Accepted to WGP '11

Towards a Categorical Foundation for Generic Programming

Ralf Hinze Nicolas Wu
Department of Computer Science,
University of Oxford,
Wolfson Building, Parks Road,
Oxford, OX1 3QD, England
{ralf.hinze, nicolas.wu}@cs.ox.ac.uk

Abstract

Generic Haskell is an extension of Haskell that supports datatype-generic programming. The central idea of Generic Haskell is to interpret a type by a function, the so-called instance of a generic function at that type. Since types in Haskell include parametric types such as 'list of', Generic Haskell represents types by terms of the simply-typed lambda calculus. This paper puts the idea of interpreting types as functions on a firm theoretical footing, exploiting the fact that the simply-typed lambda calculus can be interpreted in a cartesian closed category. We identify a suitable target category, a subcategory of **Cat**, and argue that slice, coslice and comma categories are a good fit for interpreting generic functions at base types.

Categories and Subject Descriptors D.3.1 [*Formal Definitions and Theory*]: Semantics; D.3.3 [*Programming Languages*]: Language Constructs and Features; F.3.2 [*Semantics of Programming Languages*]: Denotational semantics

General Terms Languages

Keywords Generic programming, category theory, slice category, comma category

1. Introduction

Datatype-generic programming (DGP) aims at making your life as a programmer easier by making your programs more general and more robust. Haskell offers rudimentary support for DGP in the form of the deriving mechanism. Instead of manually coding, say, equality for a datatype, the Haskell programmer attaches a **deriving** *Eq* clause to the datatype declaration. The clause instructs the compiler to auto-generate the class methods of *Eq*, equality and inequality. Simple, convenient and robust. If the datatype is changed at a later point in time, equality and inequality are modified accordingly behind the scenes.

Haskell's support for DGP is only partial, however, since the deriving mechanism is limited to a few predefined classes. In particular, the Haskell programmer cannot define her own derivable classes. This is exactly what DGP allows you to do. Informally, a derivable or generic function is defined by induction on the structure of types. Typically, the generic programmer provides code for some type constructs, the rest is taken care of automatically.

The last two decades have witnessed a multitude of proposals for DGP, differing in convenience, expressiveness and efficiency. We can roughly identify three (overlapping) periods:

- classicism (1995 –): strong background in category theory;
- romanticism (2000 –): shift towards type-theoretic approaches;
- realism (2005 –): compiler extensions and library development.

The language extension PolyP [15] is representative of the first period. It grew out of the work on the Algebra of Programming (AoP) with its emphasis on structured recursion operators (folds and unfolds). PolyP is based on a grammar for bifunctors and regular functors, so it does not cover the whole of Haskell's expressive type system. This weakness was overcome by Generic Haskell (GH) [12], a representative of the second period. GH uses the simply-typed lambda calculus to represent Haskell types. GH deviates from PolyP in that it handles type recursion implicitly, obviating the need for recursion operators. On the negative side, the generic programmer is even barred from providing an instance for type recursion, which is sometimes limiting. Finally, the third period saw a flood of proposals that aimed at supporting DGP natively within a host language. (PolyP and GH are both implemented as pre-processors.) While the approaches differ wildly in the mechanics—the way types and elements of types are represented—the principle of DGP is unchanged: (representations of) types are interpreted by functions.

This paper aspires to initiate the neoclassical period, in that it unites classical (AoP) with romantic elements (GH). Specifically, the paper makes the following contributions:

- we put the idea of interpreting types as functions on a firm theoretical footing, exploiting the fact that the simply-typed lambda calculus can be interpreted in a cartesian closed category;
- we argue that slice, coslice and comma categories are suitable for interpreting generic functions at base types;
- we show that type recursion can be handled explicitly, simply by adding type constants for least and greatest fixed points;
- we work through one example in considerable depth, providing a logical reconstruction of crush.

In a sense, we liberate GH from its origins in **Cpo**, providing a framework that can be instantiated to other categories of interest.

The rest of the paper is structured as follows. Section 2 briefly reviews GH (using material from [10]). Section 3 shows how to interpret a Haskell type as a functor. We build upon two standard results, namely, that the simply-typed lambda calculus can be interpreted in a cartesian closed category, and that **Cat**, the category of all small categories, is cartesian closed. For a categorical model of GH we then simply have to choose a suitable base category for interpreting types of kind $\star$. Section 4 argues that slice categories fit the bill for simple generic consumers. Section 5 dualises the constructors to simple generic producers and Section 6 gives the general construction of which the first two are special cases. Finally, Section 7 reviews related work and Section 8 concludes. (The Appendix contains supplementary material.)

2. Recap: Generic Haskell

This section serves as a short introduction to Generic Haskell (GH), illustrating the concepts of *type-indexed values* and *kind-indexed types* by means of a worked-out example: mapping functions. Before tackling the generic definition of map, we first look at different datatypes and associated mapping functions.

As a first example consider the list datatype.

$$\mathbf{data}\ \mathsf{List}\ a = \mathit{Nil} \mid \mathit{Cons}\ a\ (\mathsf{List}\ a)$$

Actually, List is not a type but a unary type constructor. In Haskell the 'type' of a type constructor is specified by the kind system. For instance, List has kind $\star \to \star$. The kind $\star$ represents types that contain values. The kind $\mathfrak{T} \to \mathfrak{U}$ represents type constructors that map type constructors of kind $\mathfrak{T}$ to those of kind $\mathfrak{U}$. The mapping function for List, called map_{List}, is given by

$$\begin{aligned}
&map_{\mathsf{List}} :: \forall a_1\ a_2\ .\ (a_1 \to a_2) \to (\mathsf{List}\ a_1 \to \mathsf{List}\ a_2)\\
&map_{\mathsf{List}}\ map_a\ \mathit{Nil} = \mathit{Nil}\\
&map_{\mathsf{List}}\ map_a\ (\mathit{Cons}\ a\ as) = \mathit{Cons}\ (map_a\ a)\ (map_{\mathsf{List}}\ map_a\ as)\ .
\end{aligned}$$

Observe that the definition of map_{List} rigidly follows the structure of the datatype.

The List type constructor is an example of a regular type, which can be defined as the least fixed point of a functor. In fact, Haskell is expressive enough to rephrase List using an explicit fixed point operator. We repeat this construction here as it provides us with interesting examples of datatypes and associated mapping functions. First, we define the so-called base functor of List.

$$\mathbf{data}\ \mathsf{ListF}\ a\ b = \mathit{Nil} \mid \mathit{Cons}\ a\ b$$

The type constructor ListF has kind $\star \to (\star \to \star)$. The definition below introduces a fixed point operator on the type level.

$$\mathbf{newtype}\ \mathsf{Fix}\, f = \mathit{In}\ (f\ (\mathsf{Fix}\, f))$$

The kind of Fix is $(\star \to \star) \to \star$. Using Fix we can re-define List as a fixed point of its base functor.

$$\mathbf{type}\ \mathsf{List}'\ a = \mathsf{Fix}\ (\mathsf{ListF}\ a)$$

How can we define the mapping function for lists thus defined? For a start, we define the mapping function for the base functor.

$$\begin{aligned}
&map_{\mathsf{ListF}} :: \forall a_1\ a_2\ .\ (a_1 \to a_2) \to \forall b_1\ b_2\ .\ (b_1 \to b_2)\\
&\qquad\qquad \to (\mathsf{ListF}\ a_1\ b_1 \to \mathsf{ListF}\ a_2\ b_2)\\
&map_{\mathsf{ListF}}\ map_a\ map_b\ \mathit{Nil} = \mathit{Nil}\\
&map_{\mathsf{ListF}}\ map_a\ map_b\ (\mathit{Cons}\ a\ b) = \mathit{Cons}\ (map_a\ a)\ (map_b\ b)
\end{aligned}$$

Since the base functor has two type arguments, its mapping function takes two functions, map_a and map_b, and applies them to values of type a_1 and b_1, respectively. More interesting is

$$\begin{aligned}
&map_{\mathsf{Fix}} :: \forall f_1\ f_2\ .\ (\forall a_1\ a_2\ .\ (a_1 \to a_2) \to (f_1\ a_1 \to f_2\ a_2))\\
&\qquad\qquad \to (\mathsf{Fix}\, f_1 \to \mathsf{Fix}\, f_2)\\
&map_{\mathsf{Fix}}\ map_f\ (\mathit{In}\ v) = \mathit{In}\ (map_f\ (map_{\mathsf{Fix}}\ map_f)\ v)\ ,
\end{aligned}$$

which takes a polymorphic function as an argument. The argument, map_f, has a more general type than one would probably expect: it takes a function of type $a_1 \to a_2$ to a function of type $f_1\, a_1 \to f_2\, a_2$. By contrast, the mapping function for List (which like f has kind $\star \to \star$) takes $a_1 \to a_2$ to $\mathsf{List}\ a_1 \to \mathsf{List}\ a_2$. The definition of $map_{\mathsf{List}'}$ demonstrates that the extra generality is necessary.

$$\begin{aligned}
&map_{\mathsf{List}'} :: \forall a_1\ a_2\ .\ (a_1 \to a_2) \to (\mathsf{List}'\ a_1 \to \mathsf{List}'\ a_2)\\
&map_{\mathsf{List}'}\ map_a = map_{\mathsf{Fix}}\ (map_{\mathsf{ListF}}\ map_a)
\end{aligned}$$

The argument of map_{Fix}, which is $map_{\mathsf{ListF}}\ map_a$, has the polymorphic type $\forall b_1\ b_2\ .\ (b_1 \to b_2) \to (\mathsf{ListF}\ a_1\ b_1 \to \mathsf{ListF}\ a_2\ b_2)$. In other words, f_1 is instantiated to $\mathsf{ListF}\ a_1$ and f_2 to $\mathsf{ListF}\ a_2$.

Now, let us define a generic version of map. The instances above indicate that the type of the mapping function depends on the kind of the type index. In fact, the type of map can be defined by induction on the structure of kinds. For a type t of kind $\star$ the mapping function $map_{t::\star}$ equals the identity function. Hence, its type is $t \to t$. In general, the mapping function $map_{t::\mathfrak{T}}$ has type $Map_{\mathfrak{T}}\ t\ t$, where $Map_{\mathfrak{T}}$ is inductively defined

$$\begin{aligned}
&Map_{\star}\ t_1\ t_2 = t_1 \to t_2\ ;\\
&Map_{\mathfrak{T} \to \mathfrak{U}}\ t_1\ t_2 = \forall x_1\ x_2\ .\ Map_{\mathfrak{T}}\ x_1\ x_2 \to Map_{\mathfrak{U}}\ (t_1\ x_1)\ (t_2\ x_2)\ .
\end{aligned}$$

In the base case $Map_{\star}\, t_1\, t_2$ equals the type of a conversion function. The inductive case has a characteristic form. It specifies that a 'conversion function' between the type constructors t_1 and t_2 is a function that maps a conversion function between x_1 and x_2 to a conversion function between $t_1\ x_1$ and $t_2\ x_2$, for all possible instances of x_1 and x_2. The type signatures we have encountered before are instances of this scheme.

How can we define the generic mapping function itself? It turns out that this is surprisingly easy. To define a generic value it suffices to give cases for primitive types, the unit type, sums, and products, where the latter three types are defined

$$\begin{aligned}
&\mathbf{data}\ 1 = ()\\
&\mathbf{data}\ a + b = \mathit{Inl}\ a \mid \mathit{Inr}\ b\\
&\mathbf{data}\ a \times b = (a, b)\ .
\end{aligned}$$

Assuming that we have only one primitive type, Int, the generic mapping function is given by

$$\begin{aligned}
&map_{t::\mathfrak{T}} :: Map_{\mathfrak{T}}\ t\ t\\
&map_{\mathit{Int}}\ i = i\\
&map_{1}\ () = ()\\
&map_{+}\ map_a\ map_b\ (\mathit{Inl}\ a) = \mathit{Inl}\ (map_a\ a)\\
&map_{+}\ map_a\ map_b\ (\mathit{Inr}\ b) = \mathit{Inr}\ (map_b\ b)\\
&map_{\times}\ map_a\ map_b\ (a, b) = (map_a\ a, map_b\ b)\ .
\end{aligned}$$

This straightforward definition contains all the ingredients needed to derive maps for arbitrary datatypes of arbitrary kinds. In fact, all the definitions we have seen before are instances of this scheme.

While generic mapping functions preserve the structure of the base functor, a reduction, or crush, is a generic function that collapses such a structure into a single value. An example of this is $size$, which is simply a generalisation of $length :: \forall a\ .\ \mathsf{List}\, a \to \mathit{Int}$ that works on arbitrary container types. The size function for a list is defined

$$\begin{aligned}
&size_{\mathsf{List}} :: \forall a\ .\ (a \to \mathit{Int}) \to (\mathsf{List}\ a \to \mathit{Int})\\
&size_{\mathsf{List}}\ size_a\ \mathit{Nil} = 0\\
&size_{\mathsf{List}}\ size_a\ (\mathit{Cons}\ a\ as) = size_a\ a + size_{\mathsf{List}}\ as\ .
\end{aligned}$$

Instantiating $size_a$ to $const\ 1$ gives us the familiar $length$ function over lists, and instantiating it to id gives the sum function over lists.

