GENERIC PROGRAMMING WITHIN DEPENDENTLY TYPED PROGRAMMING

Thorsten Altenkirch

School of Computer Science and Information Technology, University of Nottingham
Wollaton Road, Nottingham, NG8 1BB, UK

txa@cs.nott.ac.uk

Conor McBride

Department of Computer Science, University of Durham
South Road, Durham, DH1 3LE, UK

C.T.McBride@durham.ac.uk

Abstract We show how higher kinded generic programming can be represented faithfully within a dependently typed programming system. This development has been implemented using the OLEG system.

The present work can be seen as evidence for our thesis that extensions of type systems can be done by *programming* within a dependently typed language, using data as codes for types.

1. Introduction

Generic programming [BJJM98, HP00, JJ97, JBM98] allows programmers to explain how a single algorithm can be instantiated for a variety of datatypes, by computation over each datatype's structure. This can be viewed as a rationalization and generalization of Haskell's `derive` mechanism [PH$^+$99]. For example, the representation of λ-terms with de Bruijn variables as a nested datatype [BP99, AR99]

```
data Lam a = Var a | App (Lam a) (Lam a) | Lam (Lam (Maybe a))
```

can be given an equality function by hand

```
instance (Eq a) ⇒ Eq (Lam a) where
  Var x   == Var y    = x == y
  App t u == App t' u' = t == t' && u == u'
  Lam t   == Lam u    = t == u
  _       == _        = False
```

but we can also instruct the compiler to derive it:

```
data Lam a = Var a | App (Lam a) (Lam a) | Lam (Lam (Maybe a))
             deriving Eq
```

In contrast to the above, we may implement the `fmap` function witnessing the fact that `Lam` is a functor

```
instance Functor Lam where
  fmap f (Var x)   = Var (f x)
  fmap f (App t u) = App (fmap f t) (fmap f u)
  fmap f (Lam t)   = Lam (fmap (fmap f) t)
```

but the compiler does *not* know how to derive it, i.e. if we attempt

```
data Lam a = Var a | App (Lam a) (Lam a) | Lam (Lam (Maybe a))
             deriving Functor
```

we get an error message

```
  ERROR "lam.hs" (line 2): Cannot derive instances of class "Functor"
```

1.1. The Generic Haskell Approach

Generic Haskell [CHJ+01] overcomes this limitation by allowing the programmer to define generic functions by recursion over the structure of datatypes. For example, a generic equality function can be defined in Generic Haskell: first we must give its type, which is indexed by a kind

```
type Eq {[ * ]} t = t → t → Bool
type Eq {[ k → l ]} t = forall u. Eq {[ k ]} u → Eq {[ l ]} (t u)
```

That is, == is a binary boolean operator at ground types, but at higher kinds, it is a parametric operation, transforming an equality at the source kind to an equality at the target kind. As Hinze observed [Hin00], this parametrization is systematic—we need only implement == itself by recursion over ground types[1]:

```
(==) {| t :: k |} :: Eq {[ k ]} t
(==) {| Unit |}         _              _            = True
(==) {| :+: |} eqA eqB (Inl a1)       (Inl a2)     = eqA a1 a2
(==) {| :+: |} eqA eqB (Inl a)        (Inr b)      = False
(==) {| :+: |} eqA eqB (Inr b)        (Inl a)      = False
(==) {| :+: |} eqA eqB (Inr b1)       (Inr b2)     = eqB b1 b2
(==) {| :*: |} eqA eqB (a1 :*: b1) (a2 :*: b2) = eqA a1 a2 && eqB b1 b2
(==) {| (->) |} eqA eqB _             _            =
  error "(==) not defined for function types"
(==) {| Con c |}   eqA (Con _ a1)  (Con _ a2)  = eqA a1 a2
...
```

The map function also fits this pattern: its 'kind-indexed type' takes two parameters—the source and target of the function being mapped.

```
type Map {[ * ]} t1 t2 = t1 -> t2
type Map {[ k -> l ]} t1 t2 = forall u1 u2.
  Map {[ k ]} u1 u2 -> Map {[ l ]} (t1 u1) (t2 u2)
```

Which instances of `Map {[ k ]} t1 t2` can actually be defined? We cannot map a `t1` to a `t2` for any two type constructors. However, we can map between different applications of the *same* type constructor, provided we can map between its arguments. The top-level 'type-indexed value' is defined *only along the diagonal*, and this goes for type-indexed values in general.

```
gmap {| t :: k |} :: Map {[ k ]} t t
gmap {| Unit |}  = id
gmap {| :+: |} gmapA gmapB (Inl a)   = Inl (gmapA a)
gmap {| :+: |} gmapA gmapB (Inr b)   = Inr (gmapB b)
gmap {| :*: |} gmapA gmapB (a :*: b) = (gmapA a) :*: (gmapB b)
gmap {| (->) |} gmapA gmapB _
  = error "gmap not defined for function types"
gmap {| Con c |}   gmapA   (Con d a)   = Con d (gmapA a)
...
```

Generic Haskell is an extension of Haskell, currently implemented as a *preprocessor*. In this paper we show that **dependently typed pro-**

[1] We omit the cases for labels and base types.

gramming can already express generic operations—we need only to implement a *library*. Indeed, the reader may want to compare the Generic Haskell code above with our own implementation of generic equality and map (section 2.3).

1.2. Introducing Dependent Types

We are using an implementation of Type Theory as a dependently typed programming language: McBride's OLEG[2] system [McB99], although essentially a proof checker, serves reluctantly as a rather spartan call-by-value programming language. An alternative would have been to use the prototype Cayenne compiler [Aug98], but Cayenne does not support inductively defined families as primitive, so this would have introduced an additional overhead. For the sake of readability, we take some notational liberties—overloading, infix and postfix operations, superscripting and subscripting—but the full OLEG script, together with a document explaining its correspondence with this paper, is available online [AM02].

In a dependently typed programming language, we may define families of types which depend on values. One such is $\mathsf{Fin} : \mathbb{N} \to \mathsf{Type}$ of finite types indexed by their size:

$$\text{data} \quad \frac{n : \mathbb{N}}{\mathsf{Fin}\, n : \mathsf{Type}} \quad \text{where} \quad \frac{}{0_n : \mathsf{Fin}\, \mathsf{s}n} \quad \frac{i : \mathsf{Fin}\, n}{\mathsf{s}_n\, i : \mathsf{Fin}\, \mathsf{s}n}$$

We say that Fin is an **inductive family** of datatypes **indexed** by $\mathbb{N}$ [Dyb91]. The s_n constructor embeds $\mathsf{Fin}\, n$ as the 'old' elements of $\mathsf{Fin}\,\mathsf{s}n$, whilst 0 makes a 'new' element. Observe that both constructors target a restricted section of the family—the types with at least one element. $\mathsf{Fin}\, 0$ is quite rightly uninhabited.

In our notation, we introduce all our global identifiers with their type signatures either by datatype declarations (data ... where) or recursive definitions (let). The natural deduction presentation, although it may seem unusual at first, does hide quite a lot of inferrable detail. The 'flat' types of the above identifiers, given in full, are

$$\begin{aligned} &\mathsf{Fin} : \mathbb{N} \to \mathsf{Type} \\ &0 : \forall_{n:\mathbb{N}}.\, \mathsf{Fin}\, \mathsf{s}n \\ &\mathsf{s} : \forall_{n:\mathbb{N}}.\, \mathsf{Fin}\, n \to \mathsf{Fin}\, \mathsf{s}n \end{aligned}$$

[2] OLEG is a rearrangement of Pollack's LEGO system [LP92], with primitive support for programming.

Functions may have return types which depend on their argument values: $\rightarrow$ is just syntactic sugar for the vacuous case of the quantifier $\forall$, which binds the argument for use in the return type. Arguments on which there is nontrivial dependency can often be inferred from usage by the typechecker, just as Hindley-Milner typecheckers infer instances of polymorphic functions. Subscripting the binding in a $\forall$-type tells the typechecker to infer the argument by default—we may also write it as a subscript in an application if we wish to draw attention to it. In the natural deduction style, we can omit the subscripted $\forall$s, because we show standard *usage* as well as enough information for types to be inferred.

