rangle \rangle
\]

Polytypic membership is obtained by

\[
c \in = \langle \langle V, \text{eq } c \rangle \rangle
Here is polytypic flatten:

\[
\text{flatten}_F \in \text{List } a \leftarrow F^*a \\
\text{flatten} = \langle +, [a] \rangle
\]

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:

\[
\text{first}_F \in \text{Maybe } a \leftarrow F^*a \\
\text{first} = M\langle \langle \leftarrow \rangle \rangle \text{ where } u \leftarrow v \leftarrow u
\]

In all these examples the crush has the form \(\langle \oplus, f \rangle\) 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:

\[
\text{depth} = M\langle \ominus, 0^\oplus \rangle \text{ where } m \ominus n = (m \uparrow n) + 1
\]

Function binned returns a \(\text{Maybe}^\uparrow \text{Bin}\) value preserving the tree shape (if any) while converting type \(F^*a\) to \(\text{Bin } a\):

\[
\text{binned}_F \in \text{Maybe}(\text{Bin } a) \leftarrow F^*a \\
\text{binned} = M\langle \text{join}, \text{tip} \rangle
\]

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 \(h = \langle [\oplus, f] \rangle_{\text{bin}}\):

\[
h \cdot \text{join} = \oplus \cdot h \times h \\
h \cdot \text{tip} = f
\]

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:

\[ h \cdot \oplus = \odot \cdot h \times h \]
\[ h \cdot \nu_{\oplus} = \nu_{\odot} \]
\[ h \cdot f = g \]

then

\[ h \cdot \langle \oplus, f \rangle = \langle \odot, g \rangle \]

\[ \square \]

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

\[ \langle \oplus, f \rangle_{\text{List}} \cdot \text{flatten} = \langle \oplus, f \rangle \]

Using \text{flatten} = \langle ::, [\cdot] \rangle and putting \( h = \langle \oplus, f \rangle_{\text{List}} \), crush fusion gives the conditions:

\[ h \cdot :: = \oplus \cdot h \times h \]
\[ h \cdot [\cdot] = \nu_{\oplus} \]
\[ h \cdot [\cdot] = f \]

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 \( \langle \oplus \rangle \) 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:

\[ h \cdot \oplus = \odot \cdot h \times h \]
\[ h \cdot f = g \]

then

\[ \text{Maybe } h \cdot M\langle \oplus, f \rangle = M\langle \odot, g \rangle \]

\[ \square \]

An application is:

\[ \text{Maybe } \langle \oplus, f \rangle_{\text{Bin}} \cdot \text{binned} \_P = M\langle \odot, f \rangle_{\_P} \]
12 Some futuristic remarks

Suppose we need a function to swap two naturals, with the typing \texttt{swap} \in \texttt{Nat} \times \texttt{Nat} \rightarrow \texttt{Nat} \times \texttt{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 \texttt{swap} specialised to the naturals, but instead use a polymorphic function

\[
\texttt{swap} \in a \times b \rightarrow b \times a
\]

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 \texttt{e\in} and \texttt{sum} with the versions specialised for \texttt{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 \texttt{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 \texttt{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

Verlag, 1989.