Defining a generic version of $size$ can be done in much the same way as the previous map example. As before, we define $size$ by using induction on the structure of kinds. The generic function $size_{t::\mathfrak{T}}$ has type $Size_{\mathfrak{T}}\ t$, where $Size_{\mathfrak{T}}$ is given by

$$\begin{aligned}
&Size_{\star}\ t = t \to \mathit{Int}\ ;\\
&Size_{\mathfrak{T} \to \mathfrak{U}}\ t = \forall x\ .\ Size_{\mathfrak{T}}\ x \to Size_{\mathfrak{U}}\ (t\ x)\ .
\end{aligned}$$

The *size* function itself is defined by giving cases for each of the primitive types, so that we have

$$\begin{aligned}
&\mathit{size}_{t::\mathfrak{T}} && :: \mathit{Size}_{\mathfrak{T}}\ t\\
&\mathit{size}_{\mathit{Int}}\ i && = 0\\
&\mathit{size}_{1}\ () && = 0\\
&\mathit{size}_{+}\ \mathit{size}_{a}\ \mathit{size}_{b}\ (\mathit{Inl}\ a) && = \mathit{size}_{a}\ a\\
&\mathit{size}_{+}\ \mathit{size}_{a}\ \mathit{size}_{b}\ (\mathit{Inr}\ b) && = \mathit{size}_{b}\ b\\
&\mathit{size}_{\times}\ \mathit{size}_{a}\ \mathit{size}_{b}\ (a, b) && = \mathit{size}_{a}\ a + \mathit{size}_{b}\ b\ \ .
\end{aligned}$$

To summarise, a generic function possesses a kind-indexed type and is defined by providing instances for the type constants of GH.

3. The Λ-calculus

The central idea of generic programming is to interpret a type by a function, the so-called instance of a generic function at that type. Different approaches to generic programming differ in the language that is used to represent types [13]. PolyP [15], for instance, is based on a grammar for bifunctors and regular functors. Generic Haskell uses the simply-typed lambda calculus to model Haskell's expressive type system. The latter choice is particularly attractive as it covers a large class of types. Furthermore, the simply-typed lambda calculus can be interpreted in a cartesian closed category, which is key to the categorical treatment of Generic Haskell.

The rest of the section is structured as follows. We first revise syntax and semantics of the simply-typed lambda calculus (Section 3.1). Next a category suitable for interpreting lambda terms as functors (Section 3.2) is introduced. We then provide some background to Mendler-style folds and unfolds (Section 3.3) before specialising the interpretation of lambda terms to this category (Section 3.4).

3.1 A categorical model of the simply-typed lambda calculus

We assume a syntactic category of type constants b and a syntactic category of term constants c. The following development is parametric in this data.

The raw syntax of the lambda calculus is given below.

$$\begin{aligned}
t &\ ::=\ \ b \mid t_1 \to t_2 \mid t_1 \times t_2\\
e &\ ::=\ \ c \mid x \mid \lambda\, x : t\ .\ e \mid e_2\ e_1 \mid (e_1, e_2) \mid \mathit{fst}\ e \mid \mathit{snd}\ e
\end{aligned}$$

We have added products to the language; they are required anyway and they are jolly useful in modelling mutual recursion. For reasons of space, we omit the typing rules that identify proved lambda terms among the raw terms—they are entirely standard [5].

Turning to the semantics, let $\mathscr{C}$ be a cartesian closed category. Types are interpreted as objects in $\mathscr{C}$, and terms are interpreted as arrows. Cartesian closure requires the existence of a final object (1, !), products ($A \times B$, outl, outr, $f \vartriangle g$), and exponentials (B^A, apply, curry). An interpretation $\mathscr{I}$ is fixed by assigning objects to the type constants, $\mathscr{I}_b$, and so-called *elements*, arrows of type $\mathscr{C}(1, A)$, to the term constants, $\mathscr{I}_c$. Figure 1 lists the semantic equations. The semantics of a proved term is defined by induction over its typing derivation: $\llbracket \Gamma \vdash e : t \rrbracket : \mathscr{C}(\llbracket \Gamma \rrbracket, \llbracket t \rrbracket)$. In words, the interpretation of a term is an arrow from the interpretation of the context to the interpretation of its type. Types are interpreted in the obvious way, $\llbracket t \rrbracket : \mathscr{C}$; the interpretation of a context, $\llbracket \Gamma \rrbracket : \mathscr{C}$, is a 'run-time environment', a nested product. If $e : t$ is closed, then its interpretation $\llbracket e : t \rrbracket$ is an element of $\llbracket t \rrbracket$, an arrow of type $\mathscr{C}(1, \llbracket t \rrbracket)$.

3.2 Cartesian closure of $\mathbf{Cat}$

In order to apply the framework to the specialisation of generic functions, we have to exhibit a suitable category that allows us to interpret terms as functors. Functors are arrows in $\mathbf{Cat}$, the category of all small categories. All that is left to do is to demonstrate that $\mathbf{Cat}$ is cartesian closed. This is a known fact [17, p. 98], but it is instructive to work through the exercise. Moreover, since our goal is to calculate the definition of a functor including its arrow part from a type term, we have a vital interest in the details.

$$\begin{aligned}
\llbracket b \rrbracket &= \mathscr{I}_b & \llbracket () \rrbracket &= 1\\
\llbracket t_1 \to t_2 \rrbracket &= \llbracket t_2 \rrbracket^{\llbracket t_1 \rrbracket} & \llbracket \Gamma, x : t \rrbracket &= \llbracket \Gamma \rrbracket \times \llbracket t \rrbracket\\
\llbracket t_1 \times t_2 \rrbracket &= \llbracket t_1 \rrbracket \times \llbracket t_2 \rrbracket
\end{aligned}$$

$$\begin{aligned}
&\llbracket \Gamma \vdash c : t \rrbracket && = \mathscr{I}_c \cdot\ !\\
&\llbracket \Gamma, x : t \vdash x : t \rrbracket && = \mathit{outr}\\
&\llbracket \Gamma, y : t \vdash x : t \rrbracket && = \llbracket \Gamma \vdash x : t \rrbracket \cdot \mathit{outl}\\
&\llbracket \Gamma \vdash \lambda\, x : t_1\ .\ e_2 : t_1 \to t_2 \rrbracket && = \mathit{curry}\ \llbracket \Gamma, x : t_1 \vdash e_2 : t_2 \rrbracket\\
&\llbracket \Gamma \vdash e_2\ e_1 : t_2 \rrbracket && =\\
&\qquad \mathit{apply} \cdot (\llbracket \Gamma \vdash e_2 : t_1 \to t_2 \rrbracket \vartriangle \llbracket \Gamma \vdash e_1 : t_1 \rrbracket)\\
&\llbracket \Gamma \vdash (e_1, e_2) : t_1 \times t_2 \rrbracket && = \llbracket \Gamma \vdash e_1 : t_1 \rrbracket \vartriangle \llbracket \Gamma \vdash e_2 : t_2 \rrbracket\\
&\llbracket \Gamma \vdash \mathit{fst}\ e : t_1 \rrbracket && = \mathit{outl} \cdot \llbracket \Gamma \vdash e : t_1 \times t_2 \rrbracket\\
&\llbracket \Gamma \vdash \mathit{snd}\ e : t_2 \rrbracket && = \mathit{outr} \cdot \llbracket \Gamma \vdash e : t_1 \times t_2 \rrbracket
\end{aligned}$$

Figure 1. Semantics of types, contexts and terms.

The final object in $\mathbf{Cat}$ is $\mathbf{1}$, the category that consists of a single object $*$ and a single arrow, the identity id_*. The functor $! : \mathscr{C} \to \mathbf{1}$ is defined $!\ A = *$ and $!\ f = id_*$. Finality means that ! is the unique functor of this type.

The product category $\mathscr{C} \times \mathscr{D}$ is the product in $\mathbf{Cat}$. An object of $\mathscr{C}_1 \times \mathscr{C}_2$ is a pair (A_1, A_2) of objects $A_1 : \mathscr{C}_1$ and $A_2 : \mathscr{C}_2$; an arrow of $(\mathscr{C}_1 \times \mathscr{C}_2)((A_1, A_2), (B_1, B_2))$ is a pair (f_1, f_2) of arrows $f_1 : \mathscr{C}_1(A_1, B_1)$ and $f_2 : \mathscr{C}_2(A_2, B_2)$. Identity and composition are defined component-wise: $id = (id, id)$ and $(f_1, f_2) \cdot (g_1, g_2) = (f_1 \cdot g_1, f_2 \cdot g_2)$. Like for object-level products, we have two projection functors $\mathsf{Outl} : \mathscr{C} \times \mathscr{D} \to \mathscr{C}$ and $\mathsf{Outr} : \mathscr{C} \times \mathscr{D} \to \mathscr{D}$ given by

$$\begin{aligned}
\mathsf{Outl}\,(A, B) &= A\ , & \mathsf{Outr}\,(A, B) &= B\ ,\\
\mathsf{Outl}\,(f, g) &= f\ , & \mathsf{Outr}\,(f, g) &= g\ .
\end{aligned}$$

Let $\mathsf{F} : \mathscr{C} \to \mathscr{D}$ and $\mathsf{G} : \mathscr{C} \to \mathscr{E}$ be two functors with a common source, the 'split' functor $\mathsf{F} \vartriangle \mathsf{G} : \mathscr{C} \to \mathscr{D} \times \mathscr{E}$ is defined

$$\begin{aligned}
(\mathsf{F} \vartriangle \mathsf{G})\, A &= (\mathsf{F}\, A, \mathsf{G}\, A)\ ,\\
(\mathsf{F} \vartriangle \mathsf{G})\, f &= (\mathsf{F}\, f, \mathsf{G}\, f)\ .
\end{aligned}$$

It is not hard to see that the action on arrows preserves identity and composition. The split functor enjoys the universal property

$$\mathsf{H} = \mathsf{F} \vartriangle \mathsf{G} \qquad \Longleftrightarrow \qquad \mathsf{Outl} \cdot \mathsf{H} = \mathsf{F}\ \wedge\ \mathsf{Outr} \cdot \mathsf{H} = \mathsf{G}\ ,$$

which states that the product category is indeed a product in $\mathbf{Cat}$.

We now turn to the exponentials in $\mathbf{Cat}$, the category $\mathscr{D}^{\mathscr{C}}$ of functors and natural transformations. The application functor $\mathsf{Apply} : \mathscr{D}^{\mathscr{C}} \times \mathscr{C} \to \mathscr{D}$ is given by

$$\begin{aligned}
\mathsf{Apply}\,(\mathsf{F}, A) &= \mathsf{F}\, A\ ,\\
\mathsf{Apply}\,(\alpha, f) &= \mathsf{G}\, f \cdot \alpha\, A = \alpha\, B \cdot \mathsf{F}\, f\ .
\end{aligned}$$

The action on objects is simply the application of the functor. The action on an arrow (α, f), where $\alpha : \mathsf{F} \stackrel{.}{\to} \mathsf{G}$ and $f : A \to B$, can be given two equivalent definitions, $\mathsf{G}\, f \cdot \alpha\, A = \alpha\, B \cdot \mathsf{F}\, f$, which fall out of the naturality condition on $\alpha : \mathsf{F} \to \mathsf{G}$. Of course, we have to make sure that Apply preserves identity and composition.

Turning to the definition of currying, we first introduce the concept of a partially applied functor. Let $\mathsf{F} : \mathscr{C} \times \mathscr{D} \to \mathscr{E}$ and let $A : \mathscr{C}$, define $\mathsf{F}_A : \mathscr{D} \to \mathscr{E}$ by $\mathsf{F}_A\, B = \mathsf{F}\,(A, B)$ and $\mathsf{F}_A\, g = \mathsf{F}\,(id_A, g)$. Again, it is not hard to see that F_A is a functor. Using partial application, we define $\mathsf{Curry}\ \mathsf{F} : \mathscr{C} \to \mathscr{E}^{\mathscr{D}}$ by

$$\begin{aligned}
\mathsf{Curry}\ \mathsf{F}\, A &= \mathsf{F}_A\ ,\\
\mathsf{Curry}\ \mathsf{F}\, f &= \lambda\, g\ .\ \mathsf{F}\,(f, g)\ .
\end{aligned}$$

The action on arrows sends an arrow $f : \mathscr{C}(A, B)$ to $\mathsf{Curry}\ \mathsf{F}\ f : \mathsf{Curry}\,\mathsf{F}\,A \to \mathsf{Curry}\,\mathsf{F}\,B$. This time it is probably not immediate that $\mathsf{Curry}\ \mathsf{F}$ is a functor, so the reader is encouraged to work through the details. Currying satisfies the universal property

$$\mathsf{G} = \mathsf{Curry}\ \mathsf{F} \qquad \Longleftrightarrow \qquad \mathsf{Apply} \cdot (\mathsf{G} \times \mathsf{Id}) = \mathsf{F} \ ,$$

which states that $\mathscr{D}^{\mathscr{C}}$ is the exponential in $\mathbf{Cat}$.

