Functor Pulling

Lambert Meertens*
Department of Algorithmics and Architecture, CWI, Amsterdam, and
Department of Computing Science, Utrecht University, The Netherlands
www.cwi.nl/~lambert

1 Introduction

This paper is concerned with the construction of *basic building blocks* for polytypic programming. A polytypic program is a program that is generic in a type constructor, say F; by specializing it for $F{:=}List$, a program for lists is obtained, while for $F{:=}Tree$ we get a program for trees, and so on.

Well-known examples of such polytypic building blocks are the initial algebra for F (assuming F is such that an initial algebra exists):

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

and the *catamorphism* combinator:

$$(\!|f|\!)_F \ : \ a \leftarrow \mu F \quad \Leftarrow \quad f \ : \ a \leftarrow Fa$$

For more details, see [7] or [1].

A previous paper [8] defined a *crush* combinator

$$\langle\!\langle \oplus \rangle\!\rangle_F \ : \ a \leftarrow F\langle a, a, \ldots, a\rangle \quad \Leftarrow \quad \oplus \ : \ a \leftarrow a \times a$$

for all "regular" F, giving a generic way to reduce data structures with a binary operation.

In this paper we define a generalization of crush, and present some examples. Further polytypic basic building blocks, as well as several advanced applications, can be found in [5, 6, 4]

*Part of this work was performed while visiting Kestrel Institute, Palo Alto.

2 Preliminaries

Constant functions Given $x \in A$, the mapping $(z \mapsto x)$ always returning x is denoted as x^{K}. It can be polymorphically typed as $x^{\mathsf{K}} : A \leftarrow a$ with a type variable a. We have $x^{\mathsf{K}} \circ f = x^{\mathsf{K}}$ for any f.

Categorical embedding We assume a categorical embedding of the programming formalism, with the usual mappings:

function	$\rightarrow$	arrow
type	$\rightarrow$	object
type constructor	$\rightarrow$	functor
polymorphic function	$\rightarrow$	natural transformation

(the latter only if the typing translates to one with covariant functors.) As is usual in the functional-programming idiom, we treat polymorphic functions as multiply typable functions in the base category, and additionally we may use the typing from the functor category, so that the following — not equally general — typings are all correct:

$$
\begin{array}{ll}
\mathsf{concat} : \mathit{List}\ a & \leftarrow \mathit{List}(\mathit{List}\ a) \\
\mathsf{concat} : \mathit{List}\ \mathit{Int} & \leftarrow \mathit{List}(\mathit{List}\ \mathit{Int}) \\
\mathsf{concat} : \mathit{List} & \leftarrow \mathit{List}{\cdot}\mathit{List} \\
\mathsf{concat} : \mathit{List}{\cdot}F & \leftarrow \mathit{List}{\cdot}\mathit{List}{\cdot}F
\end{array}
$$

Function composition will be denoted with "$\circ$", whereas *functor* composition is denoted with "$\cdot$".

We assume that the category has finite sums and products and initial F-algebras for all regular F (defined in Section 3). For binary sum we use:

$$
\begin{array}{l}
\mathsf{inl} \ : \ a + b \leftarrow a \\
\mathsf{inr} \ : \ a + b \leftarrow b
\end{array}
$$

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

and for binary product:

$$
\begin{array}{l}
\mathsf{exl} \ : \ a \leftarrow a \times b \\
\mathsf{exr} \ : \ b \leftarrow a \times b
\end{array}
$$

$$
f \vartriangle g \ : \ a \times b \leftarrow c \quad \Leftarrow \quad f \ : \ a \leftarrow c \ \wedge \ g \ : \ b \leftarrow c
$$

For functors $F : \mathcal{C} \leftarrow \mathcal{E}$ and $G : \mathcal{D} \leftarrow \mathcal{E}$ the functor $F \Delta G : \mathcal{C} \times \mathcal{D} \leftarrow \mathcal{E}$ is defined by:

$$
(F \Delta G)a \ = \ \langle Fa, Ga \rangle
$$

Tuple notations To avoid the excessive use of '$\cdots$'s, we need some notation.

Indices always run through an initial segment of the naturals. Rather than writing, e.g., $\prod_{i=0}^{n-1}$, we use the terser form $\prod_i^n$. We may omit the superscript n when it can be inferred or reasonably guessed from the context.

We use the notation $\between_i^n a_i$ for the n-tuple $\langle a_0, a_1, \ldots, a_{n-1}\rangle$. This behaves like a functor: it respects typing and composition. So if $f_i : a_i \leftarrow b_i$ for all i in the range, we have

$$\between_i f_i : \between_i a_i \leftarrow \between_i b_i$$

and

$$\between_i (f_i \circ g_i) = \between_i f_i \circ \between_i g_i$$

If $a_i \in \mathcal{C}_i$, then $\between_i a_i \in \prod_i \mathcal{C}_i$. If the expression being tupled is constant, as in $\between_i a = \langle a, a, \ldots, a\rangle$, we may omit the running index altogether and write $\between a$.