We define recursive functions by dependent pattern matching, as introduced in [Coq92]. For example, $\mathbf{emb} : \forall_{n:\mathbb{N}}.\ \mathsf{Fin}\, n \rightarrow \mathsf{Fin}\, \mathsf{s}n$ witnesses the fact that there is a value preserving embedding from $\mathsf{Fin}\, n$ to $\mathsf{Fin}\, \mathsf{s}n$.

$$\text{let} \quad \frac{x \ :\ \mathsf{Fin}\, n}{\mathbf{emb}\, x \ :\ \mathsf{Fin}\, \mathsf{s}n} \qquad \begin{array}{l} \mathbf{emb}\ \ \mathsf{0} \ \ \mapsto \mathsf{0} \\ \mathbf{emb}\, (\mathsf{s}\, x) \mapsto \mathsf{s}\, (\mathbf{emb}\, x) \end{array}$$

We can also exploit Fin to give a functional representation of vectors—lists of fixed length. This is an alternative to the more common inductive representation which follows the same pattern as the definition of lists.

$$\text{let} \quad \frac{A \ :\ \mathsf{Type} \qquad n \ :\ \mathbb{N}}{A^n \ :\ \mathsf{Type}} \qquad A^n \ \mapsto\ (\mathsf{Fin}\, n) \rightarrow A$$

Given $\boldsymbol{a} : A^{\mathsf{s}n}$, its head $\mathbf{hd}\ \boldsymbol{a} : A$ is given by $\mathbf{hd}\ \boldsymbol{a} \ \mapsto\ \boldsymbol{a}\ \mathsf{0}$ and its tail $\mathbf{tl}\ \boldsymbol{a} : A^n$ by $\mathbf{tl}\ \boldsymbol{a} \ \mapsto\ \boldsymbol{a} \circ \mathsf{s}$. As we never construct vectors containing types here, we may safely overload the type constructor as the operation to construct a constant vector:

$$\text{let} \quad \frac{a \ :\ A}{a^n \ :\ A^n} \qquad a^n\ x \ \mapsto\ a$$

We can lift application to vectors, as McBride does in his Haskell definition of n-ary `zipWith` [McB02]:

$$\text{let} \quad \frac{\boldsymbol{f} \ :\ (A \rightarrow B)^n \qquad \boldsymbol{a} \ :\ A^n}{\boldsymbol{f}\ \boldsymbol{a} \ :\ B^n} \qquad \boldsymbol{f}\ \boldsymbol{a}\ x \ \mapsto\ (\boldsymbol{f}\ x)\ (\boldsymbol{a}\ x)$$

These two definitions allow us to **map** a function f across a vector $\boldsymbol{a}$ just by writing $f^n\ \boldsymbol{a}$. In fact, this reduces to the composition $f \circ \boldsymbol{a}$.

1.3. Dependent Types are Generic Types

Dependent type systems provide a natural setting for generic programming because they provide a means for **reflection**. An example of this is the construction of a simply typed universe[3] by defining a a representation $\mathsf{U}\ :\ \mathsf{Type}$ for simple types over a base type nat:

$$\text{data}\quad \frac{}{\mathsf{U}\ :\ \mathsf{Type}}$$

$$\text{where}\quad \frac{}{\mathsf{nat}\ :\ \mathsf{U}} \qquad \frac{A, B\ :\ \mathsf{U}}{A \Rightarrow B\ :\ \mathsf{U}}$$

and a decoding function $\mathbf{El}\ :\ \mathsf{U} \rightarrow \mathsf{Type}$, which assigns a meaning to each code.

$$\text{let}\quad \frac{A\ :\ \mathsf{U}}{\mathbf{El}\,A\ :\ \mathsf{Type}}$$

$$\begin{array}{lcl} \mathbf{El}\ \ \mathsf{nat} & \mapsto & \mathbb{N} \\ \mathbf{El}\,(A \Rightarrow B) & \mapsto & (\mathbf{El}\,A) \rightarrow (\mathbf{El}\,B) \end{array}$$

A program which is *generic* over such a universe can be given a type which abstracts over all the *codes* in U and refers to the elements of types decoded by $\mathbf{El}$. These types exploit the dependent function space

$$\forall A\!:\!\mathsf{U}.\ \ldots \mathbf{El}\,A \ldots$$

For example, every type in the above universe contains a 'zero value', which can be defined as follows:

$$\text{let}\quad \mathbf{zero}\ :\ \forall A\!:\!\mathsf{U}.\,\mathbf{El}\,A \qquad \begin{array}{lcl} \mathbf{zero}\ \ \mathsf{nat} & \mapsto & 0 \\ \mathbf{zero}\,(A \Rightarrow B) & \mapsto & \lambda a\!:\!\mathbf{El}\,A.\,\mathbf{zero}\,B \end{array}$$

The universe U above, containing all higher types over $\mathbb{N}$, is rather large. Hence, it has very few useful generic operations—operations which are meaningful for every type with a code in U. In general, the more specialized the set of codes, the larger the library of useful generic operations. If every operation made sense at every type, we would not need types in the first place.

Indeed, we may consider the family Fin, introduced above, as a smaller universe only containing finite types. Based on this view we may give an alternative representation of the type of λ-terms which uses finite types for the set of variables:

[3]This is a simplified version of Martin-Löf's definition of a universe with codes for $\mathbb{N}$ and dependent function spaces ($\forall$-types). [ML84, NPS90]

$$\text{data}\quad \frac{n \,:\, \mathbb{N}}{\mathsf{Lam}\, n \,:\, \mathsf{Type}}$$

$$\text{where}\quad \frac{x \,:\, \mathsf{Fin}\, n}{\mathsf{var}\, x \,:\, \mathsf{Lam}\, n} \qquad \frac{f, a \,:\, \mathsf{Lam}\, n}{\mathsf{app}\, f\ a \,:\, \mathsf{Lam}\, n} \qquad \frac{t \,:\, \mathsf{Lam}\, \mathsf{s}n}{\mathsf{lam}\, t \,:\, \mathsf{Lam}\, n}$$

$\mathsf{Lam}\ n$ is the type of λ-terms with at most n free variables, embedded from $\mathsf{Fin}\ n$ by the var constructor.

The advantage of using $\mathbb{N}$ to index Lam is that we have *two* natural modes of computation for working with λ-terms: we may exploit the structure not only of the terms themselves, but also of the datatype used to index them. For example, we may write the operation which closes a term by abstracting over the available free variables.

$$\text{let}\quad \frac{t \,:\, \mathsf{Lam}\, n}{\mathbf{close}\, n\ t \,:\, \mathsf{Lam}\, 0} \qquad \begin{array}{l} \mathbf{close}\ 0\ \ t \mapsto t \\ \mathbf{close}\, \mathsf{s}n\ t \mapsto \mathbf{close}\ n\ (\mathsf{lam}\ t) \end{array}$$

This seems not possible in Haskell, not even in Generic Haskell because the universe over which generic programs are defined is simply too large.

Another advantage of the dependently typed version is that it can be easily generalized to a precise account of simply typed λ-terms over a given *signature*, see [AR99].

1.4. Related Work

Much of this work has been influenced by the Generic Haskell project [CHJ$^+$01]. In the current version Generic Haskell is implemented as a compiler front end—typechecking at this level is not yet realized.

The topic of this paper is clearly related to [PR98], where the authors also use Type Theory to represent polytypic programs. However, they do not actually introduce a universe but simply a library of operators which work on functors and bifunctors of first order kind. This gives a convenient way to construct **map** but is hardly extensible: the library would have to be rewritten each time a new polytypic operation is added. Exploiting a universe construction, we avoid this problem and present a combinator to derive many polytypic programs. Our domain is also more general in that we allow higher order kinds and mutual inductive definitions with several parameters.

Recently, Benke [Ben02] has presented preliminary results on an implementation of generic programming in AGDA, a dependently typed system developed at Chalmers University, Göteborg. His goal is to codify generic operations on dependent datatypes.

1.5. Overview

In the present paper we show how generic programming as implemented in Generic Haskell can be coded within a dependently typed programming language: we define a universe faithfully representing concrete Haskell datatypes (sections 2.1,2.2), i.e. any Haskell datatype not involving function spaces. This is just one possible choice: we could have used positive, strictly positive or just finite types as introduced before. We include datatypes of higher kinds and nested datatypes, generalizing the construction for regular types given in [McB01]. We present a generic recursion operator, **fold**, (section 2.3) which gives one way to define generic functions such as $==$ or **map** for arbitrary kinds. In section 3 we present more details of our encoding and discuss our implementation of **fold**. Of course, generic operations which are not instances of **fold** may still be defined directly. The library approach leaves the constructed universe accessible to the programmer.

2. Generic programming in OLEG

In this section, we show how to use our implementation of generic programming in Oleg. Our intention is to be as faithful to Generic Haskell as we can. As a running example we employ the following Haskell datatype declaration:

```
data Bush a = Nil | Cons a (Bush (Bush a))
data WBush = W (Bush WBush)
```

`Bush` is a nested datatype which exploits higher types; `WBush` uses `Bush` (mutual dependency is also permitted). We chose this example, although a little artificial, because it illustrates all the features present in our encoding of datatypes. However, it is not completely pointless: `Bush` is a representation of partial functions over binary trees and `WBush` represent trees branching over a finite set of binary trees.

We will show how to represent this datatype internally, and how to implement the generic functions $==$ and **map** in our system.

2.1. Encoding datatype declarations

First we introduce $\Box$ as a representation of Haskell kinds:

$$\text{data} \quad \frac{}{\Box \;:\; \mathsf{Type}} \quad \text{where} \quad \frac{}{\star \;:\; \Box} \quad \frac{J, K \;:\; \Box}{J \Rightarrow K \;:\; \Box}$$

`WBush` is a ground type hence its kind is $\star \;:\; \Box$. `Bush` maps types to types, hence its kind is $\star \Rightarrow \star$.

Signatures $\mathsf{Sig} \;:\; \mathsf{Type}$ are sequences of kinds. As we often need access to the argument list of a kind, we shall identify $\mathsf{Sig} \;\mapsto\; \Box$ and use an alternative notation for the constructors:

$$\begin{array}{l} \varepsilon \;\mapsto\; \star \\ K;\Sigma \;\mapsto\; K \Rightarrow \Sigma \end{array}$$

When using a kind $K \;:\; \Box$ as a signature we write $\langle K \rangle \;:\; \mathsf{Sig}$. This is just the identity map $\langle K \rangle \;\mapsto\; K$ in our implementation.

As an example we define the signature Δ^{W} corresponding to the declaration of Bush and WBush:

$$\text{let} \quad \frac{}{\Delta^{\mathsf{W}} \;:\; \mathsf{Sig}} \quad \Delta^{\mathsf{W}} \;\mapsto\; \star \Rightarrow \star; \star; \varepsilon$$

We will use typed de Bruijn variables for our datatype declarations. Given a signature Σ and a kind K we define the type of variables of that kind as

$$\text{data} \quad \frac{\Sigma \;:\; \mathsf{Sig} \quad K \;:\; \Box}{\mathsf{Var}\,\Sigma\,K \;:\; \mathsf{Type}} \quad \text{where} \quad \frac{}{\mathsf{0} \;:\; \mathsf{Var}\,(K;\Sigma)\,K} \quad \frac{v \;:\; \mathsf{Var}\,\Sigma\,K}{\mathsf{s}v \;:\; \mathsf{Var}\,(J;\Sigma)\,K}$$

We will now introduce the type Ty of polynomial type expressions representing the right hand side of a datatype declaration. Ty is indexed by two signatures Δ for mutually defined datatypes and Λ for parameter kinds. To access these signatures we use two different variable constructors: D for recursive variables (like `Bush`) and V for parameters (like `a` in the definition of `Bush`).

$$\text{data}\quad \frac{\Delta,\Lambda \;:\; \mathsf{Sig} \qquad K \;:\; \Box}{\mathsf{Ty}\,\Delta\,\Lambda\,K \;:\; \mathsf{Type}}$$

$$\text{where}\quad \frac{v \;:\; \mathsf{Var}\,\Delta\,K}{\mathsf{D}\,v \;:\; \mathsf{Ty}\,\Delta\,\Lambda\,K} \qquad \frac{v \;:\; \mathsf{Var}\,\Lambda\,K}{\mathsf{V}\,v \;:\; \mathsf{Ty}\,\Delta\,\Lambda\,K}$$

$$\frac{F \;:\; \mathsf{Ty}\,\Delta\,\Lambda\,(J \Rightarrow K) \qquad X \;:\; \mathsf{Ty}\,\Delta\,\Lambda\,J}{F \cdot X \;:\; \mathsf{Ty}\,\Delta\,\Lambda\,K}$$

$$\frac{}{\mathsf{0} \;:\; \mathsf{Ty}\,\Delta\,\Lambda\,\star} \qquad \frac{S, T \;:\; \mathsf{Ty}\,\Delta\,\Lambda\,\star}{S + T \;:\; \mathsf{Ty}\,\Delta\,\Lambda\,\star}$$

$$\frac{}{\mathsf{1} \;:\; \mathsf{Ty}\,\Delta\,\Lambda\,\star} \qquad \frac{S, T \;:\; \mathsf{Ty}\,\Delta\,\Lambda\,\star}{S \times T \;:\; \mathsf{Ty}\,\Delta\,\Lambda\,\star}$$

The right hand side of a datatype declaration in kind K is an element of $\mathsf{Ty}\,\Delta\,\langle K\rangle\,\star$. For example, we represent the right hand sides of Bush and WBush as follows:

$$\text{let}\quad \frac{}{\mathsf{Bush} \;:\; \mathsf{Ty}\,\Delta^{\mathsf{W}}\,\langle \star \Rightarrow \star\rangle\,\star}$$
$$\mathsf{Bush} \;\mapsto\; \mathsf{1} + (\mathsf{V}\,\mathsf{0}) \times (\mathsf{D}\,\mathsf{0}) \cdot (\mathsf{D}\,\mathsf{0}) \cdot (\mathsf{V}\,\mathsf{0})$$

$$\text{let}\quad \frac{}{\mathsf{WBush} \;:\; \mathsf{Ty}\,\Delta^{\mathsf{W}}\,\langle \star\rangle\,\star} \qquad \mathsf{WBush} \;\mapsto\; (\mathsf{D}\,\mathsf{0}) \cdot (\mathsf{D}\,\mathsf{s0})$$

We define the meaning of a signature $[\![\Delta]\!]$ as the type of functions which assign to each variable a datatype declaration of the appropriate type:

$$\text{let}\quad \frac{\Delta \;:\; \mathsf{Sig}}{[\![\Delta]\!] \;:\; \mathsf{Type}} \qquad [\![\Delta]\!] \;\mapsto\; \forall_{K\,:\,\Box}.(\mathsf{Var}\,\Delta\,K) \to \mathsf{Ty}\,\Delta\,\langle K\rangle\,\star$$

We can now give the full representation of the Haskell declaration from the beginning of this section:

$$\text{let}\quad \frac{}{\delta^{\mathsf{W}} \;:\; [\![\Delta^{\mathsf{W}}]\!]} \qquad \begin{array}{lcl} \delta^{\mathsf{W}}\,\mathsf{0} & \mapsto & \mathsf{Bush} \\ \delta^{\mathsf{W}}\,\mathsf{s0} & \mapsto & \mathsf{WBush} \end{array}$$

2.2. Constructing data

We shall now populate our datatypes with data. For the subsequent discussion, assume as given an arbitrary declaration $\delta \;:\; [\![\Delta]\!]$. We abbreviate $\mathsf{Ty}_\Delta\,K \;\mapsto\; \mathsf{Ty}\,\Delta\,\epsilon\,K$.

