Agda-ventures with PolyP

Jeremy Gibbons[1][0000-0002-8426-9917] and
Patrik Jansson[2][0000-0003-3078-1437]

[1] University of Oxford, UK
https://www.cs.ox.ac.uk/people/jeremy.gibbons/
[2] Chalmers University of Technology and University of Gothenburg, SE
https://patrikja.owlstown.net/

Abstract. Revisiting Johan Jeuring's PolyP 30 years on, we note that a special-purpose language is no longer needed: general-purpose dependently typed programming suffices. This is a text-based adventure from software archeology, via codes to universes. Happy 60th Birthday, Johan!

1 Introduction

Among Johan Jeuring's contributions to the world, not the least is his programming language PolyP, developed in a series of papers from 1995 to 2002. One of us was his first PhD student, and part of this endeavour.

PolyP was a research language designed for the purpose of exploring the notion of *polytypic programming*: programs that are parametrized by the shape of datatypes, so that one program can be applied to many different datatypes. In the first paper on the topic [7], Jeuring quotes the definition from Webster's dictionary:

> **poly·typ·ic** [ˌpä-lē-ˈti-pik], *adj.*: having or involving several different types

Other names for the same idea include 'structurally polymorphic', 'shape polymorphic, 'type parametric', 'generic', and 'datatype-generic'. Typical polytypic programming problems are structural: equality, matching, folding, mapping, traversal, encoding, printing, parsing, unification, and so on. A crucial criterion is the maintenance of strong static type safety; in contrast, approaches based on dynamic typing may be able to express the same programs, but cannot make the same static guarantees.

PolyP was implemented [5] as a preprocessor for Haskell, providing an additional polytypic construct that gets translated into ordinary Haskell. (The source code is available at GitHub [3]. The original revision history has been preserved, predating GitHub's birth by a decade.) The work on PolyP led to a grant from the Dutch research council NWO for the *Generic Haskell* project, running 2000–2004 [8], another preprocessor for Haskell, and then in turn to many different approaches to generic programming [1].

So the ideas involved in PolyP have been influential over the past thirty years or so. But they have also been superseded by developments in programming languages. In particular, what in 1995 required a domain-specific language and a special-purpose preprocessor can be achieved in 2025 by good old-fashioned programming. This has been enabled by the advances that have since been made in *dependent types*. Whilst this theory significantly predates PolyP, it is only recently that tools originally envisioned as supporting theorem proving and formalized mathematics have become plausible programming languages.

In this short paper, we summarize the key ideas behind polytypic programming, and show how they can now be captured directly in a dependently typed programming language. Any dependently typed language will do, but we will use Agda. Maybe we can entice you back, Johan? The water is much warmer these days!

2 Polytypic programming

The general idea with PolyP is that "a polytypic function can be viewed as a family of functions: one function for each datatype" [9], defined by induction over the structure of the datatype. So first one needs to settle on the universe of datatypes.

PolyP used *polynomial types*: sums and products of some basic types, such as booleans, integers, and the unit type. For recursive datatypes such as lists of booleans and trees of integers, it used *regular functors*: initial algebras for functors constructed from polynomial operations on a type parameter, closed under certain compositions (so that one recursive datatype can be used in the shape functor for another). And to accommodate polymorphic (container) datatypes too, it extended to *regular bifunctors*.

For example [9], the Haskell datatypes of lists and rose trees

$$\begin{aligned}&\mathbf{data}\ \mathit{List}\ a\ \ = \mathit{Nil} \mid \mathit{Cons}\ a\ (\mathit{List}\ a)\\ &\mathbf{data}\ \mathit{Rose}\ a = \mathit{Fork}\ a\ (\mathit{List}\ (\mathit{Rose}\ a))\end{aligned}$$

are the initial algebras respectively of the bifunctors written in PolyP as

$$\begin{aligned}&\mathit{FList}\ \ = () + \mathbf{Par} \times \mathbf{Rec}\\ &\mathit{FRose} = \mathbf{Par} \times (\mathit{List}\ @\ \mathbf{Rec})\end{aligned}$$

For *FList*, the bifunctor is a sum, with the unit type for the left summand; the right summand is the product of the datatype parameter (that is, the first bifunctor argument) and a recursive call (the second bifunctor argument). For *FRose*, the right factor is the composition of *List* and the recursive position.