The *projection functors* $\mathsf{Ex}_i^n : \mathcal{C}_i \leftarrow \prod_j \mathcal{C}_j$, for $i < n$, can now be defined by:

$$\mathsf{Ex}_i \between_j a_j = a_i$$

We use Δ with a running index for "tupling" (the results of) a sequence of functors with a common source, thereby generalizing binary Δ: for $F_i : \mathcal{C}_i \leftarrow \mathcal{D}$, their tupling $\Delta_i F_i : \prod_i C_i \leftarrow \mathcal{D}$ is defined pointwise by:

$$(\Delta_i F_i) a = \between_i (F_i a)$$

(so $\Delta_i^2 F_i = F_0 \Delta F_1$). For a tuple of functorially typed polymorphic functions $f_i : F_i \leftarrow G_i$ the correct typing is

$$\between_i f_i : \Delta_i F_i \leftarrow \Delta_i G_i$$

(This is the consequence of treating polymorphic functions as "sloppily" typed functions in the base category; with a proper categorical treatment of these as being natural transformations we should use the typing $\Delta_i f_i : \Delta_i F_i \leftarrow \Delta_i G_i$ in the functor category.)

Here are some rules involving Δ. The $\mathsf{Ex}\Delta$-rule is:

$$\mathsf{Ex}_i \cdot \Delta_j F_j = F_i$$

The Δ-fusion rules are:

$$\begin{aligned} \Delta_i (F \cdot G_i) &= F^* \cdot \Delta_i Gi \\ \Delta_i (F_i \cdot G\,) &= \Delta_i Fi \cdot G \end{aligned}$$

For $F : \mathcal{C} \leftarrow \mathcal{D}$, its n-fold product $F^n : \mathcal{C}^n \leftarrow \mathcal{D}^n$ is defined by:

$$F^n = \prod_i^n F$$

Here $\_^n$ is a functor on functors, so $(F \cdot G)^n = F^n \cdot G^n$. Note that $F^2 = F \times F$, not $F \cdot F$. We usually use an $*$ for the superscript when it should be clear from the context.

Transposition The *transposition functors* ${}_n\mathsf{T}_m : (\mathcal{C}^n)^m \leftarrow (\mathcal{C}^m)^n$ are defined by:

$${}_n\mathsf{T}_m = \Delta_i^m \, (\mathsf{Ex}_i^m)^n$$

So, spelling it out,

$$\begin{array}{llll} \langle\, \langle\; a_{0,0}, & a_{1,0}, & \ldots, & a_{m-1,0} \;\rangle \\ \quad\langle\; a_{0,1}, & a_{1,1}, & \ldots, & a_{m-1,1} \;\rangle \\ & \vdots & & \\ \quad\langle\; a_{0,n-1}, & a_{1,n-1}, & \ldots, & a_{m-1,n-1} \rangle\,\rangle \end{array} \xmapsto{{}_n\mathsf{T}_m} \begin{array}{llll} \langle\, \langle\; a_{0,0}, & a_{0,1}, & \cdots & a_{0,n-1} \;\rangle \\ \quad\langle\; a_{1,0}, & a_{1,1}, & \ldots, & a_{1,n-1} \;\rangle \\ & \vdots & & \\ \quad\langle\; a_{m-1,0}, & a_{m-1,1}, & \ldots, & a_{m-1,n-1} \rangle\,\rangle \end{array}$$

We have the $\mathsf{T}\Delta^*$-rule:

$$\mathsf{T}\cdot(\Delta_i\, G_i)^* \;=\; \Delta_i\, (G_i)^*$$

since

$$\begin{array}{ll} & \mathsf{T}\cdot(\Delta_i\, G_i)^* \\ = & \quad \{ \text{ definition of } \mathsf{T} \;\} \\ & \Delta_i\, (\mathsf{Ex}_i)^*\cdot(\Delta_i\, G_i)^* \\ = & \quad \{\; \Delta\text{-fusion} \;\} \\ & \Delta_i\, ((\mathsf{Ex}_i)^*\cdot(\Delta_i\, G_i)^*) \\ = & \quad \{\; \_^* \text{ is a functor } \} \\ & \Delta_i\, (\mathsf{Ex}_i\cdot\Delta_i\, G_i)^* \\ = & \quad \{\; \mathsf{Ex}\Delta\text{-rule} \;\} \\ & \Delta_i\, (G_i)^* \end{array}$$

Pairs of transpositions ${}_n\mathsf{T}_m$ and ${}_m\mathsf{T}_n$ form natural isomorphisms, so we also have:

$$\mathsf{T}\cdot\Delta_i(G_i)^* \;=\; (\Delta_i\, G_i)^*$$

3 Regular functors

Regular functors are an extension of the class of *polynomial* functors with type functors. We define the latter notion first.

Type functors Given binary functor $F : \mathcal{C} \leftarrow \mathcal{D} \times \mathcal{C}$, define, for fixed $a \in \mathcal{D}$, the unary functor $F_a : \mathcal{C} \leftarrow \mathcal{C}$ by $F_a\ b = F\langle a, b\rangle$. If there exists an initial F_a-algebra for all a, then F has a (unary) *type functor* $\tau F : \mathcal{C} \leftarrow \mathcal{D}$, defined by:

$$\begin{aligned} \tau F\ a &= \mu(F_a) \\ \tau F\ f &= (\![\mathsf{in} \circ F\ f\ \mathsf{id}]\!) \end{aligned}$$

The initial algebra has functorial typing

$$\mathsf{in} : \tau F \leftarrow F \cdot \mathsf{Id} \,\Delta (\tau F)$$

For example, $\mathit{List} = \tau L$, where $L\langle a, b\rangle = (a \times b) + 1$, with initial algebra

$$\mathsf{in} : \mathit{List}\ a \leftarrow (a \times \mathit{List}\ a) + 1$$

Putting $\mathsf{cons} = \mathsf{in} \circ \mathsf{inl}$, $\mathsf{nil} = \mathsf{in} \circ \mathsf{inr}$, we have

$$\begin{aligned} &\mathsf{cons} : \mathit{List}\ a \leftarrow a \times \mathit{List}\ a \\ &\mathsf{nil}\ \ \ : \mathit{List}\ a \leftarrow 1 \end{aligned}$$

Conversely, $\mathsf{in} = \mathsf{cons} \triangledown \mathsf{nil}$. For further details, consult [7] or [1].

A grammar for regular functors A functor built only from 1, Ex, $+$, $\times$, Δ, $\cdot$ and τ is called a *regular* functor. A formal context-free grammar for the n-ary regular functors is:

$$\begin{array}{rlll} \mathrm{F}^{(n)} ::= & 1^{\mathsf{K}} & & n\text{-ary constant functor} \\ | & \mathsf{Ex}_i^n & & \text{projection, } i = 0, \ldots, n-1 \\ | & + \mid \times & (\text{only if } n = 2) & \text{binary sum and product} \\ | & \Delta_i^k \mathrm{F}_i^{(n)} & & \text{functor tupling} \\ | & \mathrm{F}_0^{(k)} \cdot \mathrm{F}_1^{(n)} & & \text{functor composition} \\ | & \tau \mathrm{F}^{(2)} & (\text{only if } n = 1) & \text{the type functor induced by } \mathrm{F}^{(2)} \end{array}$$

In the rule for functor composition, the target 'type' (a category) of F_1 should, of course, be of the form $\Pi_i^k \mathcal{C}_i$.

This can be extended to arbitrary finite sums and products, and τF can be defined to produce n-ary functors from $(n+1)$-ary functors, but these extensions can also be handled via the obvious isomorphisms such as $\mathcal{C}^{(n+1)} \simeq \mathcal{C} \times \mathcal{C}^n$. Here is how the functor *List* is produced by this grammar:

$$\mathit{List} = \tau(+ \cdot \times \Delta 1^{\mathsf{K}})$$

4 Functor pulling

Let an m-ary functor H be given. We want to define polytypic functions p such that for n-ary F:

$$p_F \;:\; \begin{array}{l} H\,\langle F\,\langle a_{0,0},\ a_{0,1},\ \ldots,\ a_{0,n-1}\rangle, \\ \quad F\,\langle a_{1,0},\ a_{1,1},\ \ldots,\ a_{1,n-1}\rangle, \\ \qquad \vdots \\ \quad F\,\langle a_{m-1,0},\ a_{m-1,1},\ \ldots,\ a_{m-1,n-1}\rangle\,\rangle \end{array} \leftarrow \begin{array}{l} F\,\langle H\,\langle a_{0,0},\ a_{1,0},\ \ldots,\ a_{m-1,0}\rangle, \\ \quad H\,\langle a_{0,1},\ a_{1,1},\ \ldots,\ a_{m-1,1}\rangle, \\ \qquad \vdots \\ \quad H\,\langle a_{0,n-1},\ a_{1,n-1},\ \ldots,\ a_{m-1,n-1}\rangle\,\rangle \end{array}$$