3.3 Mendler-style folds and unfolds

Generic Haskell treats recursion implicitly: recursion on the type level is mapped to recursion on the value level. Since we are aiming for a categorical foundation of GH, we have to make type recursion explicit. In a categorical setting, inductive datatypes are modelled by initial algebras and coinductive datatypes by final coalgebras. Let $\mathsf{F} : \mathscr{C} \to \mathscr{C}$ be a functor, we denote the initial F-algebra by $(\mu\mathsf{F}, \mathit{in})$ and the final F-coalgebra by $(\nu\mathsf{F}, \mathit{out})$. For instance, in $\mathbf{Set}$, $\mu\mathsf{L}$ with $\mathsf{L}\ A = 1 + \mathbb{N} \times A$ is the type of finite lists of natural numbers, while $\nu\mathsf{L}$ is the type of colists, which comprises both finite and infinite lists. In $\mathbf{Cpo}_{\perp}$, initial algebras and final coalgebras coincide—this is why GH is able to treat recursion uniformly.

Traditionally, functions from an initial algebra are given by folds (aka catamorphisms) and functions to a final coalgebra are given by unfolds (aka anamorphisms). We deviate from standard practise and use Mendler-style folds and unfolds [21] instead since they blend more nicely with GH. Informally, Mendler-style folds capture the idea that the semantics of a recursion equation is given by the fixed point of its associated base function. As an example, consider the function $\mathit{sum} : \mu\mathsf{L} \to \mathbb{N}$, which sums a list of natural numbers. Written in a point-free style, sum is given by the recursion equation

$$\mathit{sum} \cdot \mathit{in} = \mathit{zero} \mathbin{\triangledown} \mathit{plus} \cdot (\mathit{id} \times \mathit{sum}) \ ,$$

where $\mathit{zero} : 1 \to \mathbb{N}$ corresponds to 0 and $\mathit{plus} : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ is addition. We obtain the base function by turning the right-hand side into a function in the variable sum. Applying the Mendler-style fold to the result, $\llparenthesis \lambda\, \mathit{sum}\ .\ \mathit{zero} \mathbin{\triangledown} \mathit{plus} \cdot (\mathit{id} \times \mathit{sum}) \rrparenthesis$, then yields the *unique* solution of the recursion equation.

Formally, let Ψ be a *base function* that sends an arrow $f : \mathscr{C}(A, B)$ to an arrow $\Psi f : \mathscr{C}(\mathsf{F} A, B)$ such that $\Psi(f \cdot h) = \Psi f \cdot \mathsf{F} h$. The side condition formalises that Ψ is natural in A:

$$\Psi : \forall X\ .\ \mathscr{C}(X, B) \to \mathscr{C}(\mathsf{F}\ X, B) \ .$$

The *Mendler-style fold* $\llparenthesis \Psi \rrparenthesis : \mathscr{C}(\mu\mathsf{F}, B)$ is then characterised by the *uniqueness property* (UP)

$$h = \llparenthesis \Psi \rrparenthesis \qquad \Longleftrightarrow \qquad h \cdot \mathit{in} = \Psi\ h \ . \tag{3.1}$$

Substituting the left-hand side into the right-hand side gives the *computation law*:

$$\llparenthesis \Psi \rrparenthesis \cdot \mathit{in} = \Psi\ \llparenthesis \Psi \rrparenthesis \ ,$$

which can be seen as the defining equation of $\llparenthesis \Psi \rrparenthesis$. The UP states that $\llparenthesis \Psi \rrparenthesis$ is the *unique* solution of this equation. The computation law has a straightforward operational reading. The argument of $\Psi\ \llparenthesis \Psi \rrparenthesis$ is destructed—this can be seen more easily if we move the isomorphism $\mathit{in} : \mathscr{C}(\mathsf{F}\ (\mu\mathsf{F}), \mu\mathsf{F})$ to the right: $\llparenthesis \Psi \rrparenthesis = \Psi\ \llparenthesis \Psi \rrparenthesis \cdot \mathit{in}^{\circ}$. Thus, $\Psi\,\llparenthesis \Psi \rrparenthesis$ takes an argument of type $\mathsf{F}(\mu\mathsf{F})$. The base function Ψ then works on the F-structure, possibly applying its argument $\llparenthesis \Psi \rrparenthesis$ to recursive substructures of type $\mu\mathsf{F}$. The naturality of Ψ ensures that the substructures can *only* be passed to the recursive calls.

The UP (3.1) has three other consequences that are worth singling out. Setting $\Psi := \lambda f\ .\ \mathit{in} \cdot \mathsf{F} f$ and $h = \mathit{id}$ yields the *reflection law*:

$$\llparenthesis \lambda f\ .\ \mathit{in} \cdot \mathsf{F}\, f \rrparenthesis = \mathit{id} \ . \tag{3.2}$$

The most important consequence is the *fusion law*:

$$h \cdot \llparenthesis \Phi \rrparenthesis = \llparenthesis \Psi \rrparenthesis \quad \Longleftarrow \quad h \cdot \Phi\, f = \Psi\ (h \cdot f) \ , \tag{3.3}$$

which states a condition for fusing an application of a function with a fold to form another fold.

Finally, the type constructor μ can be turned into a higher-order functor of type $\mathscr{C}^{\mathscr{C}} \to \mathscr{C}$. The object part of this functor maps a functor to its initial algebra. The arrow part, which maps a natural transformation $\alpha : \mathsf{F} \mathrel{\dot{\to}} \mathsf{G}$ to an arrow $\mu\alpha : \mathscr{C}(\mu\mathsf{F}, \mu\mathsf{G})$, is given by:

$$\mu\alpha = \llparenthesis \lambda\, f\ .\ \mathit{in} \cdot \alpha\, f \rrparenthesis \ ,$$

where $\alpha\ f$ is shorthand for $\mathsf{Apply}\ (\alpha, f)$, the arrow part of the application functor. To establish functoriality, we have to show that $\mu\mathit{id}_{\mathsf{F}} = \mathit{id}_{\mu\mathsf{F}}$ and $\mu(\beta \cdot \alpha) = \mu\beta \cdot \mu\alpha$. That μ preserves identity is an immediate consequence of reflection. Preservation of composition is a consequence of the *functor fusion law*:

$$\llparenthesis \Psi \cdot \alpha \rrparenthesis = \llparenthesis \Psi \rrparenthesis \cdot \mu\alpha \ . \tag{3.4}$$

Functor fusion expresses that $\llparenthesis - \rrparenthesis$ is natural in F:

$$\llparenthesis - \rrparenthesis : \forall \mathsf{F}\ .\ (\forall X\ .\ \mathscr{C}(X, B) \to \mathscr{C}(\mathsf{F}\ X, B)) \to \mathscr{C}(\mu\mathsf{F}, B) \ .$$

This is a *higher-order naturality* property [8] as F is a functor. Using GH's kind-indexed types the signature can be written more succinctly as $\llparenthesis - \rrparenthesis : \mathit{Fold}_{(\star \to \star) \to \star}\ \mu$, where $\mathit{Fold}_{\star}\ X = X \to B$.

Using μ we can express that $\mathit{in} : \mathsf{F}\ (\mu\mathsf{F}) \to \mu\mathsf{F}$ is natural in F:

$$\mu\alpha \cdot \mathit{in} = \mathit{in} \cdot \alpha\ (\mu\alpha) \ . \tag{3.5}$$

As an aside, Mendler-style folds and traditional folds are in one-to-one correspondence. The proof makes use of the so-called Yoneda Lemma. Very briefly, let $\mathsf{H} : \mathscr{C} \to \mathbf{Set}$ be a $\mathbf{Set}$-valued functor, and let $A : \mathscr{C}$ be an object, then

$$\mathsf{H}\ A \cong \mathscr{C}(A, -) \mathrel{\dot{\to}} \mathsf{H} \ .$$

Instantiating H to $\mathscr{D}(\mathsf{F}\ -, Y) : \mathscr{C}^{\mathsf{op}} \to \mathbf{Set}$, we have

$$\begin{aligned} & \mathsf{H}\ Y \cong \mathscr{C}^{\mathsf{op}}(Y, -) \mathrel{\dot{\to}} \mathsf{H} \\ \Longleftrightarrow\ & \{\ \text{definition of } \mathsf{H}\ \} \\ & \mathscr{D}\ (\mathsf{F}\ Y, Y) \cong \forall A\ .\ \mathscr{C}(A, Y) \to \mathscr{D}(\mathsf{F}\ A, Y) \end{aligned}$$

If the algebra a and the base function Ψ are related by the isomorphism, then the traditional fold $\llparenthesis a \rrparenthesis$ and the Mendler-style fold $\llparenthesis \Psi \rrparenthesis$ are, in fact, equal [11]. The Yoneda Lemma is worthwhile memorising as we shall find several uses for it.

The development above nicely dualises to final coalgebras and unfolds. Let $\Psi : \forall Y\ .\ \mathscr{C}(A, Y) \to \mathscr{C}(A, \mathsf{G}\ Y)$ be a base function. The *Mendler-style unfold* $[\!(\Psi)\!] : \mathscr{C}(A, \nu\mathsf{G})$ is then characterised by the *uniqueness property*

$$h = [\!(\Psi)\!] \qquad \Longleftrightarrow \qquad \mathit{out} \cdot h = \Psi\ h \ . \tag{3.6}$$

3.4 The interpretation of lambda terms in $\mathbf{Cat}$

In Section 3.2 we have set up the general framework. To fill it with life we have to populate the syntactic categories b and c. The particulars depend on the generic language and the generic program at hand—not every function makes sense for every collection of type constructors. Re-using the semantic symbols for the syntactic entities, a fairly complete set of constants is

$$\begin{aligned} b\ &::=\ \star \ , \\ c\ &::=\ \mathit{Int} \mid 0 \mid 1 \mid + \mid \times \mid \mu \mid \nu \ . \end{aligned}$$

Since we interpret lambda terms as functors, type terms become kind terms and terms become type terms. The syntactic category b comprises only a single element: the kind $\star$ represents the type of types. This choice is influenced by Haskell's type system, which only has one base kind. The syntactic category c features constants

$$\begin{aligned}
\llbracket b \rrbracket &= \mathscr{I}_b & \llbracket () \rrbracket &= \mathbf{1}\\
\llbracket t_1 \to t_2 \rrbracket &= \llbracket t_2 \rrbracket^{\llbracket t_1 \rrbracket} & \llbracket \Gamma, x : t \rrbracket &= \llbracket \Gamma \rrbracket \times \llbracket t \rrbracket\\
\llbracket t_1 \times t_2 \rrbracket &= \llbracket t_1 \rrbracket \times \llbracket t_2 \rrbracket
\end{aligned}$$

$$\begin{aligned}
\llbracket \Gamma, x : t \vdash x : t \rrbracket\, \varrho &= \mathsf{Outr}\ \varrho\\
\llbracket \Gamma, y : t \vdash x : t \rrbracket\, \varrho &= \llbracket \Gamma \vdash x : t \rrbracket\, (\mathsf{Outl}\ \varrho)\\
\llbracket \Gamma \vdash \lambda\, x : t_1 \,.\, e_2 : t_1 \to t_2 \rrbracket\, \varrho &= \Lambda\, \mathsf{F} \,.\, \llbracket \Gamma, x : t_1 \vdash e_2 : t_2 \rrbracket\, (\varrho, \mathsf{F})\\
\llbracket \Gamma \vdash e_2\ e_1 : t_2 \rrbracket\, \varrho &=\\
&\quad (\llbracket \Gamma \vdash e_2 : t_1 \to t_2 \rrbracket\, \varrho)\, (\llbracket \Gamma \vdash e_1 : t_1 \rrbracket\, \varrho)\\
\llbracket \Gamma \vdash (e_1, e_2) : t_1 \times t_2 \rrbracket\, \varrho &= (\llbracket \Gamma \vdash e_1 : t_1 \rrbracket\, \varrho, \llbracket \Gamma \vdash e_2 : t_2 \rrbracket\, \varrho)\\
\llbracket \Gamma \vdash \mathit{fst}\ e : t_1 \rrbracket\, \varrho &= \mathsf{Outl}\ (\llbracket \Gamma \vdash e : t_1 \times t_2 \rrbracket\, \varrho)\\
\llbracket \Gamma \vdash \mathit{snd}\ e : t_2 \rrbracket\, \varrho &= \mathsf{Outr}\ (\llbracket \Gamma \vdash e : t_1 \times t_2 \rrbracket\, \varrho)
\end{aligned}$$

Figure 2. The categorical semantics specialised to $\mathbf{Cat}$.

for integers (representative for primitive types), initial objects, final objects, coproducts, products, initial algebras and final coalgebras. The kinds of these constants are fixed as:[1]

$$\begin{aligned}
\mathit{Int}, 0, 1 &\ :\ \star\ ,\\
+, \times &\ :\ \star \times \star \to \star\ ,\\
\mu, \nu &\ :\ (\star \to \star) \to \star\ .
\end{aligned}$$

Note that the fixed-point operators μ and ν are restricted to types of kind $\star$, that is, we cannot define nested datatypes [1]. We do not foresee any problems in extending fixed-points to higher types, but for now we have left this to future work. Also, the list does not include exponentials, simply because the kind system is too weak: we cannot express that $(=)^{-} : \mathscr{C}^{\mathsf{op}} \times \mathscr{C} \to \mathscr{C}$ is contravariant in its first argument. A suitable extension is again left to future work.