Continuing the example from [9], inductive datatypes $\mathbf{Mu}\ F\ A$ for bifunctor F and element type A have a constructor and a destructor:

$$\begin{aligned}&\mathit{inn} :: f\ a\ (\mathbf{Mu}\ f\ a) \to \mathbf{Mu}\ f\ a\\ &\mathit{out} :: \mathbf{Mu}\ f\ a \to f\ a\ (\mathbf{Mu}\ f\ a)\end{aligned}$$

A polytypic 'map' function for inductive datatypes

$$
\begin{aligned}
&\mathit{pmap} \quad :: (a \to b) \to \mathbf{Mu}\ f\ a \to \mathbf{Mu}\ f\ b \\
&\mathit{pmap}\ p = \mathit{inn} \cdot \mathit{fmap}\ p\ (\mathit{pmap}\ p) \cdot \mathit{out}
\end{aligned}
$$

is defined in terms of a polytypic *fmap* for regular bifunctors:

$$
\begin{aligned}
&\mathbf{polytypic}\ \mathit{fmap} :: (a \to c) \to (b \to d) \to f\ a\ b \to f\ c\ d \\
&\quad = \lambda\ p\ r \to \mathbf{case}\ f\ \mathbf{of} \\
&\qquad\quad f + g \ \to \mathit{fmap}\ p\ r \mathbin{\text{-+-}} \mathit{fmap}\ p\ r \\
&\qquad\quad () \qquad \to \mathit{id} \\
&\qquad\quad \mathbf{Con}\ t \to \mathit{id} \\
&\qquad\quad f \times g \ \to \mathit{fmap}\ p\ r \mathbin{\text{-}\times\text{-}} \mathit{fmap}\ p\ r \\
&\qquad\quad d \mathbin{@} g \ \to \mathit{pmap}\ (\mathit{fmap}\ p\ r) \\
&\qquad\quad \mathbf{Par} \quad \to p \\
&\qquad\quad \mathbf{Rec} \quad \to r
\end{aligned}
$$

where (-+-) and (-×-) map over sums and products respectively.

Note that there is no *Functor* or *Bifunctor* type class constraint on f, requiring a separate instance declaration, as would typically be the case in Haskell. Rather, the definition of *fmap* is essentially the template by which GHC would automatically *derive* a functor instance.

3 Dependently typed programming

In PolyP, a polytypic definition like that of *fmap* specifies what code should be generated for a specific type: "the compiler generates instances from the definition of the polytypic function and the type in the context where it is used" [5]. This is more than mere text processing, because PolyP does take care to type check a polytypic definition, in the sense that no generated instance will yield a Haskell type error. Still, PolyP is essentially a standalone domain-specific language for polytypic definitions, which means that the full power of the target language Haskell is not available in polytypic code.

This is a price that need not be paid, provided one can find a single language expressive enough to encompass both the polytypic templates and the actual eventual code. Then a separate code generation phase is not required: it becomes "a small matter of programming" in the one language. It turns out that a dependently typed language like Agda [13] provides the expressivity needed: types (and operations on types, such as functors and bifunctors) are values too.

So what would in PolyP be a polytypic function parametrized by a functor becomes in Agda just a function with an argument. However, that argument can't literally be a type, or a functor. We can't work with the types themselves, because we can't analyse them: they would be black boxes, and we need to perform case analyses on them. Instead, we make separate *codes* for the types in the universe, and define the interpretation mapping codes to types. Codes *can* be analysed and manipulated, since they are just terms in an algebraic datatype.

As a simple introduction, let's consider the universe of types consisting of sums and products of the unit type, naturals, and booleans. We start with an algebraic datatype of codes for the types in the universe:

```
data Code₀ : Set where
  NatT BoolT UnitT : Code₀
  _*_ _+_ : Code₀ → Code₀ → Code₀
```

For example, here is the code for the sum of the unit type and the product of naturals and booleans (that is, what would be *Maybe* (*Nat*, *Bool*) in Haskell):

```
MaybeNatBoolCode : Code₀
MaybeNatBoolCode = UnitT + (NatT * BoolT)
```

We can then define the interpretation of codes as types:

```
⟦_⟧₀ : Code₀ → Set
⟦ NatT ⟧₀   = Nat
⟦ BoolT ⟧₀  = Bool
⟦ UnitT ⟧₀  = ⊤
⟦ c * c′ ⟧₀ = ⟦ c ⟧₀ × ⟦ c′ ⟧₀
⟦ c + c′ ⟧₀ = Sum ⟦ c ⟧₀ ⟦ c′ ⟧₀
```

This interpretation is simply a straightforward function definition—we have exploited Agda's fancy mix-fix syntax, but we might as well have named the function something like "interp0". The definition is by induction over the structure of codes: interpretations of the three base type codes are given directly ("⊤" denotes the unit type, with sole element tt), and interpretations of the product and sum code constructors given inductively ("×" and "Sum" denote product and sum types respectively).