Such a function pulls, so to speak, functor H out of the inside of its parameter F. An example of such a function for $H = \times$ is the product puller:

$$\mathsf{unzip}_F = F\mathsf{exl}^* \vartriangle F\mathsf{exr}^* \;:\; F\lozenge_i a_i \times F\lozenge_i b_i \leftarrow F\lozenge_i(a_i \times b_i)$$

For the choice $H = a^{\mathsf{K}}$ the typing of p_F specializes to:

$$p_F \;:\; a \leftarrow F\,\langle a, a, \ldots, a\rangle$$

which is the type of polytypic "crush". Thus, the problem we are addressing here indeed generalizes the notion of crushes.

Not only must p_F be polytypic in F, it should also be polymorphic in the type variables a_{ij}. Using transposition functors, the typing of p_F can be rendered more succinctly and naturally as:

$$p_F \;:\; H\cdot F^*\cdot\mathsf{T} \leftarrow F\cdot H^*$$

Since ${}_1\mathsf{T}_m = \mathsf{Id}$, this simplifies to

$$p_F \;:\; H\cdot F^* \leftarrow F\cdot H$$

when F is unary — or is viewed as such, which is always possible since $\Pi_i^n \mathcal{C}_i = (\Pi_i^n \mathcal{C}_i)^1$.

The polytypic function unzip defined above is completely generic; the only requirement on F here is that it is a functor between two categories that have products. In general we are not so lucky, and need to assume that F is regular, so that we can define p_F as a polytypic combinator by induction on the structure of F. In the process we shall see what ingredients are needed for the "body" of p_F. As was done for crush, we make a concerted effort to minimize the number of ingredients that must supplied to the combinator: whenever possible, we take whatever will do when it is available "for free".

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

$$p_F \;:\; H\cdot F^*\cdot\mathsf{T} \leftarrow F\cdot H^*$$

for sufficiently simple F. We postpone the case 1^{K} to the last.

Case Ex_i: the requirement is $p_{\mathsf{Ex}_i} : H{\cdot}{\mathsf{Ex}_i}^*{\cdot}\mathsf{T} \leftarrow \mathsf{Ex}_i{\cdot}H^* = H{\cdot}\mathsf{Ex}_i \leftarrow H{\cdot}\mathsf{Ex}_i$.
The choice is obvious: $p_{\mathsf{Ex}_i} = \mathsf{id}$. So for this case we need not supply an ingredient to the combinator.

Case $+$: the requirement is $p_+ : H{\cdot}+^*{\cdot}\mathsf{T} \leftarrow +{\cdot}H^2$.
Spelling it out with type variables, the type of p_+ is $H\lozenge_i(a_i{+}b_i) \leftarrow H\lozenge_i a_i + H\lozenge_i b_i$. Using $\lozenge\mathsf{inl} : \lozenge_i(a_i{+}b_i) \leftarrow \lozenge_i a_i$ and $\lozenge\mathsf{inr} : \lozenge_i(a_i{+}b_i) \leftarrow \lozenge_i b_i$, we see that we can use

$$p_+ = H\lozenge\mathsf{inl} \triangledown H\lozenge\mathsf{inr}$$

Case $\times$: the requirement is $p_\times : H{\cdot}\times^*{\cdot}\mathsf{T} \leftarrow \times{\cdot}H^2$.
This has no free solution. For example, if $H = +$, the requirement boils down to $p_\times : (a_0{\times}b_0) + (a_1{\times}b_1) \leftarrow (a_0{+}a_1) \times (b_0{+}b_1)$, which in *Set* has no polymorphic solution (take $a_0 = b_1 = 0, a_1 = b_0 = 1$, showing that $+$ cannot be pulled in *Set* and other categories having no arrows with typing $0 \leftarrow 1$). So some ingredient $\oplus : H{\cdot}\times^*{\cdot}\mathsf{T} \leftarrow \times{\cdot}H^2$ will have to be supplied.

Case $\Delta_i F_i$: the requirement is $p_{\Delta_i F_i} : H{\cdot}(\Delta_i F_i)^*{\cdot}\mathsf{T} \leftarrow \Delta_i F_i{\cdot}H^*$.
By the inductive hypothesis we have

$$p_{F_i} : H{\cdot}{F_i}^*{\cdot}\mathsf{T} \leftarrow F_i{\cdot}H^*$$

so the tuple

$$\begin{aligned} \lozenge_i p_{F_i} &: \Delta_i(H{\cdot}(F_i)^*{\cdot}\mathsf{T}) \leftarrow \Delta_i(F_i{\cdot}H^*) \\ &= H^*{\cdot}\Delta_i(F_i)^*{\cdot}\mathsf{T} \;\leftarrow \Delta_i F_i{\cdot}H^* \end{aligned}$$

has the required typing.

Case $F{\cdot}G$: the requirement is $p_{F\cdot G} : H{\cdot}(F{\cdot}G)^*{\cdot}\mathsf{T} \leftarrow F{\cdot}G{\cdot}H^*$.
By the inductive hypothesis, viewing F as unary, we have

$$\begin{aligned} p_F &: H{\cdot}F^* \quad\;\; \leftarrow F{\cdot}H \\ p_G &: H{\cdot}G^*{\cdot}\mathsf{T} \leftarrow G{\cdot}H^* \end{aligned}$$

so that

$$\begin{aligned} p_F \quad &: H{\cdot}F^*{\cdot}G^*{\cdot}\mathsf{T} \;\leftarrow F{\cdot}H{\cdot}G^*{\cdot}\mathsf{T} \\ &= H{\cdot}(F{\cdot}G)^*{\cdot}\mathsf{T} \leftarrow F{\cdot}H{\cdot}G^*{\cdot}\mathsf{T} \\ Fp_G \; &: \qquad\qquad\qquad\quad\;\; F{\cdot}H{\cdot}G^*{\cdot}\mathsf{T} \leftarrow F{\cdot}G{\cdot}H^* \end{aligned}$$

By composing these two we obtain, for free,

$$p_F \circ F p_G$$

as having the required typing.

Case τF: the requirement is $p_{\tau F} \;:\; H\cdot(\tau F)^* \leftarrow (\tau F)\cdot H$.
Using $\tau F\ a \;=\; \mu(F_a) \;=\; \mu(b \mapsto F\langle a, b\rangle)$, we pattern match the required typing against

$$
\begin{array}{ll}
 & (\!|f|\!) \;:\; H(Ga) \leftarrow \tau F(Ha) \\
\Leftarrow & \qquad \{\ \text{catamorphism typing}\ \} \\
 & f \;:\; H(Ga) \leftarrow F\langle Ha, H(Ga)\rangle
\end{array}
$$

With functorial typing (abstracting from the type variable), this amounts to:

$$(\!|f|\!) \;:\; H\cdot G \leftarrow \tau F\cdot H \quad \Leftarrow \quad f \;:\; H\cdot G \leftarrow F\cdot H^2\cdot\mathsf{Id}\,\Delta\, G$$

So we see that for $p_{\tau F}$ we can use a catamorphism

$$(\!|f|\!) \;:\; H\cdot(\tau F)^* \leftarrow (\tau F)\cdot H$$

provided that we can construct an f with the typing

$$f \;:\; H\cdot(\tau F)^*\cdot \leftarrow F\cdot H^2\cdot\mathsf{Id}\,\Delta(\tau F)^*$$

By the inductive hypothesis we have

$$p_F \;:\; H\cdot F^*\cdot\mathsf{T} \leftarrow F\cdot H^2$$

so that

$$
\begin{array}{lll}
p_F \;:\; H\cdot F^*\cdot\mathsf{T}\cdot\mathsf{Id}\,\Delta(\tau F)^* & \leftarrow & F\cdot H^2\cdot\mathsf{Id}\,\Delta(\tau F)^* \\
\quad = H\cdot(F\cdot\mathsf{Id}\,\Delta\tau F)^* & \leftarrow & F\cdot H^2\cdot\mathsf{Id}\,\Delta(\tau F)^*
\end{array}
$$

Further we have function $\mathsf{in} \;:\; \tau F \leftarrow F\cdot\mathsf{Id}\,\Delta(\tau F)$, so that

$$H\between\mathsf{in} \;:\; H\cdot(\tau F)^* \leftarrow H\cdot(F\cdot\mathsf{Id}\,\Delta\tau F)^*$$

The free solution for this case is therefore $p_{\tau F} = (\!|H\between\mathsf{in} \circ p_F|\!)$.

Case 1^{K}: the requirement is $p_{1^{\mathsf{K}}} \;:\; H\cdot 1^{\mathsf{K}*}\cdot\mathsf{T} \leftarrow 1^{\mathsf{K}}\cdot H^* = (H\between 1)^{\mathsf{K}} \leftarrow 1^{\mathsf{K}}$.
So $p_{1^{\mathsf{K}}} = e$ where $e \;:\; H\between 1 \leftarrow 1$ (non-functorially typed). As for the case $\times$, this has no free solution (consider $H = 0^{\mathsf{K}}$). In the construction of crushes, with $H = a^{\mathsf{K}}$, this specializes to

