Generic Programming with Indexed Functors

Andres Löh
Well-Typed LLP
andres@well-typed.com

José Pedro Magalhães
Utrecht University
jpm@cs.uu.nl

Abstract

Much has been said and done about generic programming approaches in strongly-typed functional languages such as Haskell and Agda. Different approaches use different techniques and are better or worse suited for certain uses, depending on design decisions such as generic view, universe size and complexity, etc.

We present a simple and intuitive yet powerful approach to generic programming in Agda using indexed functors. We show a universe incorporating fixed points that supports composition, indexing, and isomorphisms, and generalizes a number of previous approaches to generic programming with fixed points. Our indexed functors come with a map operation which obeys the functor laws, and associated recursion morphisms. Albeit expressive, the universe remains simple enough to allow defining standard recursion schemes as well as decidable equality. As for type-indexed datatypes, we show how to compute the type of one-hole contexts and define the generic zipper.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Functional Programming

General Terms Algorithms, Languages

1. Introduction

Datatype-generic programming is a form of abstraction that allows defining a single function over an entire class of datatypes at once. Common examples are equality and serialisation, both being computations which depend only on the structure of the datatypes for which they are defined. There are several approaches to generic programming using different techniques, exploring the trade-off between complexity and expressiveness.

If the goal is to define functions that work over an entire class of datatypes, we need to somehow access the underlying structure of such datatypes. There are many ways to do so, and the choice of representation has a significant impact:

- it determines which datatypes can be described at all, and hence be in the domain of generic functions;
- it affects whether it is possible (and if so, how easy it is) to define certain generic functions on those datatypes.

In this paper, we show a particular approach which is both elegant and powerful. We use Agda, relying on the power of dependent types to encode datatypes as a universe of indexed functors. A universe is an abstract description of types as codes. Codes are then interpreted as proper types by means of an interpretation function.

Our universe is centered on the notion of an indexed functor. A normal functor is a type constructor, i.e. of Agda type Set → Set. An indexed functor allows an indexed set both as input and as output, where an indexed set is a function mapping a concrete "index" type to Set. We can then express families of mutually recursive datatypes as a single indexed functor, choosing appropriate indices. Parameterized datatypes are represented using indices as well; in this way we treat recursion and parameterization uniformly, allowing for flexible composition. Similarly to the universes of Altenkirch et al. (2007), we include the fixed-point operator within the universe, which simplifies code reuse. To allow for easy encapsulation of user-defined datatypes, we encode datatype isomorphisms in the universe itself.

The result is a universe that is extremely general. In fact, many earlier approaches such as `regular` (van Noort et al. 2008), PolyP (Jansson and Jeuring 1997), and Multirec (Rodriguez Yakushev et al. 2009) can be readily derived from our approach by parameter instantiation. Unlike Chapman et al. (2010), however, we do not strive to devise a minimal, self-encoding universe, but instead prefer a representation which maps naturally to the usual way of defining a datatype.

An added advantage of working in a dependently typed setting is that properties of generic functions are just generic functions themselves, and can be proved within the same fraemwork. We show how indexed functors adhere to the functor laws. As examples of how to work with the universe, we derive the catamorphism and other recursion schemes. Along with all the functionality that can be defined using the recursive morphisms, we show that our approach also allows defining functions by direct induction on the codes, and show decidable equality as an example. Finally, we present the basic ingredients for navigation of indexed functors using a zipper (Huet 1997), relying on an underlying derivative (McBride 2001) for the codes of our universe.

The rest of this paper is organized as follows: we start by describing our universe in Section 2 by demonstrating how to encode a few representative datatypes. In Section 3 we describe a zipper for indexed functors as an example of a type-indexed computation in our universe. Section 4 describes how to prove the functor laws for the map operation over indexed functors, and Section 5 shows the definition of decidable equality in our universe. Finally, we conclude in Section 6, pointing limitations of our approach and possible directions for future research.

We assume the reader is familiar with dependently-typed programming in Agda. Due to space constraints we do not include an Agda tutorial in this paper; the reader is referred to Norell (2009) for an introduction, also including a universe for generic programming.

We provide a full code bundle of our development in `http://dreixel.net/research/code/gpif.tar.gz`.

WGP'11, September 18, 2011, Tokyo, Japan.

2. A universe for indexed functors

If we want to define functions that depend on the structure of datatypes, we require a way of consistently representing user-definable datatypes by a small set of primitive types. Operations defined on the primitives can then be lifted to any other type, provided we have conversion functions between a user datatype and its generic representation.

As is common in a dependently-typed setting, we use a universe construction to encode the representation types: A *universe* is a type of *codes* together with an interpretation function that maps the codes to types.

The codes of our universe are supposed to describe indexed functors, and are therefore parameterized over two sets. Intuitively, these can be thought of as the indices for inputs and outputs of the type described by the code, so we generally call the first argument I and the second O. A code I ▶ O describes a type that is parameterized over inputs from I and that is itself indexed over O. In many standard examples, I and O will be instantiated to finite types. We get a classic functor of type Set → Set back by instantiating both I and O with the one-element type ⊤. And if we want a code to represent a family of n datatypes, we can instantiate O to the type Fin n that has n elements.

2.1 Basic codes

Since our universe is large, we present it in an incremental fashion, starting with the base cases:

```
data _▶_ : Set → Set → Set₁ where
  Z  : ∀ {I O} → I ▶ O
  U  : ∀ {I O} → I ▶ O
```

The constructor Z is used for empty types, and U for unit types. Both base codes are polymorphic in the input and output indices.

Disjoint sum, product and composition are used for combining codes:

```
_⊕_ : ∀ {I O} → I ▶ O → I ▶ O → I ▶ O
_⊗_ : ∀ {I O} → I ▶ O → I ▶ O → I ▶ O
_◎_ : ∀ {I M O} → M ▶ O → I ▶ M → I ▶ O
```

Sum (_⊕_) and product (_⊗_) combine two codes with input I and output O to produce an I ▶ O. For a composition F ◎ G, we require the codes to combine to be compatible: the output of code G needs to be the same as the input of the code F (namely M). We connect the output of G to the input of F, and produce a code with the input of G and the output of F.

As part of the universe we define an interpretation function, which establishes the relation between codes and Agda types. A code I ▶ O is interpreted as an *indexed functor* I ▷ O, which in turn is a function from a set indexed by I to a set indexed by O:

```
Indexed : Set → Set₁
Indexed I = I → Set

_▷_ : Set → Set → Set₁
I ▷ O = Indexed I → Indexed O
```

When defining the actual interpretation function ⟦_⟧, we thus have a parameter r : I → Set describing all the input sets and a parameter o : O selecting a particular output index available:

```
⟦_⟧ : ∀ {I O} → I ▶ O → I ