Parametric Datatype-Genericity

Jeremy Gibbons

Oxford University Computing Laboratory
Wolfson Building, Parks Road
Oxford OX1 3QD, United Kingdom
http://www.comlab.ox.ac.uk/jeremy.gibbons/

Ross Paterson

School of Informatics
City University London
London EC1V 0HB, United Kingdom
http://www.soi.city.ac.uk/~ross/

Abstract

Datatype-generic programs are programs that are parametrized by a *datatype* or *type functor*: whereas polymorphic programs abstract from the 'integers' in 'lists of integers', datatype-generic programs abstract from the 'lists of'. There are two main styles of datatype-generic programming: the *Algebra of Programming* approach, characterized by structured recursion operators arising from initial algebras and final coalgebras, and the *Generic Haskell* approach, characterized by case analysis over the structure of a datatype. We show that the former enjoys a kind of *higher-order naturality*, relating the behaviours of generic functions at different types; in contrast, the latter is ad hoc, with no coherence required or provided between the various clauses of a definition. Moreover, the naturality properties arise 'for free', simply from the parametrized types of the generic functions: we present a *higher-order parametricity* theorem for datatype-generic operators.

Categories and Subject Descriptors D.3.3 [*Programming languages*]: Language constructs and features—Polymorphism, patterns, control structures, recursion; F.3.3 [*Logics and meanings of programs*]: Studies of program constructs—Program and recursion schemes, type structure; F.3.2 [*Logics and meanings of programs*]: Semantics of programming languages—Algebraic approaches to semantics; D.3.2 [*Programming languages*]: Language classifications—Functional languages.

General Terms Languages, Design, Algorithms, Theory.

Keywords Higher-order natural transformations, parametricity, free theorems, generic programming, higher-order functions, functional programming, folds, unfolds.

1. Introduction

Consider the following familiar datatype of lists, with a fold operator and a *length* function:

$$\begin{array}{l}\textbf{data}\ \mathsf{List}\ \alpha = \mathit{Nil} \mid \mathit{Cons}\ \alpha\ (\mathsf{List}\ \alpha)\\ \mathit{foldL} :: \beta \to (\alpha \to \beta \to \beta) \to \mathsf{List}\ \alpha \to \beta\\ \mathit{foldL}\ e\ f\ \mathit{Nil} = e\\ \mathit{foldL}\ e\ f\ (\mathit{Cons}\ a\ x) = f\ a\ (\mathit{foldL}\ e\ f\ x)\end{array}$$

$$\begin{array}{l}\mathit{length} :: \mathsf{List}\ \alpha \to \mathit{Int}\\ \mathit{length} = \mathit{foldL}\ 0\ (\lambda a\ n \to 1 + n)\end{array}$$

Here also is a datatype of binary trees, with its fold operator:

$$\begin{array}{l}\textbf{data}\ \mathsf{Tree}\ \alpha = \mathit{Empty} \mid \mathit{Node}\ \alpha\ (\mathsf{Tree}\ \alpha)\ (\mathsf{Tree}\ \alpha)\\ \mathit{foldT} :: \beta \to (\alpha \to \beta \to \beta \to \beta) \to \mathsf{Tree}\ \alpha \to \beta\\ \mathit{foldT}\ e\ f\ \mathit{Empty} = e\\ \mathit{foldT}\ e\ f\ (\mathit{Node}\ a\ x\ y) = f\ a\ (\mathit{foldT}\ e\ f\ x)\ (\mathit{foldT}\ e\ f\ y)\end{array}$$

One can compute the 'left spine' of a binary tree as a list, using *foldT*:

$$\begin{array}{l}\mathit{lspine} :: \mathsf{Tree}\ \alpha \to \mathsf{List}\ \alpha\\ \mathit{lspine} = \mathit{foldT}\ \mathit{Nil}\ (\lambda a\ x\ y \to \mathit{Cons}\ a\ x)\end{array}$$

The 'left depth' of a binary tree is defined to be the length of the left spine:

$$\begin{array}{l}\mathit{ldepth} :: \mathsf{Tree}\ \alpha \to \mathit{Int}\\ \mathit{ldepth} = \mathit{length} \circ \mathit{lspine}\end{array}$$

It should come as no surprise that the two steps encoded above can be fused into one, computing the left depth of the tree directly rather than via a list:

$$\begin{array}{l}\mathit{ldepth} :: \mathsf{Tree}\ \alpha \to \mathit{Int}\\ \mathit{ldepth} = \mathit{foldT}\ 0\ (\lambda a\ m\ n \to 1 + m)\end{array}$$

This result is a consequence of the well-known *fusion law* for *foldT*, which states that

$$\begin{array}{l}h \circ \mathit{foldT}\ e\ f = \mathit{foldT}\ e'\ f'\\ \quad \Longleftarrow\ h\ e = e' \wedge h\ (f\ a\ u\ v) = f'\ a\ (h\ u)\ (h\ v)\end{array}$$

Importantly, there is nothing inherently specific to binary trees involved here. The 'fold' operator is *datatype-generic*, which is to say, parametrized by a datatype, such as List or Tree; the functions *foldL* and *foldT* above are datatype-specific instances for the datatypes List and Tree, respectively. Moreover, there is a datatype-generic fusion law for folds, of which this law for *foldT* is a datatype-specific instance.

But there is more to this particular application of fusion than meets the eye. It reveals some deep structure of the datatype-generic fold operator, relating folds on binary trees and on lists. Similar relationships hold between the folds on any two datatypes. Specifically, the central observation of this paper is that:

fold is a higher-order natural transformation.

That is, 'fold' is a rather special kind of datatype-generic operator, both enjoying and requiring coherence between its datatype-specific instances. This is in contrast to many other datatype-generic operators such as parsers, pretty-printers and marshallers, for which there need be no such coherence. To the best of our knowledge, this observation has not been recorded before.

WGP'09, August 30, 2009, Edinburgh, Scotland, UK.

The situation is analogous to that between *parametric* and *ad hoc* polymorphism. For example, the parametrically polymorphic function (now with an explicit universal type quantification) $length :: \forall\alpha.\mathsf{List}\ \alpha \to Int$ amounts to a family of monomorphic functions $length_\alpha :: \mathsf{List}\ \alpha \to Int$, one for each type α. In languages satisfying parametricity, such as Girard's and Reynolds' polymorphic lambda calculi [11, 33], a *parametricity property* relates different monomorphic instances of polymorphic functions such as *length*, (a special case of) which states that:

$$length_\beta\ (map\ f\ x) = length_\alpha\ x \quad \text{for } f :: \alpha \to \beta$$

Informally, mapping over a list preserves its length. This parametricity property follows from the type of the function *length* alone; one need not look at the function definition in order to prove it [40].

In contrast, languages that provide ad hoc polymorphism do not satisfy parametricity, and hence sacrifice these parametricity properties. For instance, Harper and Morrisett [13] present a polymorphic lambda calculus extended with *intensional polymorphism*, which supports run-time type analysis through a 'typecase' construct. This allows one to define a function $countInts :: \mathsf{List}\ a \to Int$ that counts all the integers in a list, so that for example $countInts\ [4,5,6] = 3$, but $countInts\ [True, False, True] = 0$, and $countInts\ [(1,2),(3,4)] = 0$. This too can be considered as a family of monomorphic functions $countInts_\alpha :: \mathsf{List}\ \alpha \to Int$, one for each type α. But the polymorphism is ad hoc, and there need be no corresponding coherence between specific instances: the equation that would be (a special case of) the parametricity property, namely,

$$countInts_\beta\ (map\ f\ x) = countInts_\alpha\ x \quad \text{for } f :: \alpha \to \beta$$

does not hold for the intended definition of *countInts*, so it certainly does not follow from its type.

Note that Haskell's type class mechanism [12] allows one to solve many of the problems for which ad hoc polymorphism seems attractive, without sacrificing parametricity. It is still not possible — or at least, not straightforward — to write something like *countInts* above, but one can write instead a function $count :: Eq\ \alpha \Rightarrow \alpha \to \mathsf{List}\ \alpha \to Int$ that counts the occurrences of a given element in a list (in which the notation '$Eq\ \alpha \Rightarrow$' denotes a type class context, essentially a bounded quantification over the type class *Eq* of types α possessing an equality operation [22]). Using the standard compilation technique by translation into dictionary-passing style, *count* will be implemented with an additional parameter giving the equality operation, effectively having the type

$$count' :: (\alpha \to \alpha \to Bool) \to \alpha \to \mathsf{List}\ \alpha \to Int$$

This turns the apparent ad hoc polymorphism into parametric polymorphism; the parametricity property for the type of $count'$, when specialized to functions, turns out to be

$$g\ a\ a' = h\ (f\ a)\ (f\ a') \Longrightarrow \\ count'_\beta\ h\ (f\ a)\ (map\ f\ x) = count'_\alpha\ g\ a\ x$$

for any $f :: \alpha \to \beta$, $g :: \alpha \to \alpha \to Bool$, $h :: \beta \to \beta \to Bool$. This property is valid; among other consequences, when g and h are specialized to the equality functions on α and β respectively, it states that mapping with an injective function preserves element counts.