Finally, we can define a polytypic function over this universe of types. For example, here is the equality function: it takes the code for some type in the universe, and two elements of the interpretation of that code, and returns a boolean. For the three base cases (constant types), the comparison is delegated to type-specific operators; for the two inductive cases (product and sum), it is given inductively.

```
equal₀ : { c : Code₀ } → ⟦ c ⟧₀ → ⟦ c ⟧₀ → Bool
equal₀ {NatT}  n        m        = (n ==ₙ m)
equal₀ {BoolT} x        y        = (x ==_B y)
equal₀ {UnitT} x        y        = (x ==_U y)
equal₀ {c * c′} (x , x′) (y , y′) = equal₀ x y ∧ equal₀ x′ y′
equal₀ {c + c′} (inj₁ x) (inj₁ y) = equal₀ x y
equal₀ {c + c′} (inj₂ x′) (inj₂ y′) = equal₀ x′ y′
equal₀ {c + c′} _        _        = false
```

For example, the two elements inj₁ tt and inj₂ (3 , false) in the interpretation of MaybeNatBoolCode are not equal; and indeed, the expression

```
equal₀ { MaybeNatBoolCode } (inj₁ tt) (inj₂ (3 , false))
```

normalizes to false. Note that the first argument to equal$_0$ is written in curly braces, marking it as *implicit*, since it can be inferred. In particular, it is omitted for the recursive calls, and not needed for the example either: we can write just equal$_0$ (inj$_1$ tt) (inj$_2$ (3 , false)).

4 The full story

So much for codes for types, and their interpretation as actual types. If we want to handle inductive datatypes as fixpoints, we also want codes for functors. And for polymorphic inductive datatypes, we want bifunctors. So here are three mutually recursive datatypes of codes for them:

```
mutual
  data Type : Set where
    NatTy BoolTy UnitTy : Type

  data Functor : Set where
    Fix : Bifunctor → Functor

  data Bifunctor : Set where
    _*_ _+_ : Bifunctor → Bifunctor → Bifunctor
    Const   : Type                  → Bifunctor
    _•_     : Functor   → Bifunctor → Bifunctor
    Par Rec :                         Bifunctor
```

In order to interpret codes for inductive datatypes, we need to define these:

```
{-# NO_POSITIVITY_CHECK #-}
data Mu (f : Set → Set) : Set where
  In : f (Mu f) → Mu f

out : { f : Set → Set } → Mu f → f (Mu f)
out (In xs) = xs
```

Not all functors induce inductive datatypes, so we have to turn off the check that Agda would otherwise insist on. Since we are modelling PolyP generating Haskell, we don't worry too much about the risk of non-termination.

We can now give the interpretations of the three kinds of code:

```
mutual
  [[_]]T : Type      →               Set
  [[_]]F : Functor   → Set →         Set
  [[_]]B : Bifunctor → Set → Set →   Set

  [[ NatTy ]]T  = Nat
  [[ BoolTy ]]T = Bool
```

```
⟦ UnitTy ⟧T = ⊤
⟦ Fix f ⟧F p = Mu (⟦ f ⟧B p)

⟦ f ∗ g ⟧B   p r = ⟦ f ⟧B p r × ⟦ g ⟧B p r
⟦ f + g ⟧B   p r = Sum (⟦ f ⟧B p r) (⟦ g ⟧B p r)
⟦ Const t ⟧B p r = ⟦ t ⟧T
⟦ d • f ⟧B   p r = ⟦ d ⟧F (⟦ f ⟧B p r)
⟦ Par ⟧B     p r = p
⟦ Rec ⟧B     p r = r
```

Each base code is interpreted as the corresponding base type. Our only code for a functor is for a polymorphic inductive datatype, which is interpreted accordingly, using the interpretation of its bifunctor parameter. Bifunctor codes for lifted product and sum of two bifunctors are interpreted using the standard constructors; the codes for a constant bifunctor and for the composition of a functor and a bifunctor ("@" in PolyP, which is reserved in Agda so written with a bullet here) are defined recursively; and the 'parameter' and 'recursive argument' are projections.