$e : a \leftarrow 1$. There e was taken to be the neutral element of $\oplus$, the ingredient needed for the case $\times$. For general H, neutrality of $\oplus$ in the usual sense is meaningless; only operations of some type $a \leftarrow a \times a$ can have neutral elements, and the typing of $\oplus$ does not have that form. We can, however, define a generalized notion of neutrality. Consider

$$\mathsf{exr} : H\between_i a_i \leftarrow \mathsf{1} \times H\between_i a_i$$

Using $\oplus$ and e we have *another* way of constructing a function with this typing, namely:

$$H\between_i \mathsf{exr} \circ (\oplus) \circ e \times \mathsf{id}$$

This has the same typing since:

$$\begin{array}{lllll} H\between \mathsf{exr} & : H\between_i a_i \leftarrow H\between_i(\mathsf{1} \times a_i) & & & \\ \quad\quad \oplus & : & H\between_i(\mathsf{1} \times a_i) \leftarrow H\between_i \mathsf{1} \times H\between_i a_i & & \\ \quad\quad\quad\quad e \times \mathsf{id} & : & & H\between_i \mathsf{1} \times H\between_i a_i \leftarrow \mathsf{1} \times H\between_i a_i & \end{array}$$

We require now that these two equi-typed generic functions are equal. A similar coherence condition is obtained by switching left and right. Moreover, we require that these two requirements combined have a unique solution. So, define an element e to be H-neutral for $\oplus$ when:

$$H\between_i \mathsf{exr} \circ (\oplus) \circ e \times \mathsf{id} \;=\; \mathsf{exr} \quad \wedge \quad H\between_i \mathsf{exl} \circ (\oplus) \circ \mathsf{id} \times e \;=\; \mathsf{exl}$$

The requirement is now that $\oplus$ has a *unique* H-neutral element, and then $p_{\mathsf{1}^{\mathsf{K}}} = e$. Specialization to a^{K}-neutrality gives the conventional notion of neutrality.
$\square$

We introduce now a notation for p_F thus constructed, namely $\langle\!\langle \oplus \rangle\!\rangle$, the same notation used for the special case of crush.

Summary Given a functor H, for $\oplus : H{\cdot}\times^*{\cdot}\mathsf{T} \leftarrow \times{\cdot}H^2$ with unique H-neutral element $e : H\between \mathsf{1} \leftarrow \mathsf{1}$, the H puller

$$\langle\!\langle \oplus \rangle\!\rangle_F : H{\cdot}F^*{\cdot}\mathsf{T} \leftarrow F^*{\cdot}H$$

is inductively defined on regular functors by:

$$\begin{array}{lll} \langle\!\langle \oplus \rangle\!\rangle_{\mathsf{1}^{\mathsf{K}}} & = & e \\ \langle\!\langle \oplus \rangle\!\rangle_{\mathsf{Ex}_i} & = & \mathsf{id} \\ \langle\!\langle \oplus \rangle\!\rangle_{+} & = & H\between \mathsf{inl} \triangledown H\between \mathsf{inr} \\ \langle\!\langle \oplus \rangle\!\rangle_{\times} & = & \oplus \\ \langle\!\langle \oplus \rangle\!\rangle_{\triangle_i F_i} & = & \between_i \langle\!\langle \oplus \rangle\!\rangle_{F_i} \\ \langle\!\langle \oplus \rangle\!\rangle_{F \cdot G} & = & \langle\!\langle \oplus \rangle\!\rangle_F \circ F\langle\!\langle \oplus \rangle\!\rangle_G \\ \langle\!\langle \oplus \rangle\!\rangle_{\tau F} & = & (\![H\between \mathsf{in} \circ \langle\!\langle \oplus \rangle\!\rangle_F]\!) \end{array}$$

Example specialization Specialization of $\langle\!\langle \oplus \rangle\!\rangle_{List}$, where $List = \tau(+ \cdot \times \Delta 1^{\mathsf{K}})$, gives:

$$
\begin{array}{ll}
& \langle\!\langle \oplus \rangle\!\rangle_{List} \\
= & \quad \{ \text{ definition of } \langle\!\langle \oplus \rangle\!\rangle \ \} \\
& (\![H \between \mathsf{in} \circ (H \between \mathsf{inl}) \triangledown (H \between \mathsf{inr}) \circ (\oplus) + e]\!) \\
= & \quad \{ \text{ rules for } + \text{ and } \triangledown ; H \text{ is functor } \} \\
& (\![(H \between (\mathsf{in} \circ \mathsf{inl}) \circ (\oplus)) \triangledown (H \between (\mathsf{in} \circ \mathsf{inr}) \circ e)]\!)
\end{array}
$$

5 Examples

In this section we give two examples of “functor pullers” that are useful in many different problems. We freely mix functional-programming idiom with categorical notation.