To define the interpretation of types, we introduce an iterated application operator and substitution on type expressions. Both are parametrized by an argument stack of closed types, represented via a function space, as follows:

$$\text{let}\quad \frac{\Delta\ :\ \mathsf{Sig}\qquad K\ :\ \Box}{\mathbf{Args}\,\Delta\,K\ :\ \mathsf{Type}}$$
$$\mathbf{Args}\,\Delta\,K\ \mapsto\ \forall_{J:\Box}.\,\mathsf{Var}\,\langle K\rangle\,J\ \to\ \mathsf{Ty}_\Delta\,J$$

Here, we only give the signatures of the application operator

$$\text{let}\quad \frac{X\ :\ \mathsf{Ty}_\Delta\,K\qquad \vec{Y}\ :\ \mathbf{Args}\,\Delta\,K}{X\,@\,\vec{Y}\ :\ \mathsf{Ty}_\Delta\,\star}$$

and the substitution operator.

$$\text{let}\quad \frac{X\ :\ \mathsf{Ty}\,\Delta\,\langle J\rangle\,K\qquad \vec{Y}\ :\ \mathbf{Args}\,\Delta\,J}{X[\,\vec{Y}\,]\ :\ \mathsf{Ty}_\Delta\,K}$$

Their implementations can be found in section 3.

We are now able to define the interpretation of types $[\![T]\!]_\delta$.

$$\text{data}\quad \frac{T\ :\ \mathsf{Ty}_\Delta\,\star}{[\![T]\!]_\delta\ :\ \mathsf{Type}}$$

$$\text{where}\quad \frac{t\ :\ [\![(\delta\ v)[\vec{X}]]\!]_\delta}{\mathsf{con}_v\ t\ :\ [\![\mathsf{D}\ v @ \vec{X}]\!]_\delta}$$

$$\frac{s\ :\ [\![S]\!]_\delta}{\mathsf{inl}\ s\ :\ [\![S+T]\!]_\delta}\qquad \frac{t\ :\ [\![T]\!]_\delta}{\mathsf{inr}\ t\ :\ [\![S+T]\!]_\delta}$$

$$\frac{}{\mathsf{void}\ :\ [\![1]\!]_\delta}\qquad \frac{s\ :\ [\![S]\!]_\delta\qquad t\ :\ [\![T]\!]_\delta}{\mathsf{pair}\ s\ t\ :\ [\![S\times T]\!]_\delta}$$

As an example we can derive the constructors for Bush and WBush.

$$\text{let}\quad \frac{A\ :\ \mathsf{Ty}_{\Delta\mathsf{w}}\,\star}{\mathsf{Nil}\ A\ :\ [\![\mathsf{D}\ 0\cdot A]\!]_{\delta\mathsf{w}}}$$

$$\frac{x\ :\ [\![A]\!]_{\delta\mathsf{w}}\qquad b\ :\ [\![\mathsf{D}\ 0\cdot(\mathsf{D}\ 0\cdot A)]\!]_{\delta\mathsf{w}}}{\mathsf{Cons}\ A\ x\ b\ :\ [\![\mathsf{D}\ 0\cdot A]\!]_{\delta\mathsf{w}}}$$

$$\frac{x\ :\ [\![\mathsf{D}\ 0\cdot\mathsf{D}\ \mathsf{s}0]\!]_{\delta\mathsf{w}}}{\mathsf{W}\ x\ :\ [\![\mathsf{D}\ \mathsf{s}0]\!]_{\delta\mathsf{w}}}$$

$$\begin{aligned}&\mathsf{Nil}\ A\ \mapsto\ \mathsf{con}\ (\mathsf{inl}\ \mathsf{void})\\&\mathsf{Cons}\ A\ x\ b\ \mapsto\ \mathsf{con}\ (\mathsf{inr}\ (\mathsf{pair}\ x\ b))\\&\mathsf{W}\ x\ \mapsto\ \mathsf{con}\ x\end{aligned}$$

Of course, it is not enough to construct elements of the datatypes in our universe. We must be able to compute with them too. Here, the power of dependent pattern matching, as proposed by Thierry Coquand [Coq92], delivers exactly what we need—although we have defined $[\![\cdot]\!]_\delta$ for arbitrary codes, we may define functions over particular instances of it by pattern matching, supplying cases for only those constructors which apply.

Using the previous definitions we can already implement generic functions such as **read**, which constructs a typed representation out of untyped data (or returns an error value). However, in the next section we will cover the more interesting case of generic recursion.

2.3. Using generic recursion

To define generic functions like $==$ and **map** we introduce a generic recursion operator **fold**. The design of **fold** is based on ideas due to Hinze [Hin00] and implements the kind of polytypic recursion present in Generic Haskell [CHJ+01].

To motivate **fold**'s type, let us first look at the types of $==$ and **map**.

We may introduce $==$'s type **Eq** by recursion on kinds:

$$\text{let}\quad \frac{X \;:\; \mathsf{Ty}_\Delta\, K}{\mathbf{Eq}_K\, X \;:\; \mathsf{Type}}$$
$$\begin{array}{ll} \mathbf{Eq}_\star & S \mapsto [\![S]\!] \to [\![S]\!] \to \mathsf{Bool} \\ \mathbf{Eq}_{J\Rightarrow K} & F \mapsto \forall_{X:\mathsf{Ty}_\Delta\, J}.\, \mathbf{Eq}_J\, X \to \mathbf{Eq}_K\, (F \cdot X) \end{array}$$

Now, $==$ gets the following type:

$$\frac{X \;:\; \mathsf{Ty}_\Delta\, K}{(==)_K \;:\; \mathbf{Eq}_K\, X}$$

Similarly, we define **map**'s type **Map**:

$$\text{let}\quad \frac{X, Y \;:\; \mathsf{Ty}_\Delta\, K}{\mathbf{Map}_K\, X\; Y \;:\; \mathsf{Type}}$$
$$\begin{array}{ll} \mathbf{Map}_\star & S\; T \mapsto [\![S]\!] \to [\![T]\!] \\ \mathbf{Map}_{J\Rightarrow K} & F\; G \mapsto \forall_{X,Y:\mathsf{Ty}_\Delta\, J}. \\ & \qquad\qquad \mathbf{Map}_J\, X\; Y \to \mathbf{Map}_K\, (F \cdot X)\, (G \cdot Y) \end{array}$$

As in Generic Haskell, **map**'s type is the diagonalisation of **Map**:

$$\frac{X \;:\; \mathsf{Ty}_\Delta\, K}{\mathbf{map}_K \;:\; \mathbf{Map}_K\, X\; X}$$

Eq has one type argument, whereas **Map** takes two. This can easily be generalized to $n+1$-ary operators giving rise to the type of **fold**.