Next we can define the functorial action for functors and bifunctors (the *pmap* and *fmap* we saw above), mutually recursive with catamorphisms:

```
mutual
  {-# TERMINATING #-}
  pmap : (d : Functor)      → (a → b) → ⟦ d ⟧F a → ⟦ d ⟧F b
  fmap : (f : Bifunctor)    → (a → b) → (c → d) → ⟦ f ⟧B a c → ⟦ f ⟧B b d
  cata : (f : Bifunctor)    → (⟦ f ⟧B a b → b) → ⟦ Fix f ⟧F a → b

  cata f h (In xs)          = h (fmap f id (cata f h) xs)

  pmap (Fix f) g            = cata f (In ∘ fmap f g id)

  fmap (f ∗ g) p r (x , y)  = (fmap f p r x , fmap g p r y)
  fmap (f + g) p r (inj1 x) = inj1 (fmap f p r x)
  fmap (f + g) p r (inj2 y) = inj2 (fmap g p r y)
  fmap (Const t) p r x      = x
  fmap (d • g) p r xs       = pmap d (fmap g p r) xs
  fmap Par p r              = p
  fmap Rec p r              = r
```

For example, the code ListF for the shape bifunctor for lists, the corresponding code ListC for its fixpoint, and the interpretation MyList of the latter as an actual functor are:

```
ListF : Bifunctor
ListF = Const UnitTy + (Par ∗ Rec)

ListC : Functor
ListC = Fix ListF
```

```
MyList : Set → Set
MyList = ⟦ ListC ⟧F
```

We can define constructors for these lists:

```
nilList : MyList a
nilList = In (inj₁ tt)

consList : a → MyList a → MyList a
consList x xs = In ( inj₂ ( ( x , xs )))
```

and conversion functions from and to built-in lists:

```
toMyList : List a → MyList a
toMyList = foldr consList nilList

fromMyList : MyList a → List a
fromMyList = cata ListF alg where
  alg : ⟦ ListF ⟧B a (List a) → List a
  alg (inj₁ tt)          = []
  alg (inj₂ ( x , xs )) = x :: xs
```

One canonical example of a polytypic function on polymorphic container datatypes is to "crush" it [11], aggregating the elements using a monoid:

```
mutual
  crush : (a → a → a) → a → (d : Functor) → ⟦ d ⟧F a → a
  crush  _⊕_ e (Fix f) = cata f (crushB _⊕_ e f)

  crushB : (a → a → a) → a → (f : Bifunctor) → ⟦ f ⟧B a a → a
  crushB _⊕_ e (f ∗ g)   (x , y)  = crushB _⊕_ e f x ⊕ crushB _⊕_ e g y
  crushB _⊕_ e (f + g)   (inj₁ x) = crushB _⊕_ e f x
  crushB _⊕_ e (f + g)   (inj₂ y) = crushB _⊕_ e g y
  crushB _⊕_ e (Const t) x        = e
  crushB _⊕_ e Par       p        = p
  crushB _⊕_ e Rec       r        = r
  crushB _⊕_ e (d • g)   = crush _⊕_ e d ∘ pmap d (crushB _⊕_ e g)
```

The binary operator is used to combine the two aggregations in a product, and the unit value is used for constants. For instance, we can flatten a container to a list, by making every element a singleton list then crushing using the list monoid:

```
flatten : (d : Functor) → ⟦ d ⟧F a → List a
flatten d = crush _++_ [] d ∘ pmap d (λ x → [ x ])
```

5 Polytypic packing and unpacking

Let us now look at a more extended example: another canonical piece of the polytypism literature, namely polytypic *packing*. By this we mean encoding a value

of arbitrary type as a bitstream, in such a way as to be able (given information also about the type) to decode the bitstream back to the original data.