Cross One application of $\langle\!\langle \_ \rangle\!\rangle$ is to construct a generalization of the cross-product of two sets. We assume a type-constructor *Set*, so, for example, values of type *Set Int* are sets of naturals. *Set* is made into a functor by defining

$$(Set\, f)\ xs \ = \ \{ f\ x \mid x \leftarrow xs \}$$

We want cross_F to be a generalization of

$$
\begin{array}{l}
\mathsf{cross}_\times \ :\ Set(a \times b) \leftarrow (Set\ a) \times (Set\ b) \\
\mathsf{cross}_\times \ \langle xs, ys \rangle \ = \ \{ \langle x, y \rangle \mid x \leftarrow xs, y \leftarrow ys \}
\end{array}
$$

Here a pair of sets is turned into a set of pairs. So *Set* is “pulled out”. For *List* we should have, for example:

$$\mathsf{cross}_{List}[\{u, v\}, \{x, y, z\}] = \{[u, x], [u, y], [u, z], [v, x], [v, y], [v, z]\}$$

In general an F-structure of sets is turned into a set of F-structures, *one for every way of choosing its elements from the constituent sets*. We apply this intuition to govern the construction of the *Set*-neutral element e, which must have typing $Set\ 1 \leftarrow 1$, that is, it is a “constant” of type *Set* 1. Now there are two values of this type: the empty set, or the singleton set $\{\bullet\}$, in which $\bullet$ stands for the single inhabitant of the unit type 1. Since structures of the source type have no constituent sets, we have no freedom in making choices: there is exactly one way “choosing an element from the constituent sets”. The result must therefore be a singleton set, and so $e \ \bullet \ = \ \{\bullet\}$. This solution satisfies, as required, the coherence conditions, while the other choice does not.

We define then, generically, $\mathsf{cross} = \langle\!\langle \mathsf{cross}_\times \rangle\!\rangle$, with $\mathsf{cross}_\times$ as defined above.

From the previous section we have the specialization

$$\mathsf{cross}_{List} = (\![(Set(\mathsf{in} \circ \mathsf{inl}) \circ \mathsf{cross}_\times) \triangledown (Set(\mathsf{in} \circ \mathsf{inr}) \circ \{\bullet\}^{\mathsf{K}})]\!)$$

Using $(Set\, f)\,\{x \mid x \leftarrow \cdots\} = \{f\ x \mid x \leftarrow \cdots\}$, and recalling that $\mathsf{in} \circ \mathsf{inl}$ and $\mathsf{in} \circ \mathsf{inr}$ correspond to the list constructors cons and nil, respectively, we can express this in conventional functional-programming idiom:

$$\begin{array}{lll} \mathsf{cross}_{List} & = & \mathsf{foldr}\ \langle \copyright, n \rangle\ \mathbf{where} \\ & & xs \copyright ys \ = \ \{\mathsf{cons}\ \langle x, y\rangle \mid x \leftarrow xs, y \leftarrow ys\} \\ & & n \qquad\quad = \ \{\mathsf{nil}\} \end{array}$$

Jeuring [5] defines two mutually-recursive polytypic functions cross and cp, the latter for using with type functors. With the present approach, a single definition does the job: $\mathsf{cross}_{\tau F}$ is cp.

Full Let the type constructor *Maybe* be defined as:

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

Consider a structure of some type $F\langle\rangle_i(Maybe\ a_i)$. If every "maybe" position is filled, that is, has a value of the form $\mathsf{one}\ x$, we can turn the whole structure into one of type $F\langle\rangle_i a_i$. Otherwise this is impossible: there is no generic way of inventing values to fill the missing entries none. So we can, at best, *maybe* deliver a structure of type $F\langle\rangle_i a_i$. The idea can be encapsulated in a generic function

$$\mathsf{full}_F \ : \ Maybe(F\langle\rangle_i a_i) \leftarrow F\langle\rangle_i(Maybe\ a_i)$$

In other words, we want to pull the internal *Maybe*'s to the outside. We need to define $\mathsf{full}_\times$. We express it by functional-programming-style pattern match:

$$\begin{array}{lll} \mathsf{full}_\times\ \langle \mathsf{one}\ x, \mathsf{one}\ y\rangle & = & \mathsf{one}\ \langle x, y\rangle \\ \mathsf{full}_\times\ \langle\ \_,\qquad \_\quad \rangle & = & \mathsf{none} \end{array}$$

(If both positions of the pair are filled, a pair is returned. Otherwise, at least one is missing, and none is returned.) Next we must find a *Maybe*-neutral element e. By the same reasoning as for cross, we find that $e = (\mathsf{one}\ \bullet)^{\mathsf{K}}$.