The type of **fold** is *computed* from the kind at which it acts by the function **Fold**, capturing the general case with $n+1$ type arguments via vectors, as introduced earlier. **Fold** is also parametrized by a type family Φ which characterizes the operator's behaviour at ground type.

$$\frac{S \;:\; \mathsf{Ty}_\Delta \star \qquad \boldsymbol{T} \;:\; (\mathsf{Ty}_\Delta \star)^n}{\Phi\; S\; \boldsymbol{T} \;:\; \mathsf{Type}}$$

Now we are ready to define **Fold** by recursion on kinds:

$$\text{let}\quad \frac{X \;:\; \mathsf{Ty}_\Delta\, K \qquad \boldsymbol{Y} \;:\; (\mathsf{Ty}_\Delta\, K)^n}{\mathbf{Fold}_K\; \Phi\; X\; \boldsymbol{Y} \;:\; \mathsf{Type}}$$

$$\begin{array}{lll}
\mathbf{Fold}_\star & \Phi\; S\; \boldsymbol{T} \mapsto & [\![S]\!]_\delta \to \Phi\; S\; \boldsymbol{T} \\
\mathbf{Fold}_{J \Rightarrow K} & \Phi\; F\; \boldsymbol{G} \mapsto & \forall_{X:\mathsf{Ty}_\Delta\, J}.\, \forall_{\boldsymbol{Y}:(\mathsf{Ty}_\Delta\, J)^n}. \\
& & \quad \mathbf{Fold}_J\; \Phi\; X\; \boldsymbol{Y} \;\to\; \mathbf{Fold}_K\; \Phi\; (F \cdot X)\; (\boldsymbol{G} \cdot^n \boldsymbol{Y})
\end{array}$$

We hope that it is easy to see that **Eq** and **Map** can be derived[4] from **Fold** by setting

$$\begin{array}{l}
\Phi^{==}\; S\; \boldsymbol{T} \;\mapsto\; [\![S]\!]_\delta \to \mathsf{Bool} \\
\Phi^{\mathbf{map}}\; S\; \boldsymbol{T} \;\mapsto\; [\![\mathbf{hd}\; \boldsymbol{T}]\!]_\delta
\end{array}$$

The parameters to **fold** explain how to construct Φ's at multiple instances of each type constructor, given Φ's for the relevant arguments.

$$\frac{\phi \;:\; \Phi\; ((\delta\; v)[\vec{W}])\; ((\delta\; v)^n[\vec{\boldsymbol{Z}}]^n)}{\mathit{doCon}\; v\; \phi \;:\; \Phi\; (\mathsf{D}\; v\; @\, \vec{W})\; ((\mathsf{D}\; v)^n\; @^n\, \vec{\boldsymbol{Z}})}$$

$$\frac{\phi_1 \;:\; \Phi\; S_1\; \boldsymbol{T}_1}{\mathit{doInl}\; \phi_1 \;:\; \Phi\; (S_1 + S_2)\; (\boldsymbol{T}_1 +^n \boldsymbol{T}_2)} \qquad \frac{\phi_2 \;:\; \Phi\; S_2\; \boldsymbol{T}_2}{\mathit{doInr}\; \phi_2 \;:\; \Phi\; (S_1 + S_2)\; (\boldsymbol{T}_1 +^n \boldsymbol{T}_2)}$$

$$\frac{}{\mathit{doVoid} \;:\; \Phi\; 1\; 1^n} \qquad \frac{\phi_1 \;:\; \Phi\; S_1\; \boldsymbol{T}_1 \qquad \phi_2 \;:\; \Phi\; S_2\; \boldsymbol{T}_2}{\mathit{doPair}\; \phi_1\; \phi_2 \;:\; \Phi\; (S_1 \times S_2)\; (\boldsymbol{T}_1 \times^n \boldsymbol{T}_2)}$$

Given the above, we define **fold** along the diagonal:

$$\frac{X \;:\; \mathsf{Ty}_\Delta\, K}{\mathbf{fold}_X\; \mathit{doCon}\; \mathit{doInl}\; \mathit{doInr}\; \mathit{doVoid}\; \mathit{doPair} \;:\; \mathbf{Fold}_K\; \Phi\; X\; X^n}$$

Details of the implementation can be found in the next section, or online in form of the OLEG sources [AM02].

[4]Up to trivial isomorphisms.

As an example for using **fold** let us derive $==$:

$$\begin{aligned}
&\mathit{doCon}^{==}\ v\ \phi\ (\mathsf{con}\ x) \ \mapsto\ \phi\ x\\
&\mathit{doInl}^{==}\ \phi\ (\mathsf{inl}\ x) \ \mapsto\ \phi\ x\\
&\mathit{doInl}^{==}\ \phi\ (\mathsf{inr}\ y) \ \mapsto\ \mathsf{false}\\
&\mathit{doInr}^{==}\ \phi\ (\mathsf{inl}\ x) \ \mapsto\ \mathsf{false}\\
&\mathit{doInr}^{==}\ \phi\ (\mathsf{inr}\ y) \ \mapsto\ \phi\ y\\
&\mathit{doVoid}^{==}\ \mathsf{void} \ \mapsto\ \mathsf{true}\\
&\mathit{doPair}^{==}\ \phi_1\ \phi_2\ (\mathsf{pair}\ x\ y) \ \mapsto\ (\phi_1\ x) \wedge (\phi_2\ y)
\end{aligned}$$

Note that the rules cover every case which types permit. We now set

$$\text{let}\quad \frac{X\ :\ \mathsf{Ty}_{\Delta}\ K}{(==)_X\ :\ \mathbf{Fold}_K\ \Phi^{==}\ X\ X^0}$$
$$(==)_X\ \mapsto\ \mathbf{fold}_X\ \mathit{doCon}^{==}\ \mathit{doInl}^{==}\ \mathit{doInr}^{==}\ \mathit{doVoid}^{==}\ \mathit{doPair}^{==}$$

We may also define **map**:

$$\begin{aligned}
&\mathit{doCon}^{\mathbf{map}}\ v\ \phi\ \mapsto\ \mathsf{con}\ \phi\\
&\mathit{doInl}^{\mathbf{map}}\ \phi\ \mapsto\ \mathsf{inl}\ \phi\\
&\mathit{doInr}^{\mathbf{map}}\ \phi\ \mapsto\ \mathsf{inr}\ \phi\\
&\mathit{doVoid}^{\mathbf{map}}\ \mapsto\ \mathsf{void}\\
&\mathit{doPair}^{\mathbf{map}}\ \phi_1\ \phi_2\ \mapsto\ \mathsf{pair}\ \phi_1\ \phi_2
\end{aligned}$$

$$\text{let}\quad \frac{X\ :\ \mathsf{Ty}_{\Delta}\ K}{\mathbf{map}_X\ :\ \mathbf{Fold}_K\ \Phi^{\mathbf{map}}\ X\ X^1}$$
$$\begin{aligned}\mathbf{map}_X\ \mapsto\ \mathbf{fold}_X\ &\mathit{doCon}^{\mathbf{map}}\ \mathit{doInl}^{\mathbf{map}}\ \mathit{doInr}^{\mathbf{map}}\\ &\mathit{doVoid}^{\mathbf{map}}\ \mathit{doPair}^{\mathbf{map}}\end{aligned}$$

3. The Implementation

In this section, we explain in more detail how we implement our universe of concrete Haskell datatypes and the **fold** combinator by which we construct generic operations over them.

3.1. The universe construction

The implementation of the universe construction is exactly as specified in the previous section. The only details missing are the definitions of application and substitution.

Recall that we represent an argument stack by a function space. As with vectors, such functions (nonempty signatures) admit head and tail operations, by application to $\mathsf{0}$ and composition with s, respectively.

$$\text{let} \quad \frac{\vec{f} \;:\; \forall_{J:\Box}.\,\mathsf{Var}\,(K;\Sigma)\;J \;\to\; F\;J}{\mathbf{hd}\,\vec{f} \;:\; F\;K} \qquad \mathbf{hd}\,\vec{f} \;\mapsto\; \vec{f}\;\mathsf{0}$$

$$\text{let} \quad \frac{\vec{f} \;:\; \forall_{J:\Box}.\,\mathsf{Var}\,(K;\Sigma)\;J \;\to\; F\;J}{\mathbf{tl}\,\vec{f} \;:\; \forall_{J:\Box}.\,\mathsf{Var}\,\Sigma\;J \;\to\; F\;J} \qquad \mathbf{tl}\,\vec{f} \;\mapsto\; \vec{f} \circ \mathsf{s}$$

Hence we may now define the application operator.

$$\text{let} \quad \frac{X \;:\; \mathsf{Ty}_\Delta\,K \qquad \vec{Y} \;:\; \mathbf{Args}\,\Delta\,K}{X @_K \vec{Y} \;:\; \mathsf{Ty}_\Delta\,\star}$$

$$\begin{array}{lll} T \;@_\star\; \vec{Y} & \mapsto & T \\ F \;@_{J \Rightarrow K}\; \vec{Y} & \mapsto & (F \cdot \mathbf{hd}\;\vec{Y}) \;@_K\; (\mathbf{tl}\;\vec{Y}) \end{array}$$

As usual, we omit the kind subscript when we make use of @, as this kind can be inferred from the first argument.

We define substitution by recursion over the structure of type expressions. Note that a substitution is also a stack of arguments.

$$\text{let} \quad \frac{X \;:\; \mathsf{Ty}\,\Delta\,\langle J \rangle\,K \qquad \vec{Y} \;:\; \mathbf{Args}\,\Delta\,J}{X[\vec{Y}] \;:\; \mathsf{Ty}_\Delta\,K}$$

$$\begin{array}{ll} (\mathsf{V}\;v)[\vec{Y}] & \mapsto \vec{Y}\;v \\ (\mathsf{D}\;v)[\vec{Y}] & \mapsto \mathsf{D}\;v \\ (F \cdot X)[\vec{Y}] & \mapsto F[\vec{Y}] \cdot X[\vec{Y}] \\ \mathsf{0}[\vec{Y}] & \mapsto \mathsf{0} \\ \mathsf{1}[\vec{Y}] & \mapsto \mathsf{1} \\ (S + T)[\vec{Y}] & \mapsto S[\vec{Y}] + T[\vec{Y}] \\ (S \times T)[\vec{Y}] & \mapsto S[\vec{Y}] \times T[\vec{Y}] \end{array}$$

With these in place, our universe construction is ready for use.

3.2. A generic fold operator

Our **fold** operator explains how to make an iterative operation act on every datatype in our universe and lift parametrically to higher kinds, given its definition at each data constructor. In practice, it is much easier to define fold in an uncurried style, and then curry it for export to the user. We shall first need a counterpart to **Fold**, computing the type of an uncurried fold. $\mathbf{UFold}_K\ \Phi\ X\ \boldsymbol{Y}$, is defined in terms of *tuples* of folds for K's argument kinds. These tuples are built with OLEG's unit and pair types, $\mathbf{1}$ and $S \times T$, with constructors $()$ and (s, t), respectively.

$$\text{let}\quad \frac{\begin{array}{l}\Phi\ :\ \mathsf{Ty}_\Delta\,\star\ \to\ (\mathsf{Ty}_\Delta\,\star)^n\ \to\ \mathsf{Type}\\ X\ :\ \mathsf{Ty}_\Delta\,K\qquad \boldsymbol{Y}\ :\ (\mathsf{Ty}_\Delta\,K)^n\end{array}}{\mathbf{UFold}_K\ \Phi\ X\ \boldsymbol{Y}\ :\ \mathsf{Type}}$$

$$\frac{\begin{array}{l}\Phi\ :\ \mathsf{Ty}_\Delta\,\star\ \to\ (\mathsf{Ty}_\Delta\,\star)^n\ \to\ \mathsf{Type}\\ \vec{W}\ :\ \mathbf{Args}\ \Delta\ K\qquad \vec{\boldsymbol{Z}}\ :\ (\mathbf{Args}\ \Delta\ K)^n\end{array}}{\mathbf{UFolds}_K\ \Phi\ \vec{W}\ \vec{\boldsymbol{Z}}\ :\ \mathsf{Type}}$$

$$\begin{array}{l}\mathbf{UFold}_K\ \Phi\ X\ \boldsymbol{Y}\ \mapsto\\ \quad \forall_{\vec{W}:\mathbf{Args}\ \Delta\ K}.\ \forall_{\vec{\boldsymbol{Z}}:(\mathbf{Args}\ \Delta\ K)^n}.\ \mathbf{UFolds}_K\ \Phi\ \vec{W}\ \vec{\boldsymbol{Z}}\ \to\\ \qquad [\![X @ \vec{W}]\!]_\delta\ \to\ \Phi\ (X @ \vec{W})\ (\boldsymbol{Y} @^n \vec{\boldsymbol{Z}})\end{array}$$

$$\begin{array}{lcl}\mathbf{UFolds}_\star\ \Phi\ \vec{W}\ \vec{\boldsymbol{Z}} & \mapsto & \mathbf{1}\\ \mathbf{UFolds}_{J\Rightarrow K}\ \Phi\ \vec{W}\ \vec{\boldsymbol{Z}} & \mapsto & \mathbf{UFold}_J\ \Phi\ (\mathbf{hd}\ \vec{W})\ (\mathbf{hd}^n\ \vec{\boldsymbol{Z}})\ \times\\ & & \ \mathbf{UFolds}_K\ \Phi\ (\mathbf{tl}\ \vec{W})\ (\mathbf{tl}^n\ \vec{\boldsymbol{Z}})\end{array}$$

Observe that $(\mathbf{Args}\ \Delta\ K)^n$ expands to a 'matrix' type:

$$(\mathbf{Args}\ \Delta\ K)^n\ \mapsto\ \mathsf{Fin}\ n\ \to\ \forall_{J:\Box}.\ \mathsf{Var}\ \langle K\rangle\ J\ \to\ \mathsf{Ty}_\Delta\ J$$

It is useful to introduce the corresponding 'transpose' operator:

$$\text{let}\quad \frac{\vec{\boldsymbol{Z}}\ :\ (\mathbf{Args}\ \Delta\ K)^n}{\vec{\boldsymbol{Z}}^\top\ :\ \forall_{J:\Box}.\ \mathsf{Var}\ \langle K\rangle\ J\ \to\ (\mathsf{Ty}_\Delta\ J)^n}\qquad \vec{\boldsymbol{Z}}^\top\ v\ i\ \mapsto\ \vec{\boldsymbol{Z}}\ i\ v$$

We may now explain how to project the fold for a particular argument from a tuple of folds for an argument sequence:

$$\text{let}\quad \frac{fs\ :\ \mathbf{UFolds}_K\ \Phi\ \vec{W}\ \vec{\boldsymbol{Z}}\qquad v\ :\ \mathsf{Var}\ \langle K\rangle\ J}{fs.v\ :\ \mathbf{UFold}_J\ \Phi\ (\vec{W}\ v)\ (\vec{\boldsymbol{Z}}^\top\ v)}$$

$$\begin{array}{l}(f, fs).\mathsf{0}\ \mapsto\ f\\ (f, fs).(\mathsf{s}\ v)\ \mapsto\ fs.v\end{array}$$

Given arguments $doCon\ \ldots\ doPair$ as specified in the previous section, we shall define

$$\text{let}\quad \frac{\{do\text{'s}\}\qquad X\ :\ \mathsf{Ty}_\Delta\ K}{\mathbf{ufold}\ \{do\text{'s}\}\ X\ :\ \mathbf{UFold}_K\ \Phi\ X\ X^n}$$

For a fold at a higher kind expression—necessarily the application of some $(\mathsf{D}\ v)$—the tuple of folds passed as arguments explains what to do for each of that datatype's parameters. Consequently, we must define **ufold** in terms of a more general operator, **ufoldBody** which takes a type expression X over arbitrary variables Λ, together with an environment which explains how to **ufold** at each of those variables.

$$\text{let}\quad \frac{\{do\text{'s}\} \quad es\ :\ \mathbf{UFolds}_{\Lambda}\ \Phi\ \vec{W}\ \vec{\boldsymbol{Z}} \qquad X\ :\ \mathsf{Ty}\ \Delta\ \Lambda\ K}{\mathbf{ufoldBody}\ \{do\text{'s}\}\ es\ X\ :\ \mathbf{UFold}_{K}\ \Phi\ (X[\vec{W}])\ (X^{n}[\vec{\boldsymbol{Z}}]^{n})}$$

Note that **UFold** itself expands to a polymorphic function space. Hence, this signature may be expressed equivalently in fully applied form, taking a tuple of folds for X's arguments and an inhabitant of the datatype given by applying $X[\vec{W}]$, yielding an appropriate instance of Φ.