This paper is concerned with *parametric datatype-generic* operators such as fold, which obey a parametricity property analogous to that of *length* above, and contrasts them with *ad hoc* (or *intensionally*) datatype-generic operators such as parsers, pretty-printers and marshallers, which do not. Cardelli and Wegner [6] consider parametric polymorphism to be 'true polymorphism, whereas ad hoc polymorphism is some kind of apparent polymorphism whose polymorphic character disappears at close range'. We refrain from making quite so strong a statement in the context of datatype-generic programming; however, we do consider parametric datatype-genericity to be the 'gold standard' of datatype-generic programming, to be preferred over ad hoc datatype-genericity when it is available.

Our intended audience is those intrigued by 'theorems for free' [40] and 'functional programming with bananas and lenses' [29]. A little familiarity with typed lambda calculi and the notions of natural transformation and initial algebra will be helpful, although we provide most definitions as we go along.

2. Datatype-generic programming

Computer science uses a small number of highly overloaded terms, corresponding to characteristics that recur in many areas of the field. The distinction between *static* and *dynamic* aspects is one example, applying to typing, binding, IP addresses, web pages, loading of libraries, memory allocation, and program analyses, among many other things.

The term *generic* is another example, although in this case particularly within the programming languages community rather than computer science as a whole. 'Generic programming' means different things to different people: polymorphism [1], abstract datatypes [34], metaprogramming [37], and so on. We have been using it to refer to programs parametrized by a datatype such as 'lists of' or 'trees of'; but to reduce confusion, we have coined a new term *datatype-generic programming* [9] for this specific usage. Examples of datatype-generic programs are the map and fold higher-order traversal patterns of the *origami programming* style [8], and data processing tools such as pretty-printers, parsers, encoders, marshallers, comparators, and so on that form the main applications of Generic Haskell [16].

All of the operations named above can be defined once and for all, for all datatypes, rather than over and over again for each specific datatype. (However, note that although the data processing tools are all generic in the *structure* of data—a single definition can be written to cover all shapes—they are ad hoc polymorphic in the *content*—each takes arguments to handle content, explicitly or implicitly, and each type of content requires different actual parameters.) However, the two families of operations differ in the style of definition that can be provided. Roughly speaking, the origami family of datatype-generic operations have definitions that are *parametric* in the datatype parameter, whereas those of the data processor family are in general *ad hoc* in the parameter. That is to say, in the former the datatype parameter is passed around and applied, but not analysed; whereas the latter relies on a case analysis of the parameter. As a consequence, different specific instances of a datatype-generic operation from the origami family are related, whereas there is no such constraint on instances of a datatype-generic data processor.

Some instances of parametricity can be expressed in the form of natural transformations in a suitable categorical setting. In particular, the parametricity properties enjoyed by the origami operators can be expressed as natural transformations in the category of functors, or *higher-order natural transformations*. We therefore may say that the origami operations are *(higher-order) natural* in the datatype parameter. We do not mean to suggest that non-origami datatype-generic operations such as pretty-printers and parsers are 'unnatural'. However, the definition by case analysis does not provide naturality naturally: ensuring naturality requires very careful consideration of the interaction between the different cases of the definition. By analogy with Cardelli and Wegner's observations about polymorphism, we claim that higher-order naturality is at least a useful healthiness condition on datatype-generic definitions.

Short explanations of the origami and Generic Haskell styles follow; for a more detailed comparison, see [18].

2.1 Origami programming

The *Origami* [8] or *Algebra of Programming* [4] style is based around the idea that datatypes are *fixpoints* of *functors*.

We assume a cartesian closed category $\mathbb{C}$, with initial and final objects, and all ω-colimits and ω-limits; the objects of $\mathbb{C}$ model 'types', and the arrows model 'programs' between those types. We use ω-cocontinuous and ω-continuous functors (that is, those that preserve ω-colimits and ω-limits respectively) to describe the shape of recursive datatypes; among others, this includes all the polynomial functors (that is, those constructed from some base objects using products and coproducts). An example of a polynomial functor is the operation L whose action on objects is given by $\mathsf{L}\alpha = 1 + \mathit{Int} \times \alpha$, and whose action on arrows $\mathsf{L}f$ behaves as the identity on a unit (in the left of the sum), and applies f to the second component of a pair (in the right of the sum).

An F-algebra is a pair (α, f) with $f : \mathsf{F}\alpha \to \alpha$. A standard result [36] is that an ω-cocontinuous functor F possesses an initial F-algebra, which we denote $(\mu\mathsf{F}, \mathrm{IN}_\mathsf{F})$. We write $\mathrm{FOLD}_\mathsf{F}\, f$ for the witness to the initiality, the unique homomorphism from the initial algebra to an F-algebra (α, f); thus, for particular F, the type of FOLD_F itself is

$$(\mathsf{F}\alpha \to \alpha) \to (\mu\mathsf{F} \to \alpha)$$

The uniqueness of the witness to initiality is expressed in the universal property

$$h = \mathrm{FOLD}_\mathsf{F}\, f \iff h \cdot \mathrm{IN}_\mathsf{F} = f \cdot \mathsf{F}h$$

For example, $\mu\mathsf{L}$ (where L is as defined above) is the datatype of finite lists of integers; an integer-specific version of the function *length* from Section 1 can be written

$$\mathit{length} = \mathrm{FOLD}_\mathsf{L}\ (\mathit{zero} \triangledown \mathit{inc2}) : \mathsf{List}\ \mathit{Int} \to \mathit{Int}$$

where $\mathit{zero} : 1 \to \mathit{Int}$ always yields zero, $\mathit{inc2} : \mathit{Int} \times \mathit{Int} \to \mathit{Int}$ returns the successor of the second component of a pair of integers, and $\triangledown$ is the morphism combining the two branches of a sum.

Two simple consequences of the universal property are an *evaluation rule* showing how a data structure is consumed, obtained by letting $h = \mathrm{FOLD}_\mathsf{F}\, f$:

$$\mathrm{FOLD}_\mathsf{F}\, f \cdot \mathrm{IN}_\mathsf{F} = f \cdot \mathsf{F}(\mathrm{FOLD}_\mathsf{F}\, f)$$

and a *reflection rule* (sometimes called 'Lambek's Lemma') stating that folding with constructors is the identity, obtained by letting $h = \mathit{id}_{\mu\mathsf{F}}$ and $f = \mathrm{IN}_\mathsf{F}$:

$$\mathrm{FOLD}_\mathsf{F}\ \mathrm{IN}_\mathsf{F} = \mathit{id}_{\mu\mathsf{F}}$$

A more interesting consequence is the *fusion law*, for combining a FOLD with a following function:

$$h \cdot \mathrm{FOLD}_\mathsf{F}\, f = \mathrm{FOLD}_\mathsf{F}\, g \impliedby h \cdot f = g \cdot \mathsf{F}h$$

Dually, an F-coalgebra for an ω-continuous functor F is a pair (α, f) with $f : \alpha \to \mathsf{F}\alpha$; the final F-coalgebra is $(\nu\mathsf{F}, \mathrm{OUT}_\mathsf{F})$; the witness to finality, the unique homomorphism from an F-coalgebra (α, f) to the final coalgebra, is denoted $\mathrm{UNFOLD}_\mathsf{F}\, f$, whose uniqueness is expressed in the universal property

$$h = \mathrm{UNFOLD}_\mathsf{F}\, f \iff \mathrm{OUT}_\mathsf{F} \cdot h = \mathsf{F}h \cdot f$$

Consequences of the universal property include an evaluation rule:

$$\mathrm{OUT}_\mathsf{F} \cdot \mathrm{UNFOLD}_\mathsf{F}\, f = \mathsf{F}(\mathrm{UNFOLD}_\mathsf{F}\, f) \cdot f$$

a reflection rule:

$$\mathrm{UNFOLD}_\mathsf{F}\ \mathrm{OUT}_\mathsf{F} = \mathit{id}_{\nu\mathsf{F}}$$

and a fusion law for combining an UNFOLD with a preceding function:

$$\mathrm{UNFOLD}_\mathsf{F}\, f \cdot h = \mathrm{UNFOLD}_\mathsf{F}\, g \impliedby f \cdot h = \mathsf{F}h \cdot g$$

For example, the ω-cocontinuous functor S acting on objects as $\mathsf{S}\alpha = \mathit{Int} \times \alpha$ and on arrows as $\mathsf{S}f = \mathit{id}_{\mathit{Int}} \times f$ induces a datatype $\mathit{IStream} = \nu\mathsf{S}$ of infinite streams of integers. A step function $f : \alpha \to \mathsf{S}\alpha$ induces a stream producer $\mathrm{UNFOLD}_\mathsf{S}\, f : \alpha \to \mathit{IStream}$; so $\mathit{repeat} = \mathrm{UNFOLD}_\mathsf{S}\ (\mathit{id} \vartriangle \mathit{id}) : \mathit{Int} \to \mathit{IStream}$ yields a stream of copies of a given integer, where $\vartriangle$ is the morphism making a pair using two functions, and

$$\mathit{zipAdd} = \mathrm{UNFOLD}_\mathsf{S}\ \mathit{addHeads} : \mathit{IStream} \times \mathit{IStream} \to \mathit{IStream}$$

adds two integer streams element-wise, where

$$\mathit{addHeads}\ (\mathit{SCons}\ x\ xs, \mathit{SCons}\ y\ ys) = (x + y, (xs, ys))$$

(pattern-matching on a stream constructor '*SCons*' for brevity).