Turning to the semantics, we have to interpret the kind constant $\star$ by a base category $\mathscr{C}$ and the type constants by functors over $\mathscr{C}$. Naturally, the two choices go hand in hand. In particular, if μ is meant to denote the initial algebra functor, then we have to restrict $\mathscr{C}$ to a cocomplete category and we have to make sure that cocompleteness is preserved by the constructions. Furthermore, we have to ensure that the definable functors are cocontinuous. Likewise, for ν we require completeness and continuity. If $\mathscr{C}$ is cocomplete and complete, then the other conditions are met—the details are beyond the scope of this paper. Finally, let us point out that there is no need to map, say, $+$ to a coproduct. The semantic entities only have to be functorial. For instance, the category $\mathbf{Cpo}$ of complete partial orders and continuous functions has no coproducts, so we have to interpret $+$ by the coalesced sum or by the separated sum.

It is high time to look at examples, deriving mapping functions for types of interest. Haskell like many other languages maintains a strict phase distinction. Types are compile-time entities, so we can safely assume that we only need to specialise closed type terms—Haskell makes the same assumption for its **deriving** mechanism. In Figure 2 we have specialised the categorical semantics to $\mathbf{Cat}$, making the environment explicit. The equations are easy to memorise: pairing is interpreted by pairing, application by application, and abstraction by abstraction. Now, specialising the list datatype $\mathsf{List} = \Lambda\, A \,.\, \mu(\Lambda\, B \,.\, 1 + A \times B)$ yields the mapping function

$$\lambda\, f \,.\, \llparenthesis \lambda\, g \,.\, \mathit{in} \cdot (\mathit{id}_1 + f \times g) \rrparenthesis\ .$$

We have plugged in the definitions of the type constants, in particular, $\mu\alpha = \llparenthesis \lambda f \,.\, \mathit{in} \cdot \alpha f \rrparenthesis$. For rose trees $\mathsf{Rose} = \Lambda\, A \,.\, \mu(\Lambda B \,.\, A \times \mathsf{List}\ B)$, we obtain

$$\lambda\, f \,.\, \llparenthesis \lambda\, g \,.\, \mathit{in} \cdot (f \times \mathsf{List}\ g) \rrparenthesis\ .$$

The node of a rose tree has a list of sub-trees. We can generalise the construction, if we parametrise Rose by a 'sub-tree functor': $\mathsf{GRose} = \Lambda\, \mathsf{F} \,.\, \Lambda\, A \,.\, \mu(\Lambda\, B \,.\, A \times \mathsf{F}\ B)$. The functor GRose is truly higher-order: it takes a functor to functor—in Haskell jargon, it has kind $(\star \to \star) \to (\star \to \star)$. Nonetheless, its mapping function is straightforward to determine:

$$\lambda\, \alpha \,.\, \lambda\, f \,.\, \llparenthesis \lambda\, g \,.\, \mathit{in} \cdot (f \times \alpha\ g) \rrparenthesis\ .$$

Like the type, it simply abstracts away from List.

4. Simple generic consumers

In the previous section we have seen that a lambda term can be interpreted as a functor. The functorial action on arrows corresponds to Generic Haskell's mapping function. In this section, we show how to capture simple generic consumers such as generic size or crush [19]. Perhaps surprisingly, the changes are minor: the kind $\star$ is interpreted by a different category, one that has more structure, and, as a consequence, the interpretation of the type constants has to be adapted. The one-million-dollar question is, of course, what constitutes a suitable base category. We argue that a so-called slice category fits the bill, so the interpretation of $\star$ is defined

$$\mathscr{G}_\star = \mathscr{C} \downarrow Y\ ,$$

where $\mathscr{G}$ stands for a generic interpretation. Before we adapt the interpretation of the type constants, we have to introduce slice categories first and this is what we do after a short interlude.

4.1 Recap: Generic size and crush

A *simple* generic consumer is a function of type $A \to Y$, where A is the type index or generic type and Y is some fixed type. The generic size function is the paradigmatic example of a consumer. For size, the type Y is instantiated to the type of natural numbers $\mathbb{N}$. In Section 2 we have seen the Generic Haskell version of size. Written using categorical combinators, it takes the following form:

$$\begin{aligned}
\mathit{size}_{\mathit{Int}} &= \mathit{zero}\\
\mathit{size}_0 &= \text{¡}\\
\mathit{size}_1 &= \mathit{zero}\\
\mathit{size}_+ &= \lambda\, (f, g) \,.\, f \triangledown g\\
\mathit{size}_\times &= \lambda\, (f, g) \,.\, \mathit{plus} \cdot (f \times g)\\
\mathit{size}_\mu &= \lambda\, \gamma \,.\, \llparenthesis \gamma \rrparenthesis\ .
\end{aligned}$$

Taking the size cannot sensibly be defined for final coalgebras, which is why the case for ν is missing. The definitions for 0, $+$ and μ (the colimits) are "for free" in a sense we shall make precise later. For now we just note that the instances are just the mediating arrows for these types (¡, $- \triangledown =$, $\llparenthesis - \rrparenthesis$).

So the definition of size has only two specific cases: 1 and $\times$. These are determined by the constant $\mathit{zero} : 1 \to \mathbb{N}$ and the operation $\mathit{plus} : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$. The generic function crush abstracts away from these two ingredients. Given a constant $e : 1 \to Y$ and a binary operation $\mathit{op} : Y \times Y \to Y$, it is defined

$$\begin{aligned}
\mathit{crush}_{\mathit{Int}} &= e\\
\mathit{crush}_0 &= \text{¡}\\
\mathit{crush}_1 &= e\\
\mathit{crush}_+ &= \lambda\, (f, g) \,.\, f \triangledown g\\
\mathit{crush}_\times &= \lambda\, (f, g) \,.\, \mathit{op} \cdot (f \times g)\\
\mathit{crush}_\mu &= \lambda\, \gamma \,.\, \llparenthesis \gamma \rrparenthesis\ .
\end{aligned}$$

We shall use crush as a running example. Indeed, the following can be seen as a logical reconstruction of this definition.

4.2 Slice category

How can we model size or crush in our framework? We need a way to associate an arrow with an object: crush_0 with 0, crush_1

---

[1] Although Haskell has no product kinds, we introduce them here since they are convenient in our definitions: recovering kinds that can be used in Haskell can be achieved through currying.

with 1, and so forth. This is exactly what a so-called slice category allows us to do.

Let $\mathscr{C}$ be a category and let $Y : \mathscr{C}$ be an object of $\mathscr{C}$. An object of the *slice category* $\mathscr{C} \downarrow Y$ is a pair (A, a) where $A : \mathscr{C}$ is an object and $a : A \to Y : \mathscr{C}$ is an arrow. An arrow $f : (A, a) \to (B, b) : \mathscr{C} \downarrow Y$ of the slice category is an arrow $f : A \to B : \mathscr{C}$ of the underlying category such that $a = b \cdot f$.

$$\begin{tikzcd}
A \arrow[d, "a"'] & A \arrow[r, "f"] \arrow[dr, "a"'] & B \arrow[d, "b"] & B \arrow[d, "b"] \\
Y & & Y & Y
\end{tikzcd}$$

In short, objects are arrows and arrows are commuting triangles. Identity and composition are inherited from the base category $\mathscr{C}$. Clearly, id_A serves as the identity on (A, a) as $a = a \cdot id_A$. The diagram below shows that composition takes commuting triangles to commuting triangles: $b = c \cdot g$ and $a = b \cdot f$ imply $a = c \cdot g \cdot f$.

$$\begin{tikzcd}
A \arrow[r, "id_A"] \arrow[dr, "a"'] & A \arrow[d, "a"] & A \arrow[r, "f"] \arrow[dr, "a"'] & B \arrow[r, "g"] \arrow[d, "b"] & C \arrow[dl, "c"] \\
& Y & & Y &
\end{tikzcd}$$

We can easily turn the instances of *crush* at base types into objects of the slice category $\mathscr{C} \downarrow \mathbb{N}$—assuming that $\mathbb{N}$ lives in $\mathscr{C}$. The type index is the object part, and the generic instance at that type is the arrow part: $(0, crush_0)$, $(1, crush_1)$ etc. Given $h : (A, crush_A) \to (B, crush_B)$, the arrows of $\mathscr{C} \downarrow \mathbb{N}$ satisfy the following fusion property: $crush_B = crush_A \cdot h$.

A slice category adds structure on top of a base category. In such a situation, there is a functor that forgets about the extra structure. The *forgetful* or *underlying functor* $\mathsf{U}_Y : \mathscr{C} \downarrow Y \to \mathscr{C}$ forgets about the base object Y and the arrows into Y:

$$\begin{aligned}
\mathsf{U}_Y\ (A, a) &= A \ , \\
\mathsf{U}_Y\ f &= f \ .
\end{aligned}$$

We shall also need an operation that extracts the arrow component from an object.

$$\alpha\ (A, a) = a \tag{4.1}$$

Since α maps an object to an arrow it is actually a transformation of type $\alpha\ (A, a) : (A, a) \to (Y, id_Y)$. It is furthermore natural in (A, a) as a quick calculation shows. Let $h : (A, a) \to (B, b)$, then

$$\begin{aligned}
& \alpha\ (A, a) = \alpha\ (B, b) \cdot h \\
\iff & \quad \{ \text{ definition of } \alpha \ \} \\
& a = b \cdot h \\
\iff & \quad \{ \text{ assumption: } h : (A, a) \to (B, b) \ \} \\
& true \ .
\end{aligned}$$

Let us now investigate the structure of slice categories more closely. This will pay considerable dividends later when we discuss the interpretation of the type constants. To this end we shall use the categorical concept of an adjunction. For a calculational introduction to adjunctions we refer the interested reader to the paper "Adjunctions" [7].

If the category $\mathscr{C}$ has products, then the forgetful functor U_Y has a right adjoint, the so-called *pairing functor* $\mathsf{P}_Y : \mathscr{C} \to \mathscr{C} \downarrow Y$.

$$\begin{tikzcd}
\mathscr{C} \arrow[r, bend right, "\mathsf{P}_Y"'] \arrow[r, phantom, "\perp"] & \mathscr{C} \downarrow Y \arrow[l, bend right, "\mathsf{U}_Y"']
\end{tikzcd}$$

The pairing functor is defined

$$\begin{aligned}
\mathsf{P}_Y\ A &= (A \times Y, outr) \ , \\
\mathsf{P}_Y\ f &= f \times Y \ .
\end{aligned}$$

The functor P_Y pairs its argument with Y, hence its name. It respects the types, $\mathsf{P}_Y\ f : \mathsf{P}_Y\ A \to \mathsf{P}_Y\ B$, as $outr = outr \cdot (f \times Y)$. To establish the adjunction we have to show that certain arrows in $\mathscr{C}$ are in one-to-one correspondence with certain arrows in $\mathscr{C} \downarrow Y$:

$$\mathscr{C}(\mathsf{U}_Y\ (A, a), B) \cong (\mathscr{C} \downarrow Y)((A, a), \mathsf{P}_Y\ B) \ .$$

Intuitively the adjunction captures the idea of *caching*: an attribute $a : A \to Y$ is cached by pairing B with a's value. The adjuncts make this explicit

$$\begin{aligned}
\lfloor f : \mathsf{U}_Y\ (A, a) \to B \rfloor &= f \vartriangle a \ , \\
\lceil g : (A, a) \to \mathsf{P}_Y\ B \rceil &= outl \cdot g \ .
\end{aligned}$$

The left adjunct respects the types, $\lfloor f \rfloor : (A, a) \to \mathsf{P}_Y\ B$, as $a = outr \cdot (f \vartriangle a)$. The following calculations show that $\lfloor - \rfloor$ and $\lceil - \rceil$ are indeed inverses.

$$\begin{aligned}
& \lceil \lfloor f \rfloor \rceil \\
= & \quad \{ \text{ definitions } \} \\
& outl \cdot (f \vartriangle a) \\
= & \quad \{ \text{ computation (B.3a) } \} \\
& f
\end{aligned}$$

$$\begin{aligned}
& \lfloor \lceil g \rceil \rfloor \\
= & \quad \{ \text{ definitions } \} \\
& outl \cdot g \vartriangle a \\
= & \quad \{ \text{ assumption: } a = outr \cdot g \ \} \\
& outl \cdot g \vartriangle outr \cdot g \\
= & \quad \{ \text{ fusion (B.5) } \} \\
& (outl \vartriangle outr) \cdot g \\
= & \quad \{ \text{ reflection (B.2) } \} \\
& g
\end{aligned}$$

Adjunctions come with a wealth of properties. One important fact to remember is that left adjoints preserve colimits and right adjoints preserve limits. We shall repeatedly make use of these facts.