One can think of this as simple-minded data compression. For simplicity, we will encode to lists of bits and ignore the possible refinement of packing the bits into words. We name the bits used to label left and right choices, and provide a case analysis for them:

```
leftBit rightBit : Bool
leftBit  = false
rightBit = true

caseBit : Bool → a → a → a
caseBit b x y = if b then y else x
```

the idea being that

```
b = caseBit b leftBit rightBit
```

We provide primitives toBits and fromBits to convert between natural numbers and lists of booleans (making the simplifying assumption that all numbers are distinct to bitWidth bits—we could be cleverer about this):

```
bitWidth : Nat
bitWidth = 4 - we keep it small for testing

toBits : Nat → List Bool
toBits n = reverse (go bitWidth n) where
  go : Nat → Nat → List Bool
  go zero n      = []
  go (suc m) n = let (q , r) = divMod2 n in r :: go m q

fromBits : List Bool → Nat
fromBits = foldl (λ n b → (2 *N n) +N (if b then 1 else 0)) 0
```

Now, as a first attempt we might represent a packer for data as a function from that data to lists of bits. However, an unpacker would have to be more than simply a function in the opposite direction: we have to return the unused bits too, in order to be compositional; and we have to allow for failure, to make a total function. So we define the following:

```
Unpacker : Set → Set
Unpacker a = List Bool → Maybe (a × List Bool)
```

In fact, that type supports a monad—the combination of the state monad transformer on bit-list state around the maybe monad, what would in Haskell be written *StateT* [*Bool*] *Maybe*:

```
P : Set → Set
P = Unpacker
```

It turns out to be convenient to define packer using the same type:

```
Packer : Set → Set
Packer a = a → P ⊤
```

A packer will always succeed, and will produce rather than consume some bits. In that sense, the P monad is overkill—but it allows u now to think about composing packers and unpackers. Note that an unpacker Unpacker A for some type A is isomorphic to a function of type $\top \to$ P A, which is nicely dual to the packer type $A \to$ P $\top$. (For more on this duality, see [6].)

The stateful interface is provided by two operations:

```
put : Packer (List Bool)
put bs' = λ bs → just (tt , bs')

get : Unpacker (List Bool)
get = λ bs → just (bs , bs)
```

But we will not need the full power of these two operations; we will use them only in restricted ways. For packers, we need only add some output:

```
packerTell : Packer (List Bool)
packerTell bs = do
  bs' ← get
  put (bs ++ bs')
```

(for reasons that will become clear in due course, we prepend rather than append). In particular, here are primitive packers for naturals, booleans, and the unit type:

```
packNat : Packer Nat
packNat n = packerTell (toBits n)

packBool : Packer Bool
packBool b = packerTell [ b ]

packUnit : Packer ⊤
packUnit tt = return tt
```

Note that packUnit is a no-op. Primitive unpackers for naturals and units are similarly easy:

```
unpackBool : Unpacker Bool
unpackBool = uncons

unpackUnit : Unpacker ⊤
unpackUnit = return tt
```

To unpack a natural, we read a chunk of input, then convert these bits to a number:

```
unpackNat : Unpacker Nat
unpackNat = do
```

```
  xs ← get
  let (ys , zs) = splitAt bitWidth xs
  put zs
  return (fromBits ys)
```

5.1 Packing as a monadic catamorphism

Now, to pack a structure of an inductive datatype, we will use a *monadic catamorphism* [12], which is like the ordinary catamorphism except that the algebra argument and the catamorphism itself are *Kleisli arrows*—that is, they have a monadic return type:

```
mutual
  cataM : (f : Bifunctor) → (⟦ f ⟧B a b → P b) → ⟦ Fix f ⟧F a → P b
```

We will return to the definition of cataM shortly; but let's first see how it is used.

We define three mutually recursive functions, packers respectively for a type, a functor, and a bifunctor, taking correspondingly many element packers as arguments:

```
packT : (t : Type)      →                                  Packer ⟦ t ⟧T
packF : (d : Functor)   → Packer a →                       Packer (⟦ d ⟧F a)
packB : (f : Bifunctor) → Packer a → Packer b →            Packer (⟦ f ⟧B a b)
```

The packers for types each delegate to the appropriate primitive defined earlier:

```
packT NatTy n     = packNat n
packT BoolTy b    = packBool b
packT UnitTy tt   = packUnit tt
```

We only have one code for functors, interpreted as an inductive datatype, and this is where we use the monadic catamorphism:

```
packF (Fix f) p = cataM f (packB f p packUnit)
```

The catamorphism handles the recursive calls, so at the top level we do nothing (packUnit) for the recursive positions.