There is a natural transformation to *Set* from *Maybe*, namely:

$$\begin{array}{lll} \mathsf{setify}\ \ (\mathsf{one}\ x) & = & \{x\} \\ \mathsf{setify}\ \ \mathsf{none} & = & \{\ \} \end{array}$$

in which the last r.h.s. denotes, of course, the empty set. (This is the specialization for *Maybe* of generic $\mathsf{setify} = \langle\!\langle \cup \rangle\!\rangle \circ F \Diamond \{\_\}$.) This makes it possible to express a relationship between cross and full, which we state without proof:

$$\mathsf{setify} \circ \mathsf{full}_F \;=\; \mathsf{cross}_F \circ F^{\star}\, \mathsf{setify}$$

Monads Both *Set* and *Maybe* are the functor of a monad, and so the question is if both cases above are instances of a generic construction for monads. The answer appears to be yes, provided that the functor is strong; the details have not been worked out yet, though.

Not all functor pullers arise from this monadic construction. Since $\times$ is not an endofunctor, unzip_F — which coincides with $\langle\!\langle \mathsf{unzip}_\times \rangle\!\rangle_F$ for regular functor F — cannot arise from a monad. Although a^{K} is an endofunctor, it is not the functor of a monad unless $a \simeq \mathbf{1}$, so the same holds for crushes.

6 Research questions

Half-zips Hoogendijk & Backhouse [3, 2] define even more generic "half-zips" that commute two type constructors. This is done in an allegorical setting, in which there exist arrows with typing $0 \leftarrow 1$, which makes comparison with the results here a non-trivial exercise. The relationship needs further study and clarification.

The monadic construction The details of the construction for monadic-functor pullers mentioned at the end of Section 5 should be worked out

Uniqueness of neutral element We required uniqueness of H-neutral elements. It is a simple exercise in algebra to show that "normal" neutral elements of an operation $\oplus \,:\, a \leftarrow a \times a$ are always unique. In all known examples of polymorphic functions $\oplus \,:\, H{\cdot}\times^{*}{\cdot}\mathsf{T} \leftarrow \times{\cdot}H^2$, any H-neutral element — if such an element exists at all — is also unique, but at present it is unknown if this is necessarily so.

Calculational theory What are the calculational rules for these functor pullers? Regular functors appear to be polymorphic arrows in the category of small categories. Can we use higher-order parametricity to boost calculation?

Beyond regularity Is it possible to extend the theory to non-regular functors? What is special about $\times$ that it is so easily pulled?

Specification Can we (declaratively) *specify* function pullers fully, instead of partially by their typings, which might have several inhabitants? How do we know that the functor puller constructed is the right generalization of specific instances (a question that pertains to more parts of polytypic programming). When does the typing guarantee uniqueness of inhabitants, if any?

References

[1] Richard Bird and Oege de Moor. *Algebra of Programming*, volume 100 of *Prentice Hall International Series In Computer Science*. Prentice Hall, 1997.

[2] Paul Hoogendijk. *A Generic Theory of Data Types*. PhD thesis, Eindhoven University of Technology, 1997. `www.win.tue.nl/cs/wp/papers/papers.html`.

[3] Paul Hoogendijk and Roland Backhouse. When do datatypes commute? In E. Moggi and G. Rosolini, editors, *Category Theory and Computer Science*, pages 242–260. Springer-Verlag, 1997. LNCS 1290. `www.win.tue.nl/cs/wp/papers/papers.html`.

[4] Patrik Jansson and Johan Jeuring. PolyP — a polytypic programming language extension. In *POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 470–482. ACM Press, 1997. `www.cs.chalmers.se/~johanj/publications.html`.

[5] Johan Jeuring. Polytypic pattern matching. In *Conference Record of FPCA '95, SIGPLAN-SIGARCH-WG2.8 Conference on Functional Programming Languages and Computer Architecture*, pages 238–248, 1995. `www.cs.chalmers.se/~johanj/publications.html`.

[6] Johan Jeuring and Patrik Jansson. Polytypic programming. In J. Launchbury, E. Meijer, and T. Sheard, editors, *Advanced Functional Programming, Second International School*, pages 68–114. Springer-Verlag, 1996. LNCS 1129. `www.cs.chalmers.se/~johanj/publications.html`.

[7] Grant Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14(2–3):255–279, 1990.

[8] Lambert Meertens. Calculate polytypically! In Herbert Kuchen and S. Doaitse Swierstra, editors, *Programming Languages: Implementations, Logics, and Programs. Proceedings Eighth International Symposium PLILP '96*, volume 1140 of *LNCS*, pages 1–16. Springer-Verlag, 1996.