2.2 Generic Haskell

A different approach to datatype-generic programming involves case analyses on the structure of datatypes: 'a generic program is defined by induction on structure-representation types' [18]. We take as representative of this approach the Generic Haskell extension [14, 17, 26] of Haskell [32], based on the notion of type-indexed functions with kind-indexed types [15], in which the family of type indexes is the polynomial types (sums and products of the unit type and some basic types such as *Int*); however, our remarks apply just as well to a number of related techniques, such as the 'Scrap Your Boilerplate' series [23, 24, 25].

Consider the datatype-generic function *encode*, encoding a data structure as a list of bits. In Generic Haskell, this is defined roughly as follows. (We have made some simplifications, such as omitting cases for labels and constructors, for brevity. We have also adapted the Generic Haskell syntax slightly, for consistency with the rest of this paper; in particular, we use the list constructors from Section 1, so that we can use ':' for type or kind judgements.)

$$\begin{array}{lll}
\mathit{encode}\{\!|\alpha : \star|\!\} & : & (\mathit{encode}\{\!|\alpha|\!\}) \Rightarrow \alpha \to \mathsf{List}\ \mathit{Bool} \\
\mathit{encode}\{\!|\mathit{Unit}|\!\}\ () & = & \mathit{Nil} \\
\mathit{encode}\{\!|\mathit{Int}|\!\}\ n & = & \mathit{encodeInt}\ n \\
\mathit{encode}\{\!|\alpha :+: \beta|\!\}\ (\mathit{Inl}\ x) & = & \mathit{Cons}\ (\mathit{False}, \mathit{encode}\{\!|\alpha|\!\}\ x) \\
\mathit{encode}\{\!|\alpha :+: \beta|\!\}\ (\mathit{Inr}\ y) & = & \mathit{Cons}\ (\mathit{True}, \mathit{encode}\{\!|\beta|\!\}\ y) \\
\mathit{encode}\{\!|\alpha :\times: \beta|\!\}\ (x, y) & = & \mathit{encode}\{\!|\alpha|\!\}\ x +\!\!+ \mathit{encode}\{\!|\beta|\!\}\ y
\end{array}$$

The first line gives a type declaration, as usual; it declares that *encode* specialized to a type α of kind $\star$ has type $\alpha \to \mathsf{List}\ \mathit{Bool}$. (This does not mean that *encode* can be applied only to types of kind $\star$. Rather, the type of *encode* at a type index of another kind is derived automatically from this. For example, $\mathit{encode}\{\!|\mathsf{List}|\!\}$ has type $(\alpha \to \mathsf{List}\ \mathit{Bool}) \to (\mathsf{List}\ \alpha \to \mathsf{List}\ \mathit{Bool})$; hence the slogan that 'type-indexed functions have kind-indexed types' [15].) Moreover, the context '$(\mathit{encode}\{\!|\alpha|\!\}) \Rightarrow$' indicates that *encode* depends on itself, that is, it is defined inductively. There are cases for each possible top-level structure of the type index; the base case for integers assumes a primitive function $\mathit{encodeInt} : \mathit{Int} \to \mathsf{List}\ \mathit{Bool}$. The Generic Haskell compiler uses these five cases to derive a specialization of *encode* for any polynomial datatype, such as for the types of integer lists and binary trees introduced in Section 1.

The important point is that the behaviour of *encode* is dispersed across five separate cases, and any desired coherence between different instances has to be very carefully engineered. Recall, for example, the relationship from Section 1 between folds on binary trees and on their left spines. Is there a corresponding relationship between the encodings of binary trees and their left spines? It turns out that there is some relationship between specializations of *encode* for binary trees and for their left spines; but that relationship depends on carefully engineered interaction between the behaviours in different branches, and depends non-trivially on prefix-freeness of the encoding (a happy consequence of this particular definition) — and therefore is not a naturality property in the same sense.

To restate our point: ad hoc datatype-generic programs may in fact be higher-order natural, and perhaps higher-order naturality is a useful healthiness condition for datatype genericity; but with ad hoc techniques, this naturality requires careful design, rather than arising automatically — parametricity does not hold.

3. Higher-order functors and natural transformations

It is standard to note that one can define a category whose objects are functors from a category $\mathbb{D}$ to a category $\mathbb{E}$, with natural transformations as morphisms. We shall consider initial algebras, so we limit our attention to ω-cocontinuous functors, writing $\mathbb{D} \rightsquigarrow \mathbb{E}$ for the category of such functors. This category inherits colimits from $\mathbb{E}$, constructed pointwise. We sometimes use the standard notation $\mathbb{C}(\alpha,\beta)$ for the arrows in category $\mathbb{C}$ from object α to object β.

The next step is to consider functors to and from these functor categories, i.e. *higher-order functors* (or *hofunctors* for short). Consider the initial fixpoint operator μ, which maps functors of $\mathbb{C} \rightsquigarrow \mathbb{C}$ to objects of $\mathbb{C}$. For each natural transformation $\phi : \mathsf{F} \dot{\to} \mathsf{G}$ between ω-cocontinuous functors F and G, there is a corresponding mapping $\mu\phi : \mu\mathsf{F} \to \mu\mathsf{G}$ between the initial algebras of those functors, defined by:

$$\mu\phi = \textsc{fold}_{\mathsf{F}}\ (\textsc{in}_{\mathsf{G}} \cdot \phi_{\mu\mathsf{G}})$$

Preservation of the identity, $\mu(id_{\mathsf{F}}) = id_{\mu\mathsf{F}} : \mu\mathsf{F} \to \mu\mathsf{F}$, follows from the reflection rule, and preservation of composition

$$\mu(\phi \cdot \psi) = \mu\phi \cdot \mu\psi : \mu\mathsf{F} \to \mu\mathsf{H}$$

follows from fold fusion. Moreover μ also preserves ω-colimits. We denote arbitrary hofunctors with calligraphic capitals (such as $\mathscr{H}$).

There is a notion of 'application' appropriate for the cartesian-closed structure on $\mathbb{C}$at [27, IV.6], namely the functor $*$ from $(\mathbb{D} \rightsquigarrow \mathbb{E}) \times \mathbb{D}$ to $\mathbb{E}$ defined by:

$$\begin{aligned} \mathsf{F} * \alpha &= \mathsf{F}\alpha \\ \phi * f &= \mathsf{G}f \cdot \phi_\alpha \\ &= \phi_\beta \cdot \mathsf{F}f \end{aligned}$$

for $\phi : \mathsf{F} \dot{\to} \mathsf{G}$ and $f : \alpha \to \beta$. (For later convenience, we introduce the convention that '$*$' binds tighter than '$\cdot$'.)

3.1 Higher-order naturality

We have seen that μ is a hofunctor on $\mathbb{C}$; another such hofunctor is the $\mathscr{H}$ defined by

$$\mathscr{H}x = x * \mu x$$

The action of $\mathscr{H}$ on objects of $\mathbb{C} \rightsquigarrow \mathbb{C}$ (which are functors on $\mathbb{C}$) is

$$\mathscr{H}\mathsf{F} = \mathsf{F}(\mu\mathsf{F})$$

The action of this hofunctor on arrows of $\mathbb{C} \rightsquigarrow \mathbb{C}$ (which are natural transformations) is to take $\phi : \mathsf{F} \dot{\to} \mathsf{G}$ to an arrow

$$\mathscr{H}\phi = \phi * \mu\phi : \mathsf{F}(\mu\mathsf{F}) \to \mathsf{G}(\mu\mathsf{G})$$

of $\mathbb{C}$. Expanding the above definition of the application functor, this becomes

$$\begin{aligned} \mathscr{H}\phi &= \mathsf{G}(\mu\phi) \cdot \phi_{\mu\mathsf{F}} \\ &= \phi_{\mu\mathsf{G}} \cdot \mathsf{F}(\mu\phi) \end{aligned}$$

The lifting of the notion of functor to the functor category prompts the consideration of similarly lifting the notion of natural transformation. A *higher-order natural transformation* (or *hont* for short) on a category $\mathbb{C}$ is just the normal notion of natural transformation, specialized to the functor category $\mathbb{C} \rightsquigarrow \mathbb{C}$: for hofunctors $\mathscr{H}, \mathscr{K} : (\mathbb{C} \rightsquigarrow \mathbb{C}) \rightsquigarrow \mathbb{D}$, a hont $\textsc{op} : \mathscr{H} \ddot{\to} \mathscr{K}$ is a family of arrows $\textsc{op}_{\mathsf{F}} \in \mathbb{D}(\mathscr{H}\mathsf{F}, \mathscr{K}\mathsf{F})$ for $\mathsf{F} : \mathbb{C} \rightsquigarrow \mathbb{C}$, such that $\textsc{op}_{\mathsf{G}} \cdot \mathscr{H}\phi = \mathscr{K}\phi \cdot \textsc{op}_{\mathsf{F}}$ for $\phi : \mathsf{F} \dot{\to} \mathsf{G}$. Diagrammatically:

$$\begin{array}{ccc} \mathscr{H}\mathsf{F} & \xrightarrow{\textsc{op}_{\mathsf{F}}} & \mathscr{K}\mathsf{F} \\ {\scriptstyle \mathscr{H}\phi}\downarrow & & \downarrow{\scriptstyle \mathscr{K}\phi} \\ \mathscr{H}\mathsf{G} & \xrightarrow[\textsc{op}_{\mathsf{G}}]{} & \mathscr{K}\mathsf{G} \end{array}$$

It turns out that the constructor $\textsc{in}$ of initial algebras is a hont $\mathscr{H} \ddot{\to} \mu$, where the hofunctor $\mathscr{H}$ is as defined above. That is,

$$\textsc{in}_{\mathsf{G}} \cdot \phi_{\mu\mathsf{G}} \cdot \mathsf{F}(\mu\phi) = \mu\phi \cdot \textsc{in}_{\mathsf{F}}$$

This condition is straightforward to verify:

$$\begin{array}{ll} & \mu\phi \cdot \textsc{in}_{\mathsf{F}} \\ = & \quad \{ \text{ characterization } \mu\phi = \textsc{fold}_{\mathsf{F}}\ (\textsc{in}_{\mathsf{G}} \cdot \phi_{\mu\mathsf{G}}) \ \} \\ & \textsc{fold}_{\mathsf{F}}\ (\textsc{in}_{\mathsf{G}} \cdot \phi_{\mu\mathsf{G}}) \cdot \textsc{in}_{\mathsf{F}} \\ = & \quad \{ \ \textsc{fold} \text{ evaluation} \ \} \\ & \textsc{in}_{\mathsf{G}} \cdot \phi_{\mu\mathsf{G}} \cdot \mathsf{F}(\textsc{fold}_{\mathsf{F}}\ (\textsc{in}_{\mathsf{G}} \cdot \phi_{\mu\mathsf{G}})) \\ = & \quad \{ \text{ characterization of } \mu\phi \text{ as a } \textsc{fold} \text{ again} \ \} \\ & \textsc{in}_{\mathsf{G}} \cdot \phi_{\mu\mathsf{G}} \cdot \mathsf{F}(\mu\phi) \end{array}$$

3.2 Fold

Our key example is provided by $\textsc{fold}$. Specifically, $\textsc{fold}$ is a hont $\mathscr{H} \ddot{\to} \mathscr{K}$, where α is a fixed object of $\mathbb{C}$, and contravariant hofunctors $\mathscr{H}, \mathscr{K}$ are defined by:

$$\begin{aligned} \mathscr{H}\mathsf{F} &= (\mathsf{F}\alpha \to \alpha) \\ \mathscr{H}\phi &= (\cdot\phi_\alpha) : (\mathsf{G}\alpha \to \alpha) \to (\mathsf{F}\alpha \to \alpha) \quad \text{for } \phi : \mathsf{F} \dot{\to} \mathsf{G} \\ \mathscr{K}\mathsf{F} &= (\mu\mathsf{F} \to \alpha) \\ \mathscr{K}\phi &= (\cdot\mu\phi) : (\mu\mathsf{G} \to \alpha) \to (\mu\mathsf{F} \to \alpha) \quad \text{for } \phi : \mathsf{F} \dot{\to} \mathsf{G} \end{aligned}$$

Because the hofunctors are contravariant, some of the arrows in the higher-order naturality property get reversed:

$$\begin{array}{ccc} \mathscr{H}\mathsf{G} & \xrightarrow{\textsc{op}_{\mathsf{G}}} & \mathscr{K}\mathsf{G} \\ {\scriptstyle \mathscr{H}\phi}\downarrow & & \downarrow{\scriptstyle \mathscr{K}\phi} \\ \mathscr{H}\mathsf{F} & \xrightarrow[\textsc{op}_{\mathsf{F}}]{} & \mathscr{K}\mathsf{F} \end{array}$$

Thus, our claim induces a proof obligation

$$\textsc{fold}_{\mathsf{F}}\ (f \cdot \phi_\alpha) = \textsc{fold}_{\mathsf{G}} f \cdot \mu\phi$$

for $\phi : \mathsf{F} \dot{\to} \mathsf{G}$ and $f : \mathsf{G}\alpha \to \alpha$, which, since $\mu\phi$ is itself an instance of $\textsc{fold}$, can be discharged using fold fusion:

$$\begin{array}{ll} & \textsc{fold}_{\mathsf{G}} f \cdot \textsc{in}_{\mathsf{G}} \cdot \phi_{\mu\mathsf{G}} \\ = & \quad \{ \ \textsc{fold} \text{ evaluation} \ \} \\ & f \cdot \mathsf{G}(\textsc{fold}_{\mathsf{G}} f) \cdot \phi_{\mu\mathsf{G}} \\ = & \quad \{ \text{ naturality of } \phi \ \} \\ & f \cdot \phi_\alpha \cdot \mathsf{F}(\textsc{fold}_{\mathsf{G}} f) \end{array}$$

In particular, an integer-specific instance of the example in Section 1 has $\mathsf{T}(\alpha) = 1 + \mathit{Int} \times (\alpha \times \alpha)$, so that $\mu\mathsf{T}$ is binary trees of integers, and $\mathsf{L}\alpha = 1 + \mathit{Int} \times \alpha$ as before, so that $\mu\mathsf{L}$ is lists of integers. Choose $\phi : \mathsf{T} \dot{\to} \mathsf{L} = \mathit{id} + \mathit{id} \times \mathit{fst}$, which discards right children. Then $\mathit{lspine} = \mu\phi$, and the naturality property implies that $\mathit{ldepth} = \mathit{length} \cdot \mathit{lspine}$ as required. (For a polymorphic datatype $\mathsf{Tree}\ \alpha$, one would need to use a *bifunctor* such as $\mathsf{T}_\alpha(\beta) = 1 + \alpha \times (\beta \times \beta)$; the theory generalises smoothly in this way.)

3.3 Paramorphism

We have shown that both the datatype-generic constructor $\textsc{in}$ of initial algebras and the $\textsc{fold}$ operator enjoy a rather special property, being parametric in their shape parameters. However, there is

nothing inherently specific about FOLD and IN in this regard; many other datatype-generic operators enjoy similar properties. For example, there are generalizations of FOLD, from so-called iteration to primitive recursion. Meertens [28] captures the familiar pattern of primitive recursion by defining the *paramorphism*:

$$\text{PARA}_{\mathsf{F}} : (\mathsf{F}(\alpha \times \mu\mathsf{F}) \to \alpha) \to (\mu\mathsf{F} \to \alpha)$$

This too is a hont $\text{PARA} : \mathscr{H} \dot{\to} \mathscr{K}$ where contravariant hofunctor $\mathscr{H} : (\mathbb{C} \rightsquigarrow \mathbb{C}) \rightsquigarrow \mathbb{C}^{\text{op}}$ is given by:

$$\begin{aligned}&\mathscr{H}\mathsf{F} = (\mathsf{F}(\alpha \times \mu\mathsf{F}) \to \alpha)\\ &\mathscr{H}\phi = (\cdot(\phi * (id \times \mu\phi))) : \mathscr{H}\mathsf{G} \to \mathscr{H}\mathsf{F} \quad \text{for } \phi : \mathsf{F} \dot{\to} \mathsf{G}\end{aligned}$$

and $\mathscr{K}$ is the hofunctor used for FOLD above.

4. Co-algebraic honts

We can obtain another class of honts by dualizing the constructions of the previous section, using ω-continuous functors. In this case, we obtain an ω-continuous hofunctor $\nu : (\mathbb{C} \rightsquigarrow \mathbb{C}) \rightsquigarrow \mathbb{C}$ dualizing μ, with action on natural transformation $\phi : \mathsf{F} \dot{\to} \mathsf{G}$ defined by:

$$\nu\phi = \text{UNFOLD}_{\mathsf{G}}\ (\phi_{\nu\mathsf{F}} \cdot \text{OUT}_{\mathsf{F}}) : \nu\mathsf{F} \to \nu\mathsf{G}$$

(preservation of identities and compositions is straightforward to check). We then have a family of dual natural transformations.

4.1 Out

Dually to IN, the destructor OUT of final co-algebras is natural in its functor parameter too: $\text{OUT} : \nu \dot{\to} \mathscr{K}$ where $\nu : (\mathbb{C} \rightsquigarrow \mathbb{C}) \rightsquigarrow \mathbb{C}$ is as defined above, and covariant hofunctor $\mathscr{K} : (\mathbb{C} \rightsquigarrow \mathbb{C}) \rightsquigarrow \mathbb{C}$ is given by:

$$\begin{aligned}&\mathscr{K}\mathsf{F} = \mathsf{F}(\nu\mathsf{F})\\ &\mathscr{K}\phi = \phi * \nu\phi : \mathsf{F}(\nu\mathsf{F}) \to \mathsf{G}(\nu\mathsf{G}) \quad \text{for } \phi : \mathsf{F} \dot{\to} \mathsf{G}\end{aligned}$$

That is,

$$\mathsf{G}(\nu\phi) \cdot \phi_{\nu\mathsf{F}} \cdot \text{OUT}_{\mathsf{F}} = \text{OUT}_{\mathsf{G}} \cdot \nu\phi$$

which may again easily be verified using the definition of $\nu\phi$ and the evaluation rule of UNFOLD.

4.2 Unfold

Similarly, the UNFOLD operator dualizing FOLD is a hont $\mathscr{H} \dot{\to} \mathscr{K}$ where covariant hofunctors $\mathscr{H}, \mathscr{K} : (\mathbb{C} \rightsquigarrow \mathbb{C}) \rightsquigarrow \mathbb{C}$ are given by:

$$\begin{aligned}&\mathscr{H}\mathsf{F} = (\alpha \to \mathsf{F}\alpha)\\ &\mathscr{H}\phi = (\phi_\alpha \cdot) : (\alpha \to \mathsf{F}\alpha) \to (\alpha \to \mathsf{G}\alpha) \quad \text{for } \phi : \mathsf{F} \dot{\to} \mathsf{G}\\ &\mathscr{K}\mathsf{F} = (\alpha \to \nu\mathsf{F})\\ &\mathscr{K}\phi = (\nu\phi \cdot) : (\alpha \to \nu\mathsf{F}) \to (\alpha \to \nu\mathsf{G}) \quad \text{for } \phi : \mathsf{F} \dot{\to} \mathsf{G}\end{aligned}$$

The higher-order naturality amounts to the claim that

$$\text{UNFOLD}_{\mathsf{G}}\ (\phi_\alpha \cdot f) = \nu\phi \cdot \text{UNFOLD}_{\mathsf{F}}\ f$$

for $f : \alpha \to \mathsf{F}\alpha$; by definition of $\nu\phi$ and unfold fusion, it suffices to show

$$\phi_{\nu\mathsf{F}} \cdot \text{OUT}_{\mathsf{F}} \cdot \text{UNFOLD}_{\mathsf{F}}\ f = \mathsf{G}(\text{UNFOLD}_{\mathsf{F}}\ f) \cdot \phi_\alpha \cdot f$$

which follows from the naturality of ϕ and the evaluation rule for UNFOLD:

$$\begin{aligned}&\phi_{\nu\mathsf{F}} \cdot \text{OUT}_{\mathsf{F}} \cdot \text{UNFOLD}_{\mathsf{F}}\ f\\ =\quad &\{\quad \text{UNFOLD evaluation} \quad\}\\ &\phi_{\nu\mathsf{F}} \cdot \mathsf{F}(\text{UNFOLD}_{\mathsf{F}}\ f) \cdot f\\ =\quad &\{\quad \text{naturality of } \phi \quad\}\\ &\mathsf{G}(\text{UNFOLD}_{\mathsf{F}}\ f) \cdot \phi_\alpha \cdot f\end{aligned}$$

(Note that the higher-order naturality of UNFOLD is simpler than that of FOLD, because it does not involve contravariance. Perhaps UNFOLD should be better appreciated [10] — even considered the 'ordinary' case, and FOLD its dual?)

As an application of the higher-order naturality of UNFOLD, consider Pascal's Triangle:

```
          1
        1   1
      1   2   1
    1   3   3   1
  1   4   6   4   1
1   5  10  10   5   1
...
```

The triangular shape can expressed as the final coalgebra of the bifunctor U defined by $\mathsf{U}_\alpha = \mathit{Int} \times \mathit{IStream} \times \mathit{IStream} \times \alpha$, which gives a vertex of type *Int*, two infinite edges of type *IStream Int*, and an inner structure of type α. Pascal's Triangle itself is $\text{UNFOLD}_{\mathsf{U}}\ \mathit{step}\ (\mathit{repeat}\ 1)$, where

$$\begin{aligned}&\mathit{step}\ (\mathit{SCons}\ x\ (\mathit{SCons}\ y\ \mathit{xs}))\\ &\quad = \textbf{let}\ \mathit{zs} = \mathit{SCons}\ (2 \times y)\ (\mathit{zipAdd}\ (\mathit{xs}, \mathit{zs}))\\ &\qquad \textbf{in}\ \ (x, \mathit{SCons}\ y\ \mathit{xs}, \mathit{SCons}\ y\ \mathit{xs}, \mathit{zs})\end{aligned}$$

(That is, the seed of the unfold is one of the infinite edges. The initial seed is an infinite stream of ones, and the seed evolves according to the definition of *zs* above.)

Pascal's Triangle has many nice properties. One of them is that the *n*th element of the central column $1, 2, 6, 20...$ is the number of: non-decreasing sequences of n integers drawn from $0 \ldots n$; direct routes on a grid making n steps East and n steps North in total; directed, convex polyominoes having semiperimeter $n+2$; and so on [35, Sequence A000984]. Extraction of this middle column is achieved by $\nu\phi$, where the natural transformation $\phi : \mathsf{U} \dot{\to} S$ is defined by $\phi\ (x, ys, zs, u) = (x, u)$. By higher-order naturality of UNFOLD,

$$\nu\phi \cdot \text{UNFOLD}_{\mathsf{U}}\ \mathit{step} = \text{UNFOLD}_{S}\ \mathit{step}'$$

where

$$\begin{aligned}&\mathit{step}'\ (\mathit{SCons}\ x\ (\mathit{SCons}\ y\ \mathit{xs}))\\ &\quad = \textbf{let}\ \mathit{zs} = \mathit{SCons}\ (2 \times y)\ (\mathit{zipAdd}\ (\mathit{xs}, \mathit{zs}))\\ &\qquad \textbf{in}\ \ (x, \mathit{zs})\end{aligned}$$

This yields a direct method of computing Sequence A000984, without having to generate Pascal's Triangle first.

4.3 Apomorphism

As in the initial algebra case, there is a further family of operators expressing various forms of co-iteration and primitive co-recursion, all of which are also honts. Uustalu and Vene [39] dualize Meertens' paramorphisms by defining the *apomorphism*:

$$\text{APO}_{\mathsf{F}} : (\alpha \to \mathsf{F}(\alpha + \nu\mathsf{F})) \to (\alpha \to \nu\mathsf{F})$$

This is another hont $\text{APO} : \mathscr{H} \dot{\to} \mathscr{K}$ where covariant hofunctor $\mathscr{H} : (\mathbb{C} \rightsquigarrow \mathbb{C}) \rightsquigarrow \mathbb{C}$ is given by:

$$\begin{aligned}&\mathscr{H}\mathsf{F} = (\alpha \to \mathsf{F}(\alpha + \nu\mathsf{F}))\\ &\mathscr{H}\phi = ((\phi * (id + \nu\phi))\cdot) : \mathscr{H}\mathsf{F} \to \mathscr{H}\mathsf{G} \quad \text{for } \phi : \mathsf{F} \dot{\to} \mathsf{G}\end{aligned}$$

and $\mathscr{K}$ is the hofunctor used for UNFOLD above.

5. Non-honts

This large collection of higher-order natural datatype-generic operations begs a question: is every datatype-generic operator a hont? Certainly not! In fact, the Generic Haskell function *encode* of Section 2.2 is a counterexample. If it were a hont, it would have to be of the form

$$\mathit{encode}_{\mathsf{F}} : \mu\mathsf{F} \to \mathsf{List}\ \mathit{Bool}$$

This is of the right type to be a hont $\mu \xrightarrow{\cdot} \mathscr{K}$, where $\mathscr{K}$ is the constant hofunctor defined by $\mathscr{K}\,\mathsf{F} = \mathsf{List}\ Bool$ and $\mathscr{K}\,\phi = id_{\mathsf{List}\ Bool}$. However, the naturality property corresponding to any $\mathrm{OP} : \mu \xrightarrow{\cdot} \mathscr{K}$ reduces to

$$\mathrm{OP}_{\mathsf{G}} \cdot \mu\phi = \mathrm{OP}_{\mathsf{F}}$$

for $\phi : \mathsf{F} \xrightarrow{\cdot} \mathsf{G}$ — that is, all naturally related data structures (such as a binary tree and its left spine) are equivalent under OP. In particular, if G is the constant functor defined by $\mathsf{G}\alpha = 1$ and $\mathsf{G}f = id_1$, then for any F there is a unique natural transformation ϕ from F to G and OP_{F} factors through OP_{G}. That is, OP_{F} must have the same constant value for any F, which is clearly not what is wanted.

The same applies to the generic size operation that counts the (integer) values in a data structure — another standard example in the Generic Haskell literature [17]:

$$\begin{array}{ll}
size\{\!|\alpha : \star|\!\} & : (size\{\!|\alpha|\!\}) \Rightarrow \alpha \to Int \\
size\{\!|Unit|\!\}\ () & = 0 \\
size\{\!|Int|\!\}\ n & = 1 \\
size\{\!|\alpha :+: \beta|\!\}\ (Inl\ x) & = size\{\!|\alpha|\!\}\ x \\
size\{\!|\alpha :+: \beta|\!\}\ (Inr\ y) & = size\{\!|\beta|\!\}\ y \\
size\{\!|\alpha :\times: \beta|\!\}\ (x,y) & = size\{\!|\alpha|\!\}\ x + size\{\!|\beta|\!\}\ y
\end{array}$$

As with *encode*, a hont of this type would have to be constant; all structures would have the same size.

The difference here is that Generic Haskell is based on *case analysis* of the shape, rather than blind *application* of the shape parameter; it therefore allows completely different behaviours in different branches of the analysis. This gives greater flexibility (as a corollary to the above, generic encoding and generic size cannot be defined in the Algebra of Programming style), but with that greater power comes greater responsibility.

6. Higher-order theorems for free

The statement that PARA is a hont can be proven from first principles, or using the higher-order naturality of FOLD and IN, in terms of which PARA is defined. However, by analogy with the first-order case [33, 40], one might expect to be able to deduce the result 'for free' from the type of PARA, using a kind of second-order parametricity theorem [30]. As in the first-order case, naturality is the special case of parametricity where the type takes the form of a morphism between functors. In this section, we present such a theorem.

Our method is to define a small language in which initial algebra operators like PARA may be defined, and to prove a parametricity theorem for terms in this language. Though we present only the initial algebra case, the same method can also be applied to define a language for final co-algebra operators.

Our language has two parts: a typed lambda calculus, and a notation for categorical combinators.

6.1 Lambda-calculus of types

The first part is a conventional typed λ-calculus as shown in Figure 1. We shall interpret this calculus using the cartesian-closed structure on ω-cocomplete categories and ω-cocontinuous functors. Thus kinds are interpreted as categories:

$$\begin{array}{ll}
\mathbb{C}_{\star} & = \mathbb{C} \\
\mathbb{C}_{\kappa_1 \Rightarrow \kappa_2} & = \mathbb{C}_{\kappa_1} \rightsquigarrow \mathbb{C}_{\kappa_2}
\end{array}$$

This interpretation is readily extended to type contexts:

$$\mathbb{C}_{X_1 :: \kappa_1, \ldots, X_n :: \kappa_n} = \mathbb{C}_{\kappa_1} \times \cdots \times \mathbb{C}_{\kappa_n}$$

The objects of the category $\mathbb{C}_{\Delta}$ are type environments δ. That is, each δ in $\mathbb{C}_{\Delta}$ is a family of objects, i.e. a function mapping type variables $X :: \kappa$ to objects of $\mathbb{C}_{\kappa}$. The arrows of this category are indexed transformations $\tau \in \mathbb{C}_{\Delta}(\delta, \delta')$, so that $\tau\, X : \mathbb{C}_{\kappa}\ (\delta\, X, \delta'\, X)$.

Raw syntax

Kinds	κ	$\star \mid \kappa_1 \Rightarrow \kappa_2$
Types	T	$K \mid X \mid \Lambda X :: \kappa.\ T \mid T_1\ T_2$
Type contexts	Δ	$X_1 :: \kappa_1, \ldots, X_n :: \kappa_n$

Constants

$$\begin{array}{ll}
1 & :: \star \\
(+) & :: \star \Rightarrow \star \Rightarrow \star \\
(\times) & :: \star \Rightarrow \star \Rightarrow \star \\
\mu & :: (\star \Rightarrow \star) \Rightarrow \star
\end{array}$$

Type judgements $\Delta \vdash T :: \kappa$

$$\Delta \vdash K :: \kappa \quad (K :: \kappa \in Sig) \qquad\qquad \Delta, X :: \kappa \vdash X :: \kappa$$

$$\frac{\Delta, X :: \kappa_1 \vdash T :: \kappa_2}{\Delta \vdash \Lambda X :: \kappa_1.\ T :: \kappa_1 \Rightarrow \kappa_2} \qquad\qquad \frac{\Delta \vdash T_1 :: \kappa_1 \Rightarrow \kappa_2 \quad \Delta \vdash T_2 :: \kappa_1}{\Delta \vdash T_1\ T_2 :: \kappa_2}$$

Figure 1. Lambda-calculus of types

We assume for each constant $K :: \kappa$ an interpretation $[\![K]\!] \in |\mathbb{C}_{\kappa}|$. Indeed we have used the same names for the constants in Figure 1 as the corresponding objects in the metalanguage. Thus $[\![1]\!]$ is the object of $\mathbb{C}$, while $[\![+]\!]$ and $[\![\times]\!]$ are the corresponding binary functors $\mathbb{C} \rightsquigarrow \mathbb{C} \rightsquigarrow \mathbb{C}$ and $[\![\mu]\!]$ is the functor $\mu : (\mathbb{C} \rightsquigarrow \mathbb{C}) \rightsquigarrow \mathbb{C}$. That is, μ maps functors to objects, and natural transformations to ordinary morphisms. Note that the calculus has no constant $\to$ for function types, as this does not correspond to a covariant functor. Function types will be handled specially by the next layer of the calculus in the next section.

The interpretation of a type judgement $\Delta \vdash T :: \kappa$ is standard [7], but specialised to the cartesian closed structure of this category of categories, so that it defines a functor:

$$[\![\Delta \vdash T :: \kappa]\!] \in \mathbb{C}_{\Delta} \rightsquigarrow \mathbb{C}_{\kappa}$$

This functor maps an object environment $\delta \in |\mathbb{C}_{\Delta}|$ to an object

$$[\![\Delta \vdash T :: \kappa]\!]\delta \in |\mathbb{C}_{\kappa}|$$

and a transformation of environments $\tau \in \mathbb{C}_{\Delta}(\delta, \delta')$ to

$$[\![\Delta \vdash T :: \kappa]\!]\tau \in \mathbb{C}_{\kappa}([\![\Delta \vdash T :: \kappa]\!]\delta, [\![\Delta \vdash T :: \kappa]\!]\delta')$$

The interpretation of application is the functor $*$ introduced in Section 3.

6.2 A language for initial algebra combinators

The second part of the language allows us to express categorical combinators. The syntax and type rules of our language are given in Figure 2.

The signature *Sig* consists of the kinded type constants $T :: \kappa$ of Figure 1, plus polymorphic constants $k : \Lambda X_i :: \kappa_i.\ S$, where $X_1 :: \kappa_1, \ldots X_n :: \kappa_n \vdash S$. We assume a fixed signature, containing the usual polymorphic constants for 1, $+$, and $\times$, as well as higher-order polymorphic constants:

$$\begin{array}{ll}
\mathrm{IN} & : \Lambda \mathsf{F} :: \star \Rightarrow \star.\ \mathsf{F}(\mu\mathsf{F}) \to \mu\mathsf{F} \\
\mathrm{FOLD} & : \Lambda \mathsf{F} :: \star \Rightarrow \star, A :: \star.\ (\mathsf{F}A \to A) \to (\mu\mathsf{F} \to A) \\
\mathrm{MAP} & : \Lambda \mathsf{F} :: \star \Rightarrow \star, A :: \star, B :: \star.\ (A \to B) \to (\mathsf{F}A \to \mathsf{F}B)
\end{array}$$

Our language is somewhat restrictive; in particular it can express higher-order functions on data structures, but cannot express data structures containing functions. However, it is sufficient for defining

Raw syntax

Function types	S	$T \mid S_1 \to S_2$
Expressions	e	$k_{S_1,\ldots,S_n} \mid x \mid \lambda x\!:\!S.\ e \mid e_1\ e_2$
Expression contexts	Γ	$x_1 : S_1, \ldots, x_n : S_n$

Function type judgements $\Delta \vdash S$

$$\frac{\Delta \vdash T :: \star}{\Delta \vdash T} \qquad \frac{\Delta \vdash S_1 \quad \Delta \vdash S_2}{\Delta \vdash S_1 \to S_2}$$

Expression judgements $\Delta;\Gamma \vdash e : T$

$$\frac{\Delta \vdash \Gamma \quad \Delta \vdash T_i :: \kappa_i}{\Delta;\Gamma \vdash k_{T_1,\ldots,T_n} : S[T_i/X_i]} \quad (k : \Lambda X_i :: \kappa_i.\ S \in Sig)$$

$$\frac{\Delta \vdash \Gamma \quad \Delta \vdash S}{\Delta;\Gamma,x:S \vdash x : S} \qquad \frac{\Delta;\Gamma,x:S_1 \vdash e : S_2}{\Delta;\Gamma \vdash \lambda x:S_1.\ e : S_1 \to S_2}$$

$$\frac{\Delta;\Gamma \vdash e_1 : S_1 \to S_2 \quad \Delta;\Gamma \vdash e_2 : S_1}{\Delta;\Gamma \vdash e_1\ e_2 : S_2} \qquad \frac{\Delta;\Gamma \vdash e : S_1 \quad \Delta \vdash S_1 =_{\beta\eta} S_2}{\Delta;\Gamma \vdash e : S_2}$$

Figure 2. λ-calculus of operators

the various initial algebra operators. For example, the PARA operator of Section 3.3 can be expressed as

$$\begin{array}{l}\mathsf{F} :: \star \Rightarrow \star, A :: \star; \vdash \\ \quad \lambda f : \mathsf{F}(A \times \mu\mathsf{F}) \to A. \\ \quad\quad fst \cdot \mathrm{FOLD}_{\mathsf{F}}\ (\lambda t : \mathsf{F}(A \times \mu\mathsf{F}). \\ \quad\quad\quad\quad (f\ t, \mathrm{IN}_{\mathsf{F}}\ (\mathrm{MAP}_{\mathsf{F}}\ snd\ t))) : \\ \quad (\mathsf{F}(A \times \mu\mathsf{F}) \to A) \to (\mu\mathsf{F} \to A)\end{array}$$

Next we shall define a parametric interpretation of our language, through which polymorphic types like the above yield parametricity properties, including higher-order naturality.

6.3 Interpretation of function types and expressions

We can interpret function type judgements $\Delta \vdash S$ straightforwardly as mappings of objects:

$$\begin{array}{ll}\mathfrak{O}[\![\Delta \vdash S]\!] & \in |\mathbb{C}_\Delta| \to |\mathbb{C}| \\ \mathfrak{O}[\![\Delta \vdash T]\!]\delta & = [\![\Delta \vdash T :: \star]\!]\delta \\ \mathfrak{O}[\![\Delta \vdash S_1 \to S_2]\!]\delta & = \mathfrak{O}[\![\Delta \vdash S_1]\!]\delta \to \mathfrak{O}[\![\Delta \vdash S_2]\!]\delta\end{array}$$

However we cannot define a mapping of arrows, because the function type constructor $\to$ is not covariant in both arguments. We shall instead interpret a judgement $\Delta \vdash S$ as mapping arrows of $\mathbb{C}_\Delta$ to relations between the corresponding objects of $\mathbb{C}$. Although relations support composition, this composition is not preserved by the semantic mapping, so we shall not use a category of relations, and the mapping will not be a functor.