Finally, we have one fairly simple case per bifunctor:

```
packB (f * g)   p q ( x , y ) = do packB g p q y
                                   packB f p q x
packB (f + g)   p q (inj₁ x)  = do packB f p q x
                                   packerTell [ leftBit ]
packB (f + g)   p q (inj₂ y)  = do packB g p q y
                                   packerTell [ rightBit ]
packB (Const t) p q           = packT t
```

```
packB (d • g)   p q   = packF d id ∘ pmap d (packB g p q)
packB Par       p q   = p
packB Rec       p q   = q
```

Recall that the primitive operations to write bits were defined to prepend to the list. We therefore specify that for products, we pack the right then the left component of the pair; then the left component will appear first in the output. Similarly—and more importantly—for sums, we emit the discriminator bit after packing the payload.

Now back to the monadic catamorphism. This requires a *distributive law* of the shape bifunctor over the monad—informally, this hoists the monad to the top, executing the computations for each of the recursive positions to make one composite computation collecting all the effects:

```
distr : (f : Bifunctor) → ⟦ f ⟧B a (P b) → P (⟦ f ⟧B a b)
```

Then the catamorphism deconstructs the data, makes recursive calls on each of the children, collects all their effects, applies the algebra h, and merges the effects of that:

```
{-# TERMINATING #-}
cataM f h (In xs) = join (h <$> (distr f (fmap f id (cataM f h) xs)))
```

Note that the catamorphism is bottom up: the effects from children are incurred before those of the parent. This is why we defined the primitive packers to prepend bits instead of appending them: the encoding of the root of the data structure will end up at the start of the output list, conveniently for unpacking.

The last ingredient is the distributive law. This is in essence another polytypic program, with mutually recursive definitions for functors and bifunctors (types are not needed):

```
distrF : (f : Functor)   → ⟦ f ⟧F (P a)       → P (⟦ f ⟧F a)
distrB : (f : Bifunctor) → ⟦ f ⟧B (P a) (P b) → P (⟦ f ⟧B a b)

distrF (Fix f) (In xs)    = In <$> (distrB f (fmap f id (distrF (Fix f) ) xs))

distrB (f ∗ g) (xs , ys) = liftM2 _,_ (distrB f xs) (distrB g ys)
distrB (f + g) (inj₁ x)  = inj₁ <$> distrB f x
distrB (f + g) (inj₂ y)  = inj₂ <$> distrB g y
distrB (Const t) x       = return x
distrB (d • g) xs        = distrF d (pmap d (distrB g) xs)
distrB Par x             = x
distrB Rec x             = x
```

The variant we actually use is for bifunctors, but with pure values in the parameter positions, so we must first inject these into the monad:

```
distr f = distrB f ∘ fmap f return id
```

For example, with bit width set to 4 for brevity, the expression

```
packF ListC (packT NatTy) (toMyList (1 :: 2 :: 3 :: [])) []
```

reduces to the expression in Figure 1.

```
just (tt ,
  true ::
  false :: false :: false :: true :: - toBits 1
  true ::
  false :: false :: true :: false :: - toBits 2
  true ::
  false :: false :: true :: true :: - toBits 3
  false :: [])
```

Fig. 1. The list of bits resulting from packing 1, 2, 3 is true to indicate a cons cell, then four bits representing the number 1, then similarly for the next two elements, then false to indicate a nil cell.

5.2 Unpacking as a monadic anamorphism

Let us now turn to unpacking. Being the inverse of packing, it will use the dual pattern: a *monadic anamorphism*, which is again like the ordinary anamorphism only where the coalgebra and the anamorphism itself are Kleisli arrows:

```
mutual
  {-# TERMINATING #-}
  anaM : (f : Bifunctor) → (b → P ([[ f ]]B a b)) → b → P ([[ Fix f ]]F a)
```

Unpacking is again defined in terms of three mutually recursive functions, unpackers respectively for types, functors, and bifunctors, each with the corresponding number of element unpackers as arguments:

```
unpackT : (t : Type)                    →          Unpacker [[ t ]]T
unpackF : (d : Functor)                 →
          Unpacker a →                             Unpacker ([[ d ]]F a)
unpackB : (f : Bifunctor)               →
          Unpacker a → Unpacker b →     Unpacker ([[ f ]]B a b)
```