In the previous section we have interpreted a type of kind $\star \to \star$ as a functor $\mathsf{F} : \mathscr{C} \to \mathscr{C}$. Now that the base category is $\mathscr{C} \downarrow Y$ we interpret the type as a functor $\bar{\mathsf{F}} : \mathscr{C} \downarrow Y \to \mathscr{C} \downarrow Y$. Of course, the two interpretations should be related. If we forget about the base object Y and the arrows into Y, that is the generic instances, then $\bar{\mathsf{F}}$ should behave like F. This idea is formally captured using the notion of lifting.

4.3 Lifting

A functor $\bar{\mathsf{F}} : \mathscr{C} \downarrow Y \to \mathscr{D} \downarrow Z$ is a *lifting of* $\mathsf{F} : \mathscr{C} \to \mathscr{D}$ if $\mathsf{U} \circ \bar{\mathsf{F}} = \mathsf{F} \circ \mathsf{U}$.

$$\begin{tikzcd}
\mathscr{C} \downarrow Y \arrow[r, "\bar{\mathsf{F}}"] \arrow[d, "\mathsf{U}"'] & \mathscr{D} \downarrow Z \arrow[d, "\mathsf{U}"] \\
\mathscr{C} \arrow[r, "\mathsf{F}"'] & \mathscr{D}
\end{tikzcd}$$

Liftings of F can be characterised neatly: they are in one-to-one correspondence to natural transformations of type

$$\forall A \, . \, \mathscr{C}(A, Y) \to \mathscr{D}(\mathsf{F}\ A, Z) \ . \tag{4.2}$$

Recall that in GH the instance of *size* for a type F of kind $\star \to \star$ has type $Size_{\star\to\star}\ \mathsf{F} = \forall A \ . \ (A \to \mathbb{N}) \to (\mathsf{F}\ A \to \mathbb{N})$. Equating polymorphic functions with natural transformations, the *size* instance induces an endofunctor over slice categories. The exact match between GH and the categorical model is quite reassuring and it shows that we are on the right track. The Yoneda Lemma actually allows us to simplify the type (4.2) to $\mathscr{D}(\mathsf{F}\ Y, Z)$. While this is a convenient simplification in this instance, it does not generalise to higher-order kinds.

Turning to the proof of the claim, let us first spell out the naturality condition associated with (4.2). Let τ be a natural transformation of this type and let $h : A \leftarrow B$, then

$$\tau\ A\ f \cdot \mathsf{F}\ h = \tau\ B\ (f \cdot h)\ , \tag{4.3}$$

for all $f : A \to Y$. Note that $\mathscr{C}(-, Y)$ and $\mathscr{D}(\mathsf{F}\ -, Z)$ are contravariant functors of type $\mathscr{C}^{\mathsf{op}} \to \mathbf{Set}$, which is why the direction of h is reversed.

Given a natural transformation τ, we can construct a lifting $\mathsf{F}_\tau : \mathscr{C} \downarrow Y \to \mathscr{D} \downarrow Z$ as follows

$$\mathsf{F}_\tau\ (A, a) = (\mathsf{F}\ A, \tau\ A\ a)\ , \tag{4.4a}$$

$$\mathsf{F}_\tau\ f = \mathsf{F}\ f\ . \tag{4.4b}$$

Because F_τ has to be a lifting, its action on A and f is given by F; the natural transformation τ specifies the action on a. Since F is a functor, it is immediate that F_τ preserves identity and composition. It remains to check that F_τ respects the types.

$$\mathsf{F}_\tau\ f : \mathsf{F}_\tau\ (A, a) \to \mathsf{F}_\tau\ (B, b) \quad \Longleftarrow \quad f : (A, a) \to (B, b)$$

We reason

$$\begin{aligned}
& \tau\ B\ b \cdot \mathsf{F}\ f \\
= {} & \quad \{\ \tau \text{ is natural (4.3)}\ \} \\
& \tau\ A\ (b \cdot f) \\
= {} & \quad \{\ \text{assumption}\ \} \\
& \tau\ A\ a\ .
\end{aligned}$$

Conversely, given a lifting $\bar{\mathsf{F}}$ we can define a natural transformation

$$\tau_{\bar{\mathsf{F}}}\ A\ a = \alpha\ (\bar{\mathsf{F}}\ (A, a))\ . \tag{4.5}$$

We apply $\bar{\mathsf{F}}$ to the object (A, a) and then extract the arrow. We have to show that $\tau_{\bar{\mathsf{F}}}$ is natural. Let $h : A \leftarrow B$, then

$$\begin{aligned}
& \tau_{\bar{\mathsf{F}}}\ A\ f \cdot \mathsf{F}\ h \\
= {} & \quad \{\ \text{definition of } \tau_{\bar{\mathsf{F}}} \text{ (4.5)}\ \} \\
& \alpha\ (\bar{\mathsf{F}}\ (A, f)) \cdot \mathsf{F}\ h \\
= {} & \quad \{\ \bar{\mathsf{F}} \text{ lifting of } \mathsf{F} \text{ and } h : (B, f \cdot h) \to (A, f)\ \} \\
& \alpha\ (\bar{\mathsf{F}}\ (A, f)) \cdot \bar{\mathsf{F}}\ h \\
= {} & \quad \{\ \alpha \text{ natural and } \bar{\mathsf{F}}\ h : \bar{\mathsf{F}}\ (B, f \cdot h) \to \bar{\mathsf{F}}\ (A, f)\ \} \\
& \alpha\ (\bar{\mathsf{F}}\ (B, f \cdot h)) \\
= {} & \quad \{\ \text{definition of } \tau_{\bar{\mathsf{F}}} \text{ (4.5)}\ \} \\
& \tau_{\bar{\mathsf{F}}}\ B\ (f \cdot h)\ .
\end{aligned}$$

It is not too hard to see that liftings and natural transformations of type (4.2) are in one-to-one correspondence:

$$\tau_{\mathsf{F}_\tau} = \tau\ , \tag{4.6a}$$

$$\mathsf{F}_{\tau_{\bar{\mathsf{F}}}} = \bar{\mathsf{F}}\ . \tag{4.6b}$$

One direction is just a matter of unrolling the definitions.

$$\begin{aligned}
& \tau_{\mathsf{F}_\tau}\ A\ a \\
= {} & \quad \{\ \text{definition of } \tau_{\bar{\mathsf{F}}} \text{ (4.5)}\ \} \\
& \alpha\ (\mathsf{F}_\tau\ (A, a)) \\
= {} & \quad \{\ \text{definition of } \mathsf{F}_\tau \text{ (4.4a)}\ \} \\
& \alpha\ (\mathsf{F}\ A, \tau\ A\ a) \\
= {} & \quad \{\ \text{definition of } \alpha \text{ (4.1)}\ \} \\
& \tau\ A\ a
\end{aligned}$$

For the other direction we make use of the fact that $\bar{\mathsf{F}}$ and F_τ are liftings of F.

$$\begin{aligned}
& \mathsf{F}_{\tau_{\bar{\mathsf{F}}}}\ (A, a) \\
= {} & \quad \{\ \text{definition of } \mathsf{F}_\tau \text{ (4.4a)}\ \} \\
& (\mathsf{F}\ A, \tau_{\bar{\mathsf{F}}}\ A\ a) \\
= {} & \quad \{\ \text{definition of } \tau_{\bar{\mathsf{F}}} \text{ (4.5)}\ \} \\
& (\mathsf{F}\ A, \alpha\ (\bar{\mathsf{F}}\ (A, a))) \\
= {} & \quad \{\ \bar{\mathsf{F}} \text{ lifting of } \mathsf{F}\ \} \\
& \bar{\mathsf{F}}\ (A, a)
\end{aligned}
\qquad
\begin{aligned}
& \mathsf{F}_{\tau_{\bar{\mathsf{F}}}}\ f \\
= {} & \quad \{\ \mathsf{F}_\tau \text{ lifting of } \mathsf{F}\ \} \\
& \mathsf{F}\ f \\
= {} & \quad \{\ \bar{\mathsf{F}} \text{ lifting of } \mathsf{F}\ \} \\
& \bar{\mathsf{F}}\ f
\end{aligned}$$

So far we have discussed liftings of endofunctors. Since type expressions may have arbitrary kinds, we need to generalise the notion to arbitrary higher-order functors. To this end, we set up a logical relation, defined by induction over the structure of kinds, see Figure 4. An object C of a slice category is a lifting of an object A of the underlying category, if A is C's 'carrier'. The second clause expresses that pairs are related iff the components are related. Finally, the third clause closes the logical relation under application and abstraction. For example,

$$(\mathsf{F}, \bar{\mathsf{F}}) \in \mathscr{R}_{\star \to \star} \quad \Longleftrightarrow \quad \mathsf{U} \circ \bar{\mathsf{F}} = \mathsf{F} \circ \mathsf{U}\ ,$$

as desired.

We have two interpretations of type expressions, the standard one $\mathscr{I}$ and the 'generic' one $\mathscr{G}$. The Basic Lemma of logical relations [22] guarantees that the two interpretations are related,

$$(\mathscr{I}[\![t]\!], \mathscr{G}[\![t]\!]) \in \mathscr{R}_{\mathfrak{T}} \text{ for all } t : \mathfrak{T},$$

if the interpretations of the type constants are related

$$(\mathscr{I}_c, \mathscr{G}_c) \in \mathscr{R}_{\mathfrak{T}} \text{ for all } c : \mathfrak{T}.$$

Returning to our running example, the definition of a generic crush, we shall now consider the various type constants, one by one, and discuss how to define appropriate liftings. Quite pleasingly, more than half of the definitions are "for free" in the sense that there is one canonical choice. For instance, we shall see that the coproduct in $\mathscr{C} \downarrow Y$ is the lifting of the coproduct in $\mathscr{C}$. Consequently, a canonical interpretation of $+$ is $+$, the coproduct in $\mathscr{C} \downarrow Y$. But we are leaping ahead.

4.4 Initial object

The initial object in $\mathscr{C} \downarrow Y$ is a lifting of the initial object in $\mathscr{C}$. This is because the underlying functor preserves colimits: $\mathsf{U}_Y\ 0 = 0$. Since furthermore there is a unique arrow from 0 to Y we have

$$0 = (0, ¡_Y)\ .$$

The unique arrow from $(0, ¡_Y)$ to some other object (A, a) is given by

$$¡_{(A,a)} = ¡_A\ .$$

We have to check that $¡_{(A,a)} : (0, ¡_Y) \to (A, a)$. The resulting condition, $¡_Y = a \cdot ¡_A$, is just an instance of fusion.

$$\begin{tikzcd}
0 \arrow[rr, dashed, "¡_A"] \arrow[dr, "¡_Y"'] & & A \arrow[dl, "a"] \\
& Y &
\end{tikzcd}$$

Since 0 is a lifting of 0 and there is only one lifting, the definition of a generic consumer for 0 is for free.

$$\mathscr{G}_0 = 0$$

$$\begin{aligned}
(A, C) \in \mathscr{R}_{*} &\iff A = \mathsf{U}\; C \\
(A, C) \in \mathscr{R}_{\mathfrak{T} \times \mathfrak{U}} &\iff (\mathsf{Outl}\; A, \mathsf{Outl}\; C) \in \mathscr{R}_{\mathfrak{T}} \;\wedge\; (\mathsf{Outr}\; A, \mathsf{Outr}\; C) \in \mathscr{R}_{\mathfrak{U}} \\
(\mathsf{F}, \mathsf{H}) \in \mathscr{R}_{\mathfrak{T} \to \mathfrak{U}} &\iff \forall X\; Z \;.\; (X, Z) \in \mathscr{R}_{\mathfrak{T}} \Longrightarrow (\mathsf{F}\; X, \mathsf{H}\; Z) \in \mathscr{R}_{\mathfrak{U}}
\end{aligned}$$

Figure 3. Generalising the notion of lifting to higher kinds (slice categories).

4.5 Final object

The slice category $\mathscr{C} \downarrow Y$ always has a final object:

$$1 = (Y, id_Y) \;.$$

The unique arrow $!_{(A,a)}$ from (A, a) to (Y, id_Y) has to satisfy $a = id_Y \cdot !_{(A,a)}$, Consequently,

$$!_{(A,a)} = a \;.$$

As an aside, we have seen $!_{(A,a)}$ before: it is just another name for the natural transformation $\alpha : \mathsf{Id} \xrightarrow{\cdot} \Delta\, 1$ that extracts the arrow component from a slice object. (Here, $\Delta\, 1$ is the constant functor that maps each object to 1.)

$$\begin{tikzcd}
A \arrow[rr, dashed, tail, "a"] \arrow[dr, "a"'] & & Y \arrow[dl, "id_Y"] \\
& Y &
\end{tikzcd}$$

Another way to determine the final object is to recall that P_Y preserves limits: $\mathsf{P}_Y\; 1 = 1$. The reader is invited to check the details.