To define relations, we need to identify a set of points of each object of $\mathbb{C}$. We define the functor $\mathsf{Pt} : \mathbb{C} \rightsquigarrow \mathbb{S}\mathrm{et}$ by $\mathsf{Pt} = \mathbb{C}(1,\text{-})$. The corresponding notion of application takes $f \in \mathsf{Pt}\ (A \to B)$ and $a \in \mathsf{Pt}\ A$ and yields $f \mathbin{@} a \in \mathsf{Pt}\ B$. We assume that $\mathbb{C}$ is well-pointed, that is, equality of arrows, $f = g$, reduces to equality on points, $\forall a \in \mathbb{C}(1,A).\ f \mathbin{@} a = g \mathbin{@} a$.

Then the relational interpretation follows the usual definition of a logical relation. For $\tau \in \mathbb{C}(\delta,\delta')$, we define

$$\begin{array}{ll}\mathfrak{R}[\![\Delta \vdash S]\!]\tau & \in \mathbb{R}\mathrm{el}(\mathsf{Pt}(\mathfrak{O}[\![\Delta \vdash S]\!]\delta), \mathsf{Pt}(\mathfrak{O}[\![\Delta \vdash S]\!]\delta')) \\ \mathfrak{R}[\![\Delta \vdash T]\!]\tau\ (p,p') & \equiv [\![\Delta \vdash T :: \star]\!]\tau \cdot p = p' \\ \mathfrak{R}[\![\Delta \vdash S_1 \to S_2]\!]\tau\ (f,f') & \equiv \forall p,p'.\ \mathfrak{R}[\![\Delta \vdash S_1]\!]\tau\ (p,p') \Longrightarrow \\ & \qquad \mathfrak{R}[\![\Delta \vdash S_2]\!]\tau\ (f \mathbin{@} p, f' \mathbin{@} p')\end{array}$$

For each constant $k : \Lambda\Delta.\ S$, we assume for each $\delta \in |\mathbb{C}_\Delta|$ a point

$$[\![k]\!]\delta \in \mathsf{Pt}\ (\mathfrak{O}[\![\Delta \vdash S]\!])$$

such that for $\tau \in \mathbb{C}(\delta,\delta')$ we have $\mathfrak{R}[\![\Delta \vdash S]\!]\tau\ ([\![k]\!]\delta, [\![k]\!]\delta')$.

For example, for the constant IN, the type

$$\mathrm{IN} : \Lambda\mathsf{F} :: \star \Rightarrow \star.\ \mathsf{F}(\mu\mathsf{F}) \to \mu\mathsf{F}$$

implies the property

$$\forall t,t'.\ \phi * \mu\phi \cdot t = t' \Longrightarrow \mu\phi \cdot \mathrm{IN}_{\mathsf{F}} \cdot t = \mathrm{IN}_{\mathsf{G}} \cdot t'$$

(recall that '$*$' binds tighter than '$\cdot$'), or equivalently

$$\forall t.\ \mu\phi \cdot \mathrm{IN}_{\mathsf{F}} \cdot t = \mathrm{IN}_{\mathsf{G}} \cdot \phi * \mu\phi \cdot t$$

As $\mathbb{C}$ is well-pointed, this is equivalent to

$$\mu\phi \cdot \mathrm{IN}_{\mathsf{F}} = \mathrm{IN}_{\mathsf{G}} \cdot \phi * \mu\phi$$

which is exactly the higher-order naturality property of IN.

As a second example, for the constant

$$\mathrm{FOLD} : \Lambda\mathsf{F} :: \star \Rightarrow \star, A :: \star.\ (\mathsf{F}A \to A) \to (\mu\mathsf{F} \to A)$$

the property implied by the type reduces similarly to

$$\forall f,f'.\ a \cdot f = f' \cdot \phi * a \Longrightarrow a \cdot \mathrm{FOLD}_{\mathsf{F}}\ f = \mathrm{FOLD}_{\mathsf{G}}\ f' \cdot \mu\phi$$

which combines the higher-order naturality property of FOLD with the first-order fusion law.

As a third example, for the constant

$$\mathrm{MAP} : \Lambda\mathsf{F} :: \star \Rightarrow \star, A :: \star, B :: \star.\ (A \to B) \to (\mathsf{F}A \to \mathsf{F}B)$$

the property implied by the type reduces similarly to

$$\forall f,f'.\ b \cdot f = f' \cdot a \Longrightarrow \phi * b \cdot \mathrm{MAP}_{\mathsf{F}}\ f = \mathrm{MAP}_{\mathsf{G}}\ f' \cdot \phi * a$$

Since $\mathrm{MAP}_{\mathsf{F}}$ represents the action of F on arrows, this reduces to

$$\forall f.\ \phi_{A'} \cdot \mathsf{F}f = \mathsf{G}f \cdot \phi_A$$

which is the statement of the naturality of ϕ. Note that as F occurs in both positive and negative positions in the type of MAP, the parametricity property does not correspond to naturality in F, just as FOLD is parametric but not natural in A.

For each $\Gamma;\Delta \vdash e : S$, we have an interpretation of expressions

$$[\![\Gamma;\Delta \vdash e : S]\!]\delta \in \mathbb{C}(\mathfrak{O}[\![\Delta \vdash \Gamma]\!]\delta, \mathfrak{O}[\![\Delta \vdash S]\!]\delta)$$

using the cartesian closed structure of $\mathbb{C}$ in the usual way. In the special case where the expression e is closed, the expression context Γ will be empty, so that the semantics is an arrow in $\mathbb{C}(1, \mathfrak{O}[\![\Delta \vdash S]\!]\delta)$, that is, a point.

6.4 Consequences of the interpretation

Assuming these properties for the primitive constants, our aim is to infer similar properties for other combinators that we can define in our language.

This is expressed by our main theorem: if τ is a transformation between type environments δ and δ', such that the corresponding relation holds between value environments γ and γ', then the interpretation of $\Delta;\Gamma \vdash e : S$ with respect to δ and γ is related to the interpretation with respect to δ' and γ'. As with the corresponding result for the polymorphic λ-calculus, this is established by a straightforward induction over the derivation of $\Delta;\Gamma \vdash e : S$.

Theorem 1 Given

$$\begin{array}{l}\tau \in \mathbb{C}_\Delta(\delta,\delta') \\ \gamma \in \mathsf{Pt}(\mathfrak{O}[\![\Delta \vdash \Gamma]\!]\delta) \\ \gamma' \in \mathsf{Pt}(\mathfrak{O}[\![\Delta \vdash \Gamma]\!]\delta')\end{array}$$

we have

$$\mathfrak{R}[\![\Delta \vdash \Gamma]\!]\,\tau\,(\gamma,\gamma') \Longrightarrow \\ \mathfrak{R}[\![\Delta \vdash S]\!]\,\tau\,([\![\Delta;\Gamma \vdash e:S]\!]\,\delta\cdot\gamma,[\![\Delta;\Gamma \vdash e:S]\!]\,\delta'\cdot\gamma')$$

□

When e is closed, things are much simpler. Γ will be empty, $\mathfrak{O}[\![\Delta \vdash \Gamma]\!]\,\delta$ and $\mathfrak{O}[\![\Delta \vdash \Gamma]\!]\,\delta'$ will both be 1, so the relation between them trivially holds.

Corollary 2 For any closed term $\Delta;\vdash e:S$ and $\tau \in \mathbb{C}_\Delta(\delta,\delta')$, we have

$$\mathfrak{R}[\![\Delta \vdash S]\!]\,\tau\,([\![\Delta;\vdash e:S]\!]\,\delta,[\![\Delta;\vdash e:S]\!]\,\delta')$$

□

For example, the PARA operator defined above had the type

$$\mathsf{F}::\star\Rightarrow\star, A::\star;\vdash \ldots:(\mathsf{F}(A\times\mu\mathsf{F})\to A)\to(\mu\mathsf{F}\to A)$$

From this type we can infer the property

$$\forall f,f'.\ a\cdot f=f'\cdot\phi*(a\times\mu\phi)\Longrightarrow a\cdot\mathrm{PARA}_{\mathsf{F}}\,f=\mathrm{PARA}_{\mathsf{G}}\,f'\cdot\mu\phi$$

That is, we get for free the higher-order naturality property presented in Section 3.3, combined with a first-order fusion law; specializing to $a=id$ yields the higher-order naturality alone.

7. Conclusions

We have shown that the datatype-generic FOLD operator enjoys a kind of higher-order naturality property, relating different instances connected by a natural transformation; moreover, that naturality property arises for free from the type of the operator. Similar results apply to many other datatype-generic operators in the origami programming style. Moreover, those results may be derived for free from the higher-kinded type of the operator. In contrast, in approaches to datatype-generic programming that rely on case analysis on the shape of data, such properties have to be much more carefully engineered.