For the base types we defer to the earlier primitives:

```
unpackT NatTy        = unpackNat
unpackT BoolTy       = unpackBool
unpackT UnitTy       = unpackUnit
```

For the sole functor code, we use the anamorphism:

```
unpackF (Fix f) u = anaM f (λ _ → unpackB f u unpackUnit) _
```

Note that the 'seed' of the anamorphism is the unit type: all the information driving the computation comes from the list of booleans, encoded in the monad. So the bound variable of the lambda is irrelevant, and the initial seed of the anamorphism can be inferred. As with packing, the anamorphism handles the recursive calls, so at the top level we need do nothing for the recursive positions (unpackUnit, another no-op).

And finally, there is one fairly simple case per bifunctor:

```
unpackB (f ∗ g) u v    = liftM2 _,_ (unpackB f u v) (unpackB g u v)
unpackB (f + g) u v    = do b ← unpackBool
                            caseBit b
                              (inj₁ <$> unpackB f u v)
                              (inj₂ <$> unpackB g u v)
unpackB (Const x) u v = unpackT x
unpackB (d • g) u v    = unpackF d (unpackB g u v)
unpackB Par u v        = u
unpackB Rec u v        = v
```

For products, we unpack the left then the right components; for sums, we consume one discriminator bit in order to decide which branch to take.

Now back to the monadic anamorphism. This applies the coalgebra to the seed, makes recursive calls to generate each of the children, collects all their effects, merges the effects from the coalgebra and the recursive calls, then wraps the result up in the constructor:

```
anaM f h y = In <$> join (distr f <$> (fmap f id (anaM f h) <$> h y))
```

To illustrate the round trip, it should be the case that whatever value we take, if we pack it in front of any bit sequence then unpack the resulting sequence, that composition should succeed, and should return the original value and sequence:

```
packUnpack : { a : Set } → Packer a → Unpacker a → a → List Bool → Set
packUnpack p u x bs = (p x » u) bs ≡ just (x , bs)
```

(recall that a packer returns unit, which we then discard by »). Then we can instantiate this scheme for the types, functors, and bifunctors in our universe:

```
packUnpackT : (a : Type) → ⟦ a ⟧T →                          List Bool → Set
packUnpackT a = packUnpack (packT a) (unpackT a)

packUnpackF : (d : Functor) → (a : Type) → ⟦ d ⟧F ⟦ a ⟧T → List Bool → Set
packUnpackF d a = packUnpack (packF   d (packT a)   )
                             (unpackF d (unpackT a) )
```

```
packUnpackB : (f : Bifunctor) → (a b : Type) →
                [[ f ]]B [[ a ]]T [[ b ]]T →                    List Bool → Set
packUnpackB f a b = packUnpack (packB    f (packT a)    (packT b)    )
                               (unpackB f (unpackT a) (unpackT b) )
```

For example, we can check the round trip property on a list of three naturals:

```
packUnpackList : ∀ (bs : List Bool) →
                   packUnpackF ListC NatTy (toMyList (1 :: 2 :: 3 :: [])) bs
packUnpackList bs = refl
```

The value we give to packUnpackList is simply refl, which indicates that (and is only type correct when) the two sides of the equivalence are definitionally equal.

6 Discussion

The technique we have used of identifying an algebraic datatype of *codes* for types drawn from some *universe* is a standard pattern in dependently typed programming. It is an instance in microcosm of the "formulation á la Tarski" that Martin-Löf [10] used in macrocosm to construct a universe of discourse for intuitionistic type theory. Another way of looking at it is specifying an embedded domain specific language for types (namely the codes), and semantics by way of a shallow embedding into the host language (namely the interpretation) [2].

This paper is literate Agda, although some of the gory details have been elided for presentation purposes. The full story can be seen in the source, which is available on GitHub [4], and typechecks at least with version 2.7.0.1 of Agda and version 2.2 of the Agda standard library.

We have seen that the polytypic programming features that Johan pioneered with PolyP can be done nowadays as 'mere programming', given a sufficiently rich language—in particular, a dependently typed one. We have chosen Agda, but Idris would work just as well.

Even Haskell is almost powerful enough these days: much of the PolyP functionality was achieved in Haskell already in 2003 [14]. But dependent types have the additional advantage that proofs become part of the language. We have exploited this briefly above: the code contains some unit tests, which are run as part of typechecking. And indeed we have exploited these tests while writing the programs: although many silly errors are ruled out by the types, it is in particular still possible to write out the wrong bit sequences.

But a powerful and informative type system like Agda's is not just there to prevent accidents. It is also hugely valuable when it comes to writing the programs in the first place: the type specifies much about the program, so with suitable interaction between the type checker and the editor—for example, in the Agda mode for Emacs—much of the program can be written automatically. Some values can be inferred; case analyses can be automatically generated; programs can be

typechecked while still containing holes, and these holes can be explored with information about the goal type and the variables in scope.

Programming in this type-driven style in many ways feels like a text-based adventure game. You find yourself in a hole, with various objects at your disposal, and you have to find a way out. You keep getting sent on side-quests. Sometimes it feels like you are fighting the typechecker; but sometimes it feels like the universe is on your side, and the obstacles are magically eliminated. In recent years, Johan's research interests have shifted from programming languages to technology for education, including 'serious games': perhaps Johan can see scope for closing the circle by bringing the two back together?

References

1. Garcia, R., Järvi, J., Lumsdaine, A., Siek, J.G., Willcock, J.: An extended comparative study of language support for generic programming. Journal of Functional Programming **17**(2), 145–205 (2007). https://doi.org/10.1017/S0956796806006198
2. Gibbons, J., Wu, N.: Folding domain-specific languages: Deep and shallow embeddings. In: International Conference on Functional Programming (Sep 2014). https://doi.org/10.1145/2628136.2628138
3. Jansson, P.: PolyP source code. https://github.com/patrikja/PolyP
4. Jansson, P., Gibbons, J.: Source code for the "PolyP 30" paper. https://github.com/DSLsofMath/PolyP30
5. Jansson, P., Jeuring, J.: PolyP – A polytypic programming language extension. In: Principles of Programming Languages (POPL). pp. 470–482. ACM Press (1997). https://doi.org/10.1145/263699.263763
6. Jansson, P., Jeuring, J.: Polytypic data conversion programs. Science of Computer Programming **43**, 35–75 (2002). https://doi.org/10.1016/S0167-6423(01)00020-X
7. Jeuring, J.: Polytypic pattern matching. In: Functional Programming Languages and Computer Architecture. pp. 238–248. ACM (1995). https://doi.org/10.1145/224164.224212
8. Jeuring, J.: Generic Haskell: A language for generic programming. NWO grant 612.069.000 (2000), https://www.nwo.nl/en/projects/612069000
9. Jeuring, J., Jansson, P.: Polytypic programming. In: Launchbury, J., Meijer, E., Sheard, T. (eds.) Advanced Functional Programming. Lecture Notes in Computer Science, vol. 1129, pp. 68–114. Springer-Verlag (1996). https://doi.org/10.1007/3-540-61628-4_3
10. Martin-Löf, P.: Intuitionistic Type Theory. Studies in Proof Theory, Bibliopolis (1984), notes by Giovanni Sambin of a series of lectures given in Padua, June 1980. ISBN 88-7088-105-9.
11. Meertens, L.: Calculate polytypically! In: Kuchen, H., Swierstra, S.D. (eds.) Programming Languages: Implementations, Logics, and Programs. Lecture Notes in Computer Science, vol. 1140, pp. 1–16. Springer-Verlag (1996). https://doi.org/10.1007/3-540-61756-6_73
12. Meijer, E., Jeuring, J.: Merging monads and folds for functional programming. In: Jeuring, J., Meijer, E. (eds.) Advanced Functional Programming. Lecture Notes in Computer Science, vol. 925, pp. 228–266. Springer (1995). https://doi.org/10.1007/3-540-59451-5_7

13. Norell, U.: Towards a Practical Programming Language Based on Dependent Type Theory, vol. 32. Chalmers University of Technology (2007), https://www.cse.chalmers.se/~ulfn/papers/thesis.pdf, ISBN 978-91-7291-996-9.
14. Norell, U., Jansson, P.: Polytypic programming in Haskell. In: Trinder, P., Michaelson, G.J., Peña, R. (eds.) Implementation of Functional Languages. Lecture Notes in Computer Science, vol. 3145, pp. 168–184. Springer, Berlin, Heidelberg (2005). https://doi.org/10.1007/978-3-540-27861-0_11