Turning to the definition of the instance for crush, we note that $1 : \mathscr{C} \downarrow Y$ is *not* a lifting of $1 : \mathscr{C}$. (This also implies that U_Y has no left adjoint.) Hence, the generic programmer has to supply an instance definition:

$$\mathscr{G}_1 = \text{①}\; \mathbf{where}\; \text{①} = (1, e) \;.$$

Here, $e : 1 \to Y$ is the constant given to us.

4.6 Coproduct

Since a coproduct is a colimit, we might hope to obtain this instance for free, as well. Recall that a coproduct consists of four pieces of data: an object $A + B$, constructors *inl* and *inr*, and a mediating arrow $\triangledown$. Since U_Y preserves colimits, the carrier of the coproduct in a slice category is easy to determine: we have $\mathsf{U}_Y\; ((A, a) + (B, b)) = \mathsf{U}_Y\; (A, a) + \mathsf{U}_Y\; (B, b) = A + B$. To determine the arrow component, we reason as follows. One would hope that *inl* and *inr* also serve as constructors in the slice category. This is the case if $inl : (A, a) \to (A + B, x)$ and $inr : (B, b) \to (A + B, x)$, where x is the unknown arrow. Let's calculate.

$$\begin{aligned}
& inl : (A, a) \to (A + B, x) \;\wedge\; inr : (B, b) \to (A + B, x) \\
\iff\; & \{ \text{definition of } \mathscr{C} \downarrow Y \} \\
& a = inl \cdot x \;\wedge\; b = inr \cdot x \\
\iff\; & \{ \text{universal property} \} \\
& a \triangledown b = x
\end{aligned}$$

Consequently, the coproduct in a slice category is defined

$$(A, a) + (B, b) = (A + B, a \triangledown b) \;.$$

It remains to show that the mediating arrow $\triangledown$ respects the types.

$$\begin{aligned}
& f \triangledown g : (A + B, a \triangledown b) \to (C, c) \\
\Longleftarrow\; & f : (A, a) \to (C, c) \;\wedge\; g : (B, b) \to (C, c)
\end{aligned}$$

We reason

$$\begin{aligned}
& c \cdot (f \triangledown g) \\
=\; & \{ \text{fusion} \} \\
& (c \cdot f) \triangledown (c \cdot g) \\
=\; & \{ \text{assumption} \} \\
& a \triangledown b \;.
\end{aligned}$$

Since $+$ is a lifting, the instance of generic crush is indeed for free.

$$\mathscr{G}_{+} = +$$

We should point out, however, that this is merely a canonical choice, it is by no means the only one. Generalising the argument of Section 4.3 to bifunctors, liftings of $+$ are in one-to-one correspondence to natural transformations of type

$$\forall A\; B \;.\; \mathscr{C}(A, Y) \times \mathscr{C}(B, Y) \to \mathscr{C}(A + B, Y) \;.$$

Using Yoneda's Lemma once more, we find that natural transformations of this type in turn are in one-to-one correspondence to arrows of type $\mathscr{C}(Y + Y, Y)$.

$$\mathscr{C}(Y + Y, Y) \cong \forall A\, B \;.\; \mathscr{C}(A, Y) \times \mathscr{C}(B, Y) \to \mathscr{C}(A + B, Y)$$

As an example, a generic encoder that maps a value to a bit string might use the lifting induced by $cons\; 0 \triangledown cons\; 1$, where *cons* prepends a bit to a bit string.

4.7 Product

A product in a slice category is a so-called pullback in the underlying category, so $\times$ is not a lifting of $\times$. We have to start afresh.

In order to define a lifting of $\times$ we use the characterisation of bifunctors provided in the previous section. Liftings of $\times$ are in one-to-one correspondence to binary operations of type $Y \times Y \to Y$. Using the operation $op : Y \times Y \to Y$ given to us, a suitable lifting for crush is defined

$$(A, a) \otimes (B, b) = (A \times B, op \cdot (a \times b)) \;. \tag{4.7}$$

The interpretation of $\times$ then uses this lifting.

$$\mathscr{G}_{\times} = \otimes$$

4.8 Initial algebra

Initial algebras are colimits so again one might hope to get the definition for free. The only slight 'complication' is that the kind of the type constructor is more complicated: μ is a higher-order functor that takes a functor to an object. Instantiating the logical relation of Section 4.3 to kind $(\star \to \star) \to \star$ we have to show that (as usual we overload μ to denote both the initial algebra in the slice category and in the underlying category)

$$\mathsf{U} \circ \bar{\mathsf{F}} = \mathsf{F} \circ \mathsf{U} \implies \mathsf{U}\, (\mu \bar{\mathsf{F}}) = \mu \mathsf{F} \;.$$

Note that we can *assume* that the argument of μ is a lifting. We only have to determine the initial algebra of liftings, which simplifies matters. The implication already fixes the 'carrier' of $\mu \bar{\mathsf{F}}$, it remains to determine the arrow component. We apply the same reasoning as for coproducts: we speculate that in and $\llparenthesis - \rrparenthesis$ are inherited from the base category. For the algebra in, this entails

$$\begin{aligned}
& in : \bar{\mathsf{F}}\,(\mu\bar{\mathsf{F}}) \to \mu\bar{\mathsf{F}} \\
\iff\; & \{\ \text{setting } \mu\bar{\mathsf{F}} = (\mu\mathsf{F}, x)\ \} \\
& in : \bar{\mathsf{F}}\,(\mu\mathsf{F}, x) \to (\mu\mathsf{F}, x) \\
\iff\; & \{\ \text{characterisation of liftings: } \mathsf{F}_{\tau_{\bar{\mathsf{F}}}} = \bar{\mathsf{F}} \text{ (4.6b) }\} \\
& in : (\mu\mathsf{F}, \tau_{\bar{\mathsf{F}}}\,x) \to (\mu\mathsf{F}, x) \\
\iff\; & \{\ \text{definition of } \mathscr{C} \downarrow Y\ \} \\
& \tau_{\bar{\mathsf{F}}}\,x = x \cdot in \\
\iff\; & \{\ \text{uniqueness property (3.1) }\} \\
& x = \llparenthesis \tau_{\bar{\mathsf{F}}} \rrparenthesis\ .
\end{aligned}$$

Consequently, the initial algebra is given by

$$\mu\bar{\mathsf{F}} = (\mu\mathsf{F}, \llparenthesis \tau_{\bar{\mathsf{F}}} \rrparenthesis)\ . \tag{4.8}$$

It remains to show that $\llparenthesis - \rrparenthesis$ also respects the types.

$$\begin{aligned}
& \llparenthesis - \rrparenthesis : (\forall \bar{X}\ .\ (\bar{X} \to \bar{B}) \to (\bar{\mathsf{F}}\,\bar{X} \to \bar{B})) \to (\mu\bar{\mathsf{F}} \to \bar{B}) \\
\iff\; & \{\ \text{definitions }\} \\
& \tau_{\bar{\mathsf{F}}}\,(b \cdot f) = b \cdot \Psi\,f \quad\Longrightarrow\quad \llparenthesis \tau_{\bar{\mathsf{F}}} \rrparenthesis = b \cdot \llparenthesis \Psi \rrparenthesis \\
\iff\; & \{\ \text{fusion (3.3) }\} \\
& true
\end{aligned}$$

So (4.8) defines a lifting and once more we obtain an instance of generic crush for free.

$$\mathscr{G}_{\mu} = \mu$$

Again, we should point out that this is only a canonical choice. Adapting the argument of Section 4.3 to higher-order functors, liftings of μ are in one-to-one correspondence to higher-order natural transformations of type

$$\forall \mathsf{F}\ .\ (\forall X\ .\ \mathscr{C}(X, B) \to \mathscr{C}(\mathsf{F}\,X, B)) \to \mathscr{C}(\mu\mathsf{F}, B)\ .$$

We have discussed in Section 3.3 that $\llparenthesis - \rrparenthesis$ indeed enjoys this property. We are free to use a different recursion operator instead, but the replacement must satisfy the same higher-order naturality condition.

4.9 Summary: Generic crush

To summarise, the development in this section is an instance of the general framework set up in Section 3. The only minor change is that kind $\star$ is interpreted by a slice category over the ambient category.

$$\mathscr{G}_{\star} = \mathscr{C} \downarrow Y$$

The slice category allows us to associate an arrow with an object, the instance of a generic function at that type.

Type expressions are interpreted as functors over this slice category. These functors cannot be arbitrary, they have to be liftings of the standard interpretation to ensure that the arrows actually represent generic instances.

In Generic Haskell, generic instances are polymorphic functions of higher ranks. We have seen that their categorical counterparts, higher-order natural transformations, are in one-to-one correspondence to liftings, which nicely reinforces the approach.

We have discussed generic crush as a running example, which is given by

$$\begin{aligned}
&\mathscr{G}_0 = 0 \qquad && \mathscr{G}_+ = + \qquad && \mathscr{G}_\mu = \mu\ . \\
&\mathscr{G}_1 = ① && \mathscr{G}_\times = \otimes &&
\end{aligned}$$

Since the underlying functor preserves colimits, initial objects, coproducts and initial algebras are, in fact, liftings. We have argued that for $crush$, these canonical functors are indeed the right choices. Consequently, the generic programmer only has to supply definitions for the final object and products. These are uniquely determined by a constant $e : 1 \to Y$ and a binary operation $op : Y \times Y \to Y$, the two defining ingredients of a crush. It is quite pleasing to see how everything falls into place.

Let us finally consider some example instantiations. The "Hello World" example of generic programming, the list datatype $\mathsf{List} = \Lambda\,A\ .\ \mu(\Lambda\,B\ .\ 1 + A \times B)$, yields the functor (we only show the object mapping, the action on arrows is as before)

$$\begin{aligned}
& \Lambda\,\bar{A}\ .\ \mu(\Lambda\,\bar{B}\ .\ ① + \bar{A} \otimes \bar{B}) \\
=\; & \{\ \text{liftings: } \bar{A} = (A, a) \text{ and } \bar{B} = (B, b)\ \} \\
& \Lambda\,(A, a)\ .\ \mu(\Lambda\,(B, b)\ .\ ① + (A, a) \otimes (B, b)) \\
=\; & \{\ \text{definition of } ①, + \text{ and } \otimes\ \} \\
& \Lambda\,(A, a)\ .\ \mu(\Lambda\,(B, b)\ .\ (1 + A \times B, e \mathbin{\triangledown} (op \cdot (a \times b)))) \\
=\; & \{\ \text{definition of } \mu \text{ (4.8) }\} \\
& \Lambda\,(A, a)\ .\ (\mathsf{List}\,A, \llparenthesis \lambda\,b\ .\ e \mathbin{\triangledown} (op \cdot (a \times b)) \rrparenthesis)\ .
\end{aligned}$$

For the higher-order functor GRose we obtain

$$\begin{aligned}
& \Lambda\,\bar{\mathsf{F}}\ .\ \Lambda\,(A, a)\ .\ \mu(\Lambda\,(B, b)\ .\ (A, a) \otimes \bar{\mathsf{F}}\,(B, b)) \\
=\; & \{\ \text{definition of } \otimes \text{ (4.7) }\} \\
& \Lambda\,\bar{\mathsf{F}}\ .\ \Lambda\,(A, a)\ .\ \mu(\Lambda\,(B, b)\ .\ (A \times \mathsf{F}\,B, op \cdot (a \otimes \tau_{\bar{\mathsf{F}}}\,b))) \\
=\; & \{\ \text{definition of } \mu \text{ (4.8) }\} \\
& \Lambda\,\bar{\mathsf{F}}\ .\ \Lambda\,(A, a)\ .\ (\mathsf{GRose}\,\mathsf{F}\,A, \llparenthesis \lambda\,b\ .\ op \cdot (a \otimes \tau_{\bar{\mathsf{F}}}\,b) \rrparenthesis)\ .
\end{aligned}$$

5. Simple generic producers

Let us turn our attention to generic producers. We can now reap the fruits of categorical duality—producers and consumers are dual, and everything we said in Section 4 nicely dualises to producers. For that reason, we only sketch the construction and work through an example.