$$\text{let}\quad \frac{\begin{array}{ll} & es\ :\ \mathbf{UFolds}_{\Lambda}\ \Phi\ \vec{W}\ \vec{\boldsymbol{Z}} \qquad X\ :\ \mathsf{Ty}\ \Delta\ \Lambda\ K \\ \{do\text{'s}\} & fs\ :\ \mathbf{UFolds}_{K}\ \Phi\ \vec{W}'\ \vec{\boldsymbol{Z}}' \qquad t\ :\ [\![(X[\vec{W}])@\vec{W}']\!]_{\delta}\end{array}}{\mathbf{ufoldBody}\ \{do\text{'s}\}\ es\ X\ fs\ t\ :\ \Phi\ ((X[\vec{W}])@\vec{W}')\ ((X^{n}[\vec{\boldsymbol{Z}}]^{n})@^{n}\vec{\boldsymbol{Z}}')}$$

The definition is now straightforward. Variables are handled by projection from the environment; for applications, we extend the arguments tuple; when we go under a constructor, the old arguments tuple becomes the new environment tuple.

$$\begin{array}{l}
\mathbf{ufoldBody}\ \{do\text{'s}\}\ es\ \ (\mathsf{V}\ v)\ \ fs\quad\ t \quad\ \mapsto\ es.v\ fs\ t \\
\mathbf{ufoldBody}\ \{do\text{'s}\}\ es\ \ (F \cdot A)\ \ fs\quad\ t \quad\ \mapsto \\
\quad \mathbf{ufoldBody}\ \{do\text{'s}\}\ es\ F\ (\mathbf{ufoldBody}\ \{do\text{'s}\}\ es\ A, fs)\ t \\
\mathbf{ufoldBody}\ \{do\text{'s}\}\ es\ \ (\mathsf{D}\ v)\ \ fs\ \ (\mathsf{con}\ t)\ \mapsto \\
\quad doCon\ v\ (\mathbf{ufoldBody}\ \{do\text{'s}\}\ fs\ (\delta v)\ ()\ t) \\
\mathbf{ufoldBody}\ \{do\text{'s}\}\ es\ (S+T)\ ()\ \ (\mathsf{inl}\ s)\ \mapsto \\
\quad doInl\ (\mathbf{ufoldBody}\ \{do\text{'s}\}\ fs\ S\ ()\ s) \\
\mathbf{ufoldBody}\ \{do\text{'s}\}\ es\ (S+T)\ ()\ \ (\mathsf{inr}\ t)\ \mapsto \\
\quad doInr\ (\mathbf{ufoldBody}\ \{do\text{'s}\}\ fs\ T\ ()\ t) \\
\mathbf{ufoldBody}\ \{do\text{'s}\}\ es\quad 1\quad\ ()\quad \mathsf{void}\quad \mapsto\ doVoid \\
\mathbf{ufoldBody}\ \{do\text{'s}\}\ es\ (S \times T)\ ()\ (\mathsf{pair}\ s\ t) \mapsto \\
\quad doPair\ (\mathbf{ufoldBody}\ \{do\text{'s}\}\ fs\ S\ ()\ s) \\
\qquad\qquad (\mathbf{ufoldBody}\ \{do\text{'s}\}\ fs\ T\ ()\ t)
\end{array}$$

When we come to define **ufold** in terms of **ufoldBody**, we should like simply to instantiate the latter, taking Λ to be ε and es to be $()$. Unfortunately, this does not quite work for an annoying technical reason: substitution is defined by recursion over type expressions, so as to commute with constructors of Ty. Hence, the typechecker cannot tell that the trivial substitution on closed expressions,

$$\iota \;:\; \forall_{K:\Box}.\, \mathsf{Var}\,\varepsilon\, K \;\to\; \mathsf{Ty}_{\triangle}\, K$$

is, in fact, the identity. We may, however, *prove* this fact.

$$\text{let} \quad \frac{X \;:\; \mathsf{Ty}_{\triangle}\, K}{\mathbf{idLemma}\, X \;:\; X[\iota] = X}$$

The proof goes by induction on X. With this knowledge, we may explain to the typechecker that a **UFold** for $X[\iota]$ is a **UFold** for X. This $=$ is 'propositional equality', a relation internal to our type system. It has a congruence property which may be used to 'rewrite' types:

$$\frac{q \;:\; x = y \qquad t \;:\; T\, x}{q \triangleright t \;:\; T\, y}$$

Hence we may define **ufold** in terms of **ufoldBody** as follows:

$$\mathbf{ufold}\, \{do\text{'s}\}\, X \;\mapsto\; \mathbf{idLemma}\, X \triangleright \mathbf{ufoldBody}\, \{do\text{'s}\}_{\iota\, \iota^n}\, ()\, X$$

All that remains is to define **fold** by currying **ufold**. That is, we must compute a **Fold** from a **UFold**. In fact, given the recursive structure of folds, we shall need to go both ways:

$$\text{let} \quad \frac{u \;:\; \mathbf{UFold}_K\, \Phi\, X\, \vec{Y}}{\mathbf{Curry}_K\, u \;:\; \mathbf{Fold}_K\, \Phi\, X\, \vec{Y}}$$

$$\frac{f \;:\; \mathbf{Fold}_K\, \Phi\, X\, \vec{Y}}{\mathbf{Uncurry}_K\, f \;:\; \mathbf{UFold}_K\, \Phi\, X\, \vec{Y}}$$

$$\begin{array}{l}
\mathbf{Curry}_{\star} \quad u \;\mapsto\; u() \\
\mathbf{Curry}_{J \Rightarrow K}\; u\, f \mapsto \\
\quad \mathbf{Curry}_K\, (\lambda_{\vec{W}}.\, \lambda_{\vec{Z}}.\, \lambda us.\, u\, (\mathbf{Uncurry}_J\, f, us))
\end{array}$$

$$\begin{array}{l}
\mathbf{Uncurry}_{\star} \quad f \quad () \quad \mapsto f \\
\mathbf{Uncurry}_{J \Rightarrow K}\, f\, (u, us) \mapsto \\
\quad \mathbf{Uncurry}_K\, (f\, (\mathbf{Curry}_J\, u))\, us
\end{array}$$

Hence we may define

$$\begin{array}{l}
\mathbf{fold}_X\; doCon\; doInl\; doInr\; doVoid\; doPair \;\mapsto \\
\quad \mathbf{Curry}\, (\mathbf{ufold}\; doCon\; doInl\; doInr\; doVoid\; doPair\; X)
\end{array}$$

4. Conclusions and further work

The code we have presented in this paper shows how a programming language with dependent types can 'swallow' another type system by means of a universe construction, simply by writing down its typing rules as the definition of an inductive family. Generic programming within that universe comes from the same notion of computation on codes for types that we use for ordinary data—codes for types *are* ordinary data. To some extent, our work represents not just an implementation of generic programming, but a formally checked presentation of its theory.

We chose the concrete datatypes of Haskell as our example, delivering power comparable to that of Generic Haskell [CHJ+01], but we could equally have chosen Tullsen's calculus of polyadic functions [Tul00] or Pierce and Hosoya's language of generic operations for valid XML [HVP00]. With dependently types, generic programming is just programming: it is not necessary to write a new compiler each time a useful universe presents itself. Moreover, any instance of a generic program can be statically checked and should not introduce any overhead at runtime.

In contrast to Generic Haskell where a specific type checker has to be implemented, in our approach type checking comes for free because we exploit the stronger type system of our ambient language. Note that the often cited *undecidability of type checking* [Aug98] is not an issue here, because we do not introduce partial functions as indices of types.

This paper's treatment of generic programming over concrete Haskell datatypes delivers a basic functionality, but some additional flexibility is clearly desirable and remains the subject of further work. At present, our generic operations, such as **map** and $==$, yield a standard behaviour derived systematically. Ideally, we should be able to override this behaviour on an ad hoc basis. It is straightforward to adapt the existing code, parametrising each operation by its instances for each datatype and allowing us to plug in either the standard behaviour or our own.

A further concern is that our generic operations at present apply only to data which is 'internal' to the inductive family $[\![T]\!]_\delta$. We should also like to profit from this genericity when manipulating the 'external' datatypes of which our universe makes copies—our programming language's native $\mathbb{N}$, Lam, Bush and so on. We need the means to treat the internal copy as a *view*, in Wadler's sense [Wad87], of the external structure. The same is true of Generic Haskell, where the isomorphisms between Haskell datatypes and their standardized sum-of-products presentations are constructed automatically.

Dependent types may make this construction easier, for as McBride and McKinna have shown, views are already first class citizens of a dependently typed language [MM01]—dependency on terms allows *admissible* notions of pattern matching to be specified by types. We plan to integrate this technology with universe constructions, defining generic programs abstractly over any type which admits the constructors and pattern matching behaviour appropriate to a given datatype code.

However, the key advantage of the dependently typed approach is this: it respects the reality that different generic programs work over different universes. Here, we implemented the *concrete* datatypes, which admit generic $==$ and **member** operations. We could readily extend this universe to include *positive* datatypes by adding function spaces in the appropriate way—losing $==$ and **member**, but retaining **map**. We could also shrink it to just the *finite* datatypes, allowing the **listAllElements** operation. Indeed, we can restrict a universe simply by retraction over just the required type codes. For example, we can recover the Fin universe via an embedding in $\mathbb{N} \to \mathsf{Ty}_{\Delta}\,\star$, and hence restore the structure missing to the internal **Lam** in $\star \Rightarrow \star$.

We contrast this freedom with the 'compiler extension' approach, which privileges the one universe supported by the implementation. This universe is necessarily as large as possible, resulting in operations which are undefined at some of types it contains. Although the corresponding type-level 'match exceptions' are trapped at compile-time, the types given to these operations promise more than can be delivered. Some mitigation is, perhaps, offered by the type class mechanism, which can be used to constrain the choice of types to smaller sets. But, as McBride amply demonstrates in [McB02], type classes do not deliver the power of inductive families.

By allowing programmers to construct their own universes, we leave open the question of which universes provide the genericity which is relevant to a particular problem. Indeed, this paper provides considerable evidence that genericity over nested types is unlikely to be particularly useful for dependently typed programming—the datatypes we routinely exploit are indexed over much smaller sets than $\star$, and are hence more precisely structured. Whilst it has been shown that invariants such as squareness of matrices and balancing of trees can be enforced by nested types [Oka99], it takes an impressive degree of ingenuity to deliver what are basically trivial instances of dependent types. We have yet to see which universes capture the classes of dependent datatypes over which we may wish to abstract.

This work leads us to believe that dependent types provide a natural setting within which existing and novel notions of genericity can be expressed and implemented. The theory of dependent type systems is mature. Indeed, the proposers of a number of type system extensions have already observed the power of that theory in the explanation of their work, even if they stop short of exploiting its full power [PM97, Tul00, HJL02, RJ01]. Currently, the greatest obstacle for using dependent types is the shortage of compilers and programming environments which support them effectively. Removing that obstacle is what the authors of this paper are about. You have nothing to lose but your chains, you have universes to gain.

References

[AM02] Thorsten Altenkirch and Conor McBride. OLEG code for *Generic Programming Within Dependently Typed Programming*. Available from `http://www.dur.ac.uk/c.t.mcbride/generic/`, 2002.

[AR99] Thorsten Altenkirch and Bernhard Reus. Monadic presentations of lambda-terms using generalized inductive types. In *Computer Science Logic 1999*, 1999.

[Aug98] Lennart Augustsson. Cayenne—a language with dependent types. In *ACM International Conference on Functional Programming*. ACM, September 1998.

[Ben02] Marcin Benke. Towards generic programming in Type Theory. Talk at the workshop TYPES 2002, Berg en Dal, Netherlands, April 2002.

[BJJM98] Roland Backhouse, Patrik Jansson, Johan Jeuring, and Lambert Meertens. Generic Programming—An Introduction. In S. Doaitse Sweierstra, Pedro R. Henriques, and José N. Oliveira, editors, *Advanced Functional Programming, Third International Summer School (AFP '98); Braga, Portugal*, LNCS 1608, pages 28–115. Springer-Verlag, 1998.

[BP99] Richard Bird and Ross Paterson. de Bruijn notation as a nested datatype. *Journal of Functional Programming*, 9(1):77–92, 1999.

[CHJ+01] Dave Clarke, Ralf Hinze, Johan Jeuring, Andres Löh, and Jan de Wit. The Generic Haskell user's guide. Technical Report UU-CS-2001-26, Utrecht University, 2001.

[Coq92] Thierry Coquand. Pattern Matching with Dependent Types. In *Proceedings of the Logical Framework workshop at Båstad*, June 1992.

[Dyb91] Peter Dybjer. Inductive Sets and Families in Martin-Löf's Type Theory. In G. Huet and G. Plotkin, editors, *Logical Frameworks*. CUP, 1991.

[Hin00] Ralf Hinze. Generic programs and proofs. Habilitationsschrift, Universität Bonn, 2000.

[HJL02] Ralf Hinze, Johan Jeuring, and Andres Löh. Type-indexed data types. In *Mathematics of Progam Construction*, LNCS 2386, pages 148–174, 2002.

[HP00] Ralf Hinze and Simon Peyton Jones. Derivable type classes. In Graham Hutton, editor, *Proceedings of the Haskell Workshop 2000*, 2000.

[HVP00] Haruo Hosoya, Jerome Vouillon, and Benjamin C. Pierce. Regular expression types for XML. In *International Conference on Functional Programming*, pages 11–22, 2000.

[JBM98] C. Barry Jay, Gianna Belle, and Eugenio Moggi. Functorial ML. *Journal of Functional Programming*, 8(6):573–619, 1998.

[JJ97] Patrik Jansson and Johan Jeuring. PolyP—a polytypic programming language extension. In *Proceedings of POPL '97*, pages 470–482. ACM, January 1997.

[LP92] Zhaohui Luo and Randy Pollack. LEGO Proof Development System: User's Manual. Technical Report ECS-LFCS-92-211, Laboratory for Foundations of Computer Science, University of Edinburgh, May 1992.

[McB99] Conor McBride. *Dependently Typed Functional Programs and their Proofs*. PhD thesis, University of Edinburgh, 1999.

[McB01] Conor McBride. The Derivative of a Regular Type is its Type of One-Hole Contexts. Electronically available, 2001.

[McB02] Conor McBride. Faking It: Simulating Dependent Types in Haskell. *J. Functional Programming*, 2002. Accepted; to appear.

[ML84] Per Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, Napoli, 1984.

[MM01] Conor McBride and James McKinna. The view from the left. Submitted to the Journal of Functional Programming, Special Issue: Dependent Type Theory Meets Programming Practice, December 2001.

[NPS90] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory. An Introduction*. OUP, 1990.

[Oka99] Chris Okasaki. From Fast Exponentiation to Square Matrices: An Adventure in Types. In *ACM International Conference on Functional Programming '99*, 1999.

[PH+99] Simon Peyton Jones, John Hughes, et al. Haskell 98: A non-strict purely functional language. Available from: http://www.haskell.org/, 1999.

[PM97] Simon Peyton Jones and Erik Meijer. Henk: a typed intermediate language, 1997. ACM Workshop on Types in Compilation.

[PR98] H. Pfeifer and H. Rueß. Polytypic abstraction in type theory. In Roland Backhouse and Tim Sheard, editors, *Workshop on Generic Programming (WGP'98)*. Dept. of Computing Science, Chalmers Univ. of Techn. and Göteborg Univ., June 1998.

[RJ01] Jan-Willem Roorda and Johan Jeuring. Pure type systems for functional programming. In preparation, 2001.

[Tul00] Mark Tullsen. The Zip Calculus. In Roland Backhouse and José Nuno Oliviera, editors, *Mathematics of Program Construction*, LNCS 1837, pages 28–44. Springer-Verlag, 2000.

[Wad87] Philip Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *POPL'87*. ACM, 1987.