7.1 Inspiration

Our inspiration for the higher-order natural transformations described in this paper is Paul Hoogendijk's very elegant work with Roland Backhouse [19, 20] on generic 'zip' functions $\mathrm{ZIP}_{\mathsf{F},\mathsf{G}}:\mathsf{F}\circ\mathsf{G}\mathrel{\dot{\to}}\mathsf{G}\circ\mathsf{F}$ that commute or transpose two functors F and G. The general case requires a relational setting, because such a transposition might be partial (for instance, yielding no result on mismatched data structures, such as when zipping a pair of lists of differing length) and non-deterministic (for instance, yielding results of arbitrary shape, such as when zipping with a constant functor F). However, the essence of the idea can be seen from considering the special case when G corresponds to a fixed-shape datatype such as Pair. In this case, the partially-parametrized remainder is conventionally called a 'generic unzip' $\mathrm{UNZIP}_{\mathsf{F}}:\mathsf{F}\circ\mathsf{Pair}\mathrel{\dot{\to}}\mathsf{Pair}\circ\mathsf{F}$, and can easily be given an explicit definition:

$$\mathrm{UNZIP}_{\mathsf{F}}=\mathsf{F}\mathit{fst}\vartriangle\mathsf{F}\mathit{snd}$$

where *fst* and *snd* are the projections from Pair, and $\vartriangle$ generates a Pair using two element-generating functions. It is not hard to show that UNZIP is a hont $(\circ\mathsf{Pair})\mathrel{\dot{\to}}(\mathsf{Pair}\circ)$. The higher-order naturality property arising from this observation was used in [5] to transform an $O(n\log n)$-time algorithm for computing bit-reversal permutations to $O(n)$-time.

7.2 Related work

The equation capturing the higher-order naturality of FOLD has appeared before; for example, it is Theorem 6.12 of the second author's PhD thesis [31], Equation 29 in the influential 'bananas paper' [29], and Equation (4) of [38]. However, to the best of our knowledge, the fact that this property is a naturality property has not been noted previously. Although Hoogendijk [19] coins the term 'parametric polytypism' (for what we call 'parametric datatype-genericity'), and contrasts it with 'ad hoc polytypism', the only hont he considers is the generic ZIP described above. In particular, although Hoogendijk certainly makes use of FOLD, there is no evidence from his writing that he realised that it too was a hont. The closest observation we are aware of in the literature is an offhand remark ('catamorphisms on different types can be related, but the precise details are not clear to us') by Jeuring and Jansson [21]. The extension to other datatype-generic operators such as UNFOLD, and the higher-order parametricity result itself, appear to be novel, albeit perhaps not very surprising with the benefit of hindsight.

7.3 Acknowledgements

We would like to thank the members of the Algebra of Programming group at Oxford, for helpful discussions on this work; in particular, Bruno Oliveira showed us how to carry out the fusion in Example 4.2. Philip Wadler, Patty Johann, Neil Ghani, and the anonymous referees all provided advice that improved our presentation.

References

[1] Matthew H. Austern. *Generic Programming and the STL*. Addison-Wesley, 1999.

[2] Roland Backhouse and Jeremy Gibbons, editors. *Summer School on Generic Programming*, volume 2793 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.

[3] Roland Backhouse, Jeremy Gibbons, Ralf Hinze, and Johan Jeuring, editors. *Spring School on Datatype-Generic Programming*, volume 4719 of *Lecture Notes in Computer Science*. Springer-Verlag, 2007.

[4] Richard Bird and Oege de Moor. *Algebra of Programming*. Prentice-Hall, 1996.

[5] Richard Bird, Jeremy Gibbons, and Geraint Jones. Program optimisation, naturally. In J. W. Davies, A. W. Roscoe, and J. C. P. Woodcock, editors, *Millenial Perspectives in Computer Science*. Palgrave, 2000.

[6] Luca Cardelli and Peter Wegner. On understanding types, data abstraction and polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

[7] Roy L. Crole. *Categories for Types*. Cambridge University Press, 1994.

[8] Jeremy Gibbons. Origami programming. In Jeremy Gibbons and Oege de Moor, editors, *The Fun of Programming*, Cornerstones in Computing, pages 41–60. Palgrave, 2003.

[9] Jeremy Gibbons. Datatype-generic programming. In Backhouse et al. [3].

[10] Jeremy Gibbons and Geraint Jones. The under-appreciated unfold. In *International Conference on Functional Programming*, pages 273–279, Baltimore, Maryland, September 1998.

[11] Jean-Yves Girard. *Interprétation Fonctionnelle et Élimination des Coupures de l'Arithmétique d'Ordre Supérieur*. PhD thesis, Université de Paris VII, 1972.

[12] Cordelia Hall, Kevin Hammond, Simon Peyton Jones, and Philip Wadler. Type classes in Haskell. *ACM Transactions on Programming Languages and Systems*, 18(2):19–138, 1996.

[13] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *POPL*, 1995.

[14] Ralf Hinze. A generic programming extension for Haskell. In Erik Meijer, editor, *Third Haskell Workshop*, 1999.

[15] Ralf Hinze. Polytypic values possess polykinded types. In Roland Backhouse and José Nuno Oliveira, editors, *Mathematics of Program*

Construction, volume 1837 of *Lecture Notes in Computer Science*, pages 2–27. Springer-Verlag, 2000.

[16] Ralf Hinze and Johan Jeuring. Generic Haskell: Applications. In Backhouse and Gibbons [2], pages 57–97.

[17] Ralf Hinze and Johan Jeuring. Generic Haskell: Practice and theory. In Backhouse and Gibbons [2], pages 1–56.

[18] Ralf Hinze, Johan Jeuring, and Andres Löh. Comparing approaches to generic programming in Haskell. In Backhouse et al. [3].

[19] Paul Hoogendijk. *A Generic Theory of Datatypes*. PhD thesis, Technische Universiteit Eindhoven, 1997.

[20] Paul Hoogendijk and Roland Backhouse. When do datatypes commute? In Eugenio Moggi and Guiseppe Rosolini, editors, *Category Theory and Computer Science*, volume 1290 of *Lecture Notes in Computer Science*, pages 242–260. Springer-Verlag, September 1997.

[21] Johan Jeuring and Patrick Jansson. Polytypic programming. In John Launchbury, Erik Meijer, and Tim Sheard, editors, *Advanced Functional Programming*, volume 1129 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[22] Mark P. Jones. Functional programming with overloading and higher-order polymorphism. In Johan Jeuring and Erik Meijer, editors, *LNCS 925: Advanced Functional Programming*. Springer-Verlag, 1995. Lecture notes from the First International Spring School on Advanced Functional Programming Techniques, Båstad, Sweden.

[23] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: A practical design pattern for generic programming. In *Types in Language Design and Implementation*, 2003.

[24] Ralf Lämmel and Simon Peyton Jones. Scrap more Boilerplate: Reflection, zips, and generalised casts. In *International Conference on Functional Programming*, pages 244–255. ACM Press, 2004.

[25] Ralf Lämmel and Simon Peyton Jones. Scrap your Boilerplate with class: Extensible generic functions. In *International Conference on Functional Programming*, pages 204–215. ACM Press, September 2005.

[26] Andres Löh. *Exploring Generic Haskell*. PhD thesis, Utrecht University, 2004.

[27] Saunders Mac Lane. *Categories for the Working Mathematician*. Springer-Verlag, 1971.

[28] Lambert Meertens. Paramorphisms. *Formal Aspects of Computing*, 4(5):413–424, 1992.

[29] Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In John Hughes, editor, *Functional Programming Languages and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 124–144. Springer-Verlag, 1991.

[30] John C. Mitchell and Albert R. Meyer. Second-order logical relations. In Rohit Parikh, editor, *Logics of Programs*, volume 193 of *Lecture Notes in Computer Science*, pages 225–236, 1985.

[31] Ross A. Paterson. *Reasoning about Functional Programs*. PhD thesis, University of Queensland, 1987.

[32] Simon Peyton Jones. *The Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

[33] John C. Reynolds. Towards a theory of type structure. In B. Robinet, editor, *Colloque sur la Programmation*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425. Springer-Verlag, 1974.

[34] Jeremy G. Siek, Lie-Quan Lee, and Andrew Lumsdaine. *The Boost Graph Library*. Addison-Wesley, 2002.

[35] Neil J. A. Sloane. On-line encyclopedia of integer sequences. `http://www.research.att.com/~njas/sequences/`. Accessed May 2007.

[36] Michael B. Smyth and Gordon D. Plotkin. The category-theoretic solution of recursive domain equations. *SIAM Journal on Computing*, 11(4):761–783, November 1982.

[37] Walid Taha. A gentle introduction to multi-stage programming. In Christian Lengauer, Don Batory, Charles Consel, and Martin Odersky, editors, *Domain-Specific Program Generation*, number 3016 in Lecture Notes in Computer Science, pages 30–50. Springer-Verlag, 2004.

[38] Tarmo Uustalu, Varmo Vene, and Alberto Pardo. Recursion schemes from comonads. *Nordic Journal of Computing*, 2001.

[39] Varmo Vene and Tarmo Uustalu. Functional programming with apomorphisms (corecursion). *Proceedings of the Estonian Academy of Sciences: Physics, Mathematics*, 47(3):147–161, 1998. 9th Nordic Workshop on Programming Theory.

[40] Philip Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture*, pages 347–359. ACM, 1989.