The dual of a slice category $\mathscr{C} \downarrow Y$ is a *coslice* category $X \downarrow \mathscr{C}$, whose objects are arrows of type $\mathscr{C}(X, A)$ and whose arrows are commuting triangles.

$$\begin{tikzcd}
X \arrow[d, "f"'] & & X \arrow[dl, "f"'] \arrow[dr, "g"] & & X \arrow[d, "g"] \\
A & A \arrow[rr, "h"'] & & B & B
\end{tikzcd}$$

The standard textbook example of a coslice category is $1 \downarrow \mathbf{Set}$, the *category of pointed sets*. An arrow of type $\mathbf{Set}(1, A)$ selects a so-called *base-point* in A. The arrows in $1 \downarrow \mathbf{Set}$ preserve this base-point. We can turn the example into an application of generic programming by providing a generic definition of the selector.

$$\begin{aligned}
null_{Int} &= zero \cdot !_X \\
null_1 &= !_X \\
null_+ &= \lambda\,(f, g)\ .\ inl \cdot f \\
null_\times &= \lambda\,(f, g)\ .\ f \mathbin{\vartriangle} g \\
null_\nu &= \lambda\,\gamma\ .\ \llbracket \gamma \rrbracket
\end{aligned}$$

Two cases are missing, $null$ cannot be defined for 0 and μ. In general, there is no arrow $\mathscr{C}(1, 0)$. (In a cartesian closed category, the existence of an arrow $\mathscr{C}(1, 0)$ implies that $\mathscr{C}$ is degenerate.) For initial algebras, the reasoning is as follows. To construct an element of an initial algebra, we have to use $in : \mathscr{C}(\mathsf{F}\,(\mu\mathsf{F}), \mu\mathsf{F})$. This leaves us with the task of constructing an element of $\mathsf{F}\,(\mu\mathsf{F})$. The argument of $null_\mu$ of type $\forall X\ .\ \mathscr{C}(1, X) \to (1, \mathsf{F}\,X)$ allows us to do this, provided we have an arrow of type $\mathscr{C}(1, \mu\mathsf{F})$—a vicious circle. (More formally, since $\mu\mathsf{Id} \cong 0$ the above argument for final objects also applies here.) The definition for 1, $\times$ and ν (the limits) are "for free": the instances of $null$ are the mediating arrows for these types $(!, - \vartriangle =, [\![-]\!])$. For Int and $+$ there is a choice, quite arbitrarily we select 0 as the base-point in Int, and $inl\, null_A$ as the base-point in $A + B$.

Coinductive types such as the type of streams admit base-points. Specialising the definition to $\mathsf{Stream} = \Lambda\, A\, .\, \nu(\Lambda\, B\, .\, A \times B)$ yields the functor (again, we only show the object part)

$$\Lambda\, (A, a)\, .\, (\mathsf{Stream}\, A, [\![\lambda\, b\, .\, a \vartriangle b]\!])\ .$$

The associated natural transformation $null_{\mathsf{Stream}}$ is a base-point transformer, it takes selectors to selectors: $\forall A\, .\, \mathscr{C}(1, A) \to \mathscr{C}(1, \mathsf{Stream}\, A)$. For example, $null_{\mathsf{Stream}}\, (zero \cdot !)$ yields the constant streams of zeros. The naturality of $null_{\mathsf{Stream}}$ means that the instance enjoys a simple fusion property: $\mathsf{Stream}\, h \cdot null_{\mathsf{Stream}}\, f = null_{\mathsf{Stream}}\, (h \cdot f)$.

6. Outlook: Generic programs

This section works towards modelling the whole of Generic Haskell. For reasons of space, we only provide an overview, sketching the categorical constructions.

The development of Section 4 is not general enough to model generic consumers such as equality, where the source is a pair of elements of the type index: $\mathscr{C}(A \times A, Bool)$. To accommodate for this, we allow A to appear in a context, modelled by a functor S. (The name S is mnemonic for Source.) Thus, at base types, the type of a generic consumer is $\mathscr{C}(\mathsf{S}\, A, Y)$. While the target of S has to be $\mathscr{C}$, its source can be an arbitrary category.

$$consume \quad : \quad \mathscr{S} \xrightarrow{\ \mathsf{S}\ } \mathscr{C} \xleftarrow{\ Y\ } \mathbf{1}$$

Simple consumers are a special case of this construction where $\mathsf{S} = \mathsf{Id}_{\mathscr{C}}$. For generic equality, a possible source functor is $\mathsf{Sq} : \mathscr{C} \to \mathscr{C}$ with $\mathsf{Sq}\, A = A \times A$ and $\mathsf{Sq}\, f = f \times f$. However, there is an alternative choice, which leads to a more general notion of equality.

$$equal \quad : \quad \mathscr{C} \times \mathscr{C} \xrightarrow{\ \times\ } \mathscr{C} \xleftarrow{\ Bool\ } \mathbf{1}$$

The source is a product category. Since A in $\mathscr{C}(\times A, Bool)$ ranges over objects in $\mathscr{C} \times \mathscr{C}$, the element types can actually be different: $\mathscr{C}(A_1 \times A_2, Bool)$. For example, the instance of equality for lists has type $\forall A_1\ A_2\ .\ \mathscr{C}(A_1 \times A_2, Bool) \to \mathscr{C}(\mathsf{List}\, A_1 \times \mathsf{List}\, A_2, Bool)$, whereas in the first model the element types have to be identical: $\forall A\, .\, \mathscr{C}(A \times A, Bool) \to \mathscr{C}(\mathsf{List}\, A \times \mathsf{List}\, A, Bool)$. Clearly, the second model is more general.

Turning to the dual setting, the paradigmatic example of a generic producer is $read$, which constructs an element from some string representation. Ignoring the details of the representation, $read$ is interesting, as it involves a parsing monad to organise the working. How can we fit monadic computations into the picture? The answer is simple: we dualise the approach above so that the type index can be embedded in a context: $\mathscr{C}(X, \mathsf{T}\, A)$. (The name T is mnemonic for Target.)

$$read \quad : \quad \mathbf{1} \xrightarrow{\ 1\ } \mathscr{C} \xleftarrow{\ Parser\ } \mathscr{C}$$

We are now ready for the general construction, which involves one further generalisation step. Slice and coslice categories abstract over one object A, which appears either in the source, $\mathscr{C}(\mathsf{S}\, A, Y)$, or in the target, $\mathscr{C}(X, \mathsf{T}\, A)$. An obvious generalisation is to abstract away from both the source and the target: $\mathscr{C}(\mathsf{S}\, A, \mathsf{T}\, B)$. Two different objects are involved, because we may need to interpret the type index differently for the source and the target. The functors S and T do not have to be endofunctors; their source categories can be arbitrary.

$$generic \quad : \quad \mathscr{S} \xrightarrow{\ \mathsf{S}\ } \mathscr{C} \xleftarrow{\ \mathsf{T}\ } \mathscr{T}$$

All in all, there are three knobs to turn. First of all and most importantly, we have to pick a base category $\mathscr{C}$. Choices include $\mathbf{Set}$, $1 \downarrow \mathbf{Set}$ or $\mathbf{Cpo}_{\perp}$. Second, we have to identify the source context, in which the type index appears, fixing a category $\mathscr{S}$ and a functor S. Third, we have to do the same for the target. Before we introduce the generalisation of (co)slice categories, let us examine how standard examples of generic functions fit into this picture.

Generic consumers are a special case of the construction.

$$consume \quad : \quad \mathscr{S} \xrightarrow{\ \mathsf{S}\ } \mathscr{C} \xleftarrow{\ Y\ } \mathbf{1}$$

The target functor is the constant functor which sends $*$, the only object of $\mathbf{1}$, to Y. (As usual, we identify a functor of type $\mathbf{1} \to \mathscr{C}$ with an object of $\mathscr{C}$.) For generic producers, the situation is dual.

$$produce \quad : \quad \mathbf{1} \xrightarrow{\ X\ } \mathscr{C} \xleftarrow{\ \mathsf{T}\ } \mathscr{T}$$

In the examples above, one of the two functors is constant. This need not be the case: to model mapping functions we use

$$map \quad : \quad \mathscr{C} \xrightarrow{\ \mathsf{Id}\ } \mathscr{C} \xleftarrow{\ \mathsf{Id}\ } \mathscr{C}\ ,$$

which provides an alternative view on GH's map function introduced in Section 2. As a final example, generic zipping, like equality, involves a product category.

$$zip \quad : \quad \mathscr{C} \times \mathscr{C} \xrightarrow{\ \times\ } \mathscr{C} \xleftarrow{\ \mathsf{Id}\ } \mathscr{C}$$

Haskell's $zipWith$ function, which combines two lists into a single list, is an instance of this scheme.

The slogan of the paper is that generic functions are functors between comma categories, a notion we introduce next. Let $\mathsf{S} : \mathscr{S} \to \mathscr{C}$ and $\mathsf{T} : \mathscr{T} \to \mathscr{C}$ be functors. The *comma category* $\mathsf{S} \downarrow \mathsf{T}$ has as objects arrows and as arrows commuting squares:

$$\begin{tikzcd}
\mathsf{S}\, A \arrow[d, "f"'] & & \mathsf{S}\, A \arrow[r, "\mathsf{S}\, h"] \arrow[d, "f"'] & \mathsf{S}\, C \arrow[d, "g"] & & \mathsf{S}\, C \arrow[d, "g"] \\
\mathsf{T}\, B & & \mathsf{T}\, B \arrow[r, "\mathsf{T}\, k"'] & \mathsf{T}\, D & & \mathsf{T}\, D\ .
\end{tikzcd}$$

Formally, an object of the comma category $\mathsf{S} \downarrow \mathsf{T}$ is a triple (A, f, B) where A is an object of $\mathscr{S}$, B is an object of $\mathscr{T}$ and $f : \mathsf{S}\, A \to \mathsf{T}\, B$ is an arrow of $\mathscr{C}$. An arrow $(h, k) : (A, f, B) \to (C, g, D) : \mathsf{S} \downarrow \mathsf{T}$ of the comma category is a pair of arrows $h : A \to C : \mathscr{S}$ and $k : B \to D : \mathscr{T}$ such that $\mathsf{T}\, k \cdot f = g \cdot \mathsf{S}\, h$.

The triple (A, f, B) models an instance of a generic function at some type t of kind $\star$. Generally, a generic function is determined by three pieces of information: we have to show how to interpret t as an object A in $\mathscr{S}$, we have to interpret t as an object B in $\mathscr{T}$, and we have to provide an arrow of type $\mathscr{C}(\mathsf{S}\, A, \mathsf{T}\, B)$.

$$\begin{tikzcd}
& \mathbf{1} \arrow[dl, "\mathscr{I}[\![t]\!]"'] \arrow[dr, "\mathscr{J}[\![t]\!]"] & & & \mathsf{S}\, (\mathscr{I}[\![t]\!]) \arrow[d, "\mathscr{G}[\![t]\!]"] \\
\mathscr{S} \arrow[r, "\mathsf{S}"'] & \mathscr{C} & \mathscr{T} \arrow[l, "\mathsf{T}"] & & \mathsf{T}\, (\mathscr{J}[\![t]\!])
\end{tikzcd}$$

In Generic Haskell these three pieces of information are given separately. Using a comma category we tie them together.

For (co)slice categories we had a forgetful functor to the underlying category. Since an object in a comma category combines two objects, there are two projection functors: $\mathsf{Src} : \mathsf{S} \downarrow \mathsf{T} \to \mathscr{S}$ extracts the source and $\mathsf{Trg} : \mathsf{S} \downarrow \mathsf{T} \to \mathscr{T}$ extracts the target object. The following non-commutative diagram summarises the type information.

$$\begin{tikzcd}
\mathsf{S}\downarrow\mathsf{T} \arrow[r, "\mathsf{Trg}"] \arrow[d, "\mathsf{Src}"'] & \mathscr{T} \arrow[d, "\mathsf{T}"] \\
\mathscr{S} \arrow[r, "\mathsf{S}"'] & \mathscr{C}
\end{tikzcd}$$

Like before, we require that the interpretation of a generic function constitutes a lifting. For a functor between two comma categories the notion of lifting is defined as follows. A functor $\mathsf{H} : \mathsf{S}\downarrow\mathsf{T} \to \mathsf{S}'\downarrow\mathsf{T}'$ is a *lifting* of $\mathsf{F} : \mathscr{S} \to \mathscr{S}'$ and $\mathsf{G} : \mathscr{T} \to \mathscr{T}'$ if $\mathsf{Src} \circ \mathsf{H} = \mathsf{F} \circ \mathsf{Src}$ and $\mathsf{Trg} \circ \mathsf{H} = \mathsf{G} \circ \mathsf{Trg}$.

$$\begin{tikzcd}
\mathscr{S} \arrow[d, "\mathsf{F}"'] & \mathsf{S}\downarrow\mathsf{T} \arrow[l, "\mathsf{Src}"'] \arrow[r, "\mathsf{Trg}"] \arrow[d, "\mathsf{H}"] & \mathscr{T} \arrow[d, "\mathsf{G}"] \\
\mathscr{S}' & \mathsf{S}'\downarrow\mathsf{T}' \arrow[l, "\mathsf{Src}"] \arrow[r, "\mathsf{Trg}"'] & \mathscr{T}'.
\end{tikzcd}$$

Generalising the argument of Section 4.3 one can then show that liftings of F and G are in one-to-one correspondence to natural transformations of type

$$\forall A\ B\ .\ \mathscr{C}(\mathsf{S}\ A, \mathsf{T}\ B) \to \mathscr{C}'(\mathsf{S}'\ (\mathsf{F}\ A), \mathsf{T}'\ (\mathsf{G}\ B))\ \ .$$

As a brief example, in the case of generic map, that is, $\mathsf{S} = \mathsf{T} = \mathsf{S}' = \mathsf{T}' = \mathsf{Id}$, we obtain the familiar type

$$\forall A\ B\ .\ \mathscr{C}(A, B) \to \mathscr{C}'(\mathsf{F}\ A, \mathsf{G}\ B)\ \ .$$

To generalise the notion of lifting to arbitrary functors, we use again a logical relation, this time a ternary one, see Figure 4. The interpretation of a generic function $\mathscr{G}[\![t]\!]$ has to be a lifting of the interpretations for the source $\mathscr{I}[\![t]\!]$ and the target $\mathscr{J}[\![t]\!]$:

$$(\mathscr{I}[\![t]\!], \mathscr{G}[\![t]\!], \mathscr{J}[\![t]\!]) \in \mathscr{R}_{\mathfrak{T}} \text{ for all } t : \mathfrak{T}.$$

Again, the Basic Lemma of logical relations guarantees that this holds if the interpretations of the type constants are related.

Let us conclude by noting that the categorical framework also accommodates type-indexed datatypes [14]. A type that is defined by induction on the structure of types is simply an interpretation in the sense of Section 3.4, such as $\mathscr{I}[\![t]\!]$ and $\mathscr{J}[\![t]\!]$.

7. Related and future work

There is a considerable body of work on datatype-generic programming (DGP), see [13, 23] for recent overviews. PolyP [15], one of the first languages with support for DGP, grew out of the work on the Algebra of Programming [18, 20]. PolyP is based on a grammar for bifunctors ($\star \times \star \to \star$) and regular functors ($\star \to \star$):

$$\begin{aligned}
&\mathsf{F} = \mathsf{K}\ T \mid \mathsf{K}\ 1 \mid \mathsf{Par} \mid \mathsf{Rec} \mid \mathsf{F} + \mathsf{F} \mid \mathsf{F} \times \mathsf{F} \mid \mathsf{D} \circ \mathsf{F}\ \ ;\\
&\mathsf{D} = \mu\mathsf{F}\ \ .
\end{aligned}$$

Though this language of functors is less general than our language based on the simply typed lambda calculus (STLC), the generic programmer actually has to provide instances for more cases, including two cases for type variables (Par and Rec). PolyP only considers initial algebras, μ *applied* to some functor It is the observation that μ itself constitutes a functor that makes the current paper fly.

The semantics described here is based on the interpretation of the STLC in a cartesian closed category. A previous approach by the first author [10] provided a syntactic model: building on the notion of an applicative structure, type terms are interpreted by terms of the polymorphic lambda calculus. The reconciliation of the two approaches is left for future work. Generic Haskell [3, 4] is a fairly substantial language. The semantics presented here covers the core of the language including type-indexed datatypes [14].

Mendler-style (un-) folds were introduced in a type-theoretic setting by, well, Mendler [21]. The categorical justification of Mendler-style recursion is due to de Bruin [6]. Uustalu and Vene [24] explored Mendler-style folds in more depth, extending them among other things to simultaneous recursion. Mendler-style folds blend nicely with GH in that the recursion operator $(\!|-|\!)$ has the kind-indexed type of a consumer at kind $(\star \to \star) \to \star$.

The first account of the connection between STLC and cartesian closed categories was given by Lambek [16]. The specialisation to **Cat** was sketched by Gibbons and Paterson [8]. Among other things, they present a parametricity theorem for recursion operators. (Naturality is a special case of parametricity where the type takes the form of an arrow between *functors*.) A more expressive calculus building on ends and coends was defined by Cáccamo and Winskel [2]. Their paper aims at formalising informal categorical parlance such as "this isomorphism is natural in A", providing a basis for the mechanisation of categorical reasoning. The use of slice, coslice and comma categories for interpreting generic functions is to best of the authors' knowledge original.

On a related note, Gibbons and Paterson [8] have argued that AoP is more principled than GH. Briefly, their argument is that because GH works by case analysis over the structure of types, generic functions lack the coherence properties the recursion operators of AoP enjoy—folds and friends are higher-order natural transformations. The present paper shows that this is a misconception. In fact, we have seen that the two approaches nicely complement each other. Briefly, our argument in rebuttal is that AoP is concerned with only a single case (recursion), which is why coherence across cases is not an issue. Indeed, *instances* of generic functions in the sense of GH enjoy the same higher-order naturality properties as the recursion operators of AoP—sometimes simply because the generic instance *is* a recursion operator.

8. Conclusion

Category theory has been advocated for structuring definitions and theories [9]. The present paper supports this view. After the initial set-up—interpreting type terms as functors and generic functions as objects in slice categories—everything falls into place. The categorical definition of a generic function clearly exhibits its structure. For example, we observed that the instances of *crush* for colimits are just the mediating arrows. The development not only provides a semantic footing for Generic Haskell, it also suggests streamlining the language. For example, there is no need for kind-indexed types with more than two arguments. Multiple arguments or results can be handled using product categories. This approach also supports datatypes defined by mutual recursion without further ado.

References

[1] R. Bird and L. Meertens. Nested datatypes. In J. Jeuring, editor, *Fourth International Conference on Mathematics of Program Construction, MPC'98, Marstrand, Sweden*, volume 1422 of *Lecture Notes in Computer Science*, pages 52–67. Springer Berlin / Heidelberg, June 1998.

[2] M. Cáccamo and G. Winskel. A higher-order calculus for categories. In R. Boulton and P. Jackson, editors, *Theorem Proving in Higher Order Logics*, volume 2152 of *Lecture Notes in Computer Science*, pages 136–153. Springer Berlin / Heidelberg, 2001. URL `http://dx.doi.org/10.1007/3-540-44755-5_11`.

[3] D. Clarke and A. Löh. Generic Haskell, specifically. In J. Gibbons and J. Jeuring, editors, *Proceedings of the IFIP TC2 Working Conference on Generic Programming, Schloss Dagstuhl*, pages 21–48. Kluwer Academic Publishers, July 2002.

$$
\begin{aligned}
(A, C, B) \in \mathscr{R}_* &\iff A = \mathsf{Src}\ C \ \wedge\ \mathsf{Trg}\ C = B \\
(A, C, B) \in \mathscr{R}_{\mathfrak{T}\times\mathfrak{U}} &\iff (\mathsf{Outl}\ A, \mathsf{Outl}\ C, \mathsf{Outl}\ B) \in \mathscr{R}_{\mathfrak{T}} \ \wedge\ (\mathsf{Outr}\ A, \mathsf{Outr}\ C, \mathsf{Outr}\ B) \in \mathscr{R}_{\mathfrak{U}} \\
(\mathsf{F}, \mathsf{H}, \mathsf{G}) \in \mathscr{R}_{\mathfrak{T}\to\mathfrak{U}} &\iff \forall X\ Z\ Y\ .\ (X, Z, Y) \in \mathscr{R}_{\mathfrak{T}} \Longrightarrow (\mathsf{F}\ X, \mathsf{H}\ Z, \mathsf{G}\ Y) \in \mathscr{R}_{\mathfrak{U}}
\end{aligned}
$$

Figure 4. Generalising the notion of lifting to higher kinds (comma categories).

[4] D. Clarke, R. Hinze, J. Jeuring, A. Löh, and J. de Wit. The Generic Haskell user's guide. Technical Report UU-CS-2001-26, Universiteit Utrecht, November 2001.

[5] R. L. Crole. *Categories for Types*. Cambridge University Press, 1994.

[6] P. J. de Bruin. *Inductive types in constructive languages*. PhD thesis, University of Groningen, 1995.

[7] M. M. Fokkinga and L. Meertens. Adjunctions. Technical Report Memoranda Inf 94-31, University of Twente, Enschede, Netherlands, June 1994.

[8] J. Gibbons and R. Paterson. Parametric datatype-genericity. In P. Jansson, editor, *Proceedings of the 2009 ACM SIGPLAN workshop on Generic programming*, pages 85–93. ACM Press, August 2009.

[9] J. A. Goguen. A categorical manifesto. *Mathematical Structures in Computer Science*, 1:49–67, 1991. doi: doi:10.1017/S0960129500000050.

[10] R. Hinze. Polytypic values possess polykinded types. *Science of Computer Programming*, 43(2-3):129–159, May-June 2002. doi: 10.1016/S0167-6423(02)00025-4.

[11] R. Hinze. Adjoint folds and unfolds, or: Scything through the thicket of morphisms. In C. Bolduc, J. Desharnais, and B. Ktari, editors, *10th International Conference on Mathematics of Program Construction (MPC '10)*, volume 6120 of *Lecture Notes in Computer Science*, pages 195–228. Springer Berlin / Heidelberg, 2010. doi: 10.1007/978-3-642-13321-3_13.

[12] R. Hinze and J. Jeuring. Generic Haskell: Practice and theory. In R. Backhouse and J. Gibbons, editors, *Generic Programming: Advanced Lectures*, volume 2793 of *Lecture Notes in Computer Science*, pages 1–56. Springer Berlin / Heidelberg, 2003. doi: 10.1007/978-3-540-45191-4_1.

[13] R. Hinze and A. Löh. Generic programming in 3D. *Science of Computer Programming*, 74(8):590–628, June 2009. doi: 10.1016/j.scico.2007.10.006.

[14] R. Hinze, J. Jeuring, and A. Löh. Type-indexed data types. *Science of Computer Programming*, 51(1-2):117–151, May 2004. doi: 10.1016/j.scico.2003.07.001.

[15] P. Jansson and J. Jeuring. PolyP—a polytypic programming language extension. In *Conference Record 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97), Paris, France*, pages 470–482. ACM Press, January 1997.

[16] J. Lambek. From lambda-calculus to cartesian closed categories. In J. Seldin and J. Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 376–402. Academic Press, 1980.

[17] S. Mac Lane. *Categories for the Working Mathematician*. Graduate Texts in Mathematics. Springer-Verlag, Berlin, 2nd edition, 1998.

[18] G. Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14(2–3):255–280, 1990.

[19] L. Meertens. Calculate polytypically! In H. Kuchen and S. Swierstra, editors, *Proceedings 8th International Symposium on Programming Languages: Implementations, Logics, and Programs, PLILP'96, Aachen, Germany*, volume 1140 of *Lecture Notes in Computer Science*, pages 1–16. Springer Berlin / Heidelberg, September 1996.

[20] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In J. Hughes, editor, *5th ACM Conference on Functional Programming Languages and Computer Architecture, FPCA'91, Cambridge, MA, USA*, volume 523 of *Lecture Notes in Computer Science*, pages 124–144. Springer Berlin / Heidelberg, 1991.

[21] N. P. Mendler. Inductive types and type constraints in the second-order lambda calculus. *Annals of Pure and Applied Logic*, 51(1–2):159–172, 1991.

[22] J. C. Mitchell. *Foundations for Programming Languages*. The MIT Press, Cambridge, MA, 1996.

[23] A. Rodriguez, J. Jeuring, P. Jansson, A. Gerdes, O. Kiselyov, and B. C. d. S. Oliveira. Comparing libraries for generic programming in Haskell. In A. Gill, editor, *Haskell '08: Proceedings of the first ACM SIGPLAN symposium on Haskell*, pages 111–122, New York, NY, USA, 2008. ACM Press. ISBN 978-1-60558-064-7. doi: 10.1145/1411286.1411301.

[24] T. Uustalu and V. Vene. Coding recursion a la Mendler (extended abstract). In J. Jeuring, editor, *Proceedings of the 2nd Workshop on Generic Programming, Ponte de Lima, Portugal*, pages 69–85, July 2000. The proceedings appeared as a technical report of Universiteit Utrecht, UU-CS-2000-19.

A. Coproduct Laws

Uniqueness

$$f = g_1 \triangledown g_2 \iff f \cdot \mathit{inl} = g_1 \ \wedge\ f \cdot \mathit{inr} = g_2 \tag{A.1}$$

Reflection law

$$\mathit{id} = \mathit{inl} \triangledown \mathit{inr} \tag{A.2}$$

Computation law

$$(g_1 \triangledown g_2) \cdot \mathit{inl} = g_1 \tag{A.3a}$$

$$(g_1 \triangledown g_2) \cdot \mathit{inr} = g_2 \tag{A.3b}$$

Fusion law

$$k \cdot (g_1 \triangledown g_2) = k \cdot g_1 \triangledown k \cdot g_2 \tag{A.4}$$

Functor fusion law

$$(g_1 \triangledown g_2) \cdot (h_1 + h_2) = g_1 \cdot h_1 \triangledown g_2 \cdot h_2 \tag{A.5}$$

B. Product Laws

Uniqueness

$$f_1 = \mathit{outl} \cdot g \ \wedge\ f_2 = \mathit{outr} \cdot g \iff f_1 \vartriangle f_2 = g \tag{B.1}$$

Reflection law

$$\mathit{outl} \vartriangle \mathit{outr} = \mathit{id} \tag{B.2}$$

Computation law

$$f_1 = \mathit{outl} \cdot (f_1 \vartriangle f_2) \tag{B.3a}$$

$$f_2 = \mathit{outr} \cdot (f_1 \vartriangle f_2) \tag{B.3b}$$

Fusion law

$$(f_1 \vartriangle f_2) \cdot h = f_1 \cdot h \vartriangle f_2 \cdot h \tag{B.4}$$

Functor fusion law

$$(k_1 \times k_2) \cdot (f_1 \vartriangle f_2) = k_1 \cdot f_1 \vartriangle k_2 \cdot f_2 \tag{B.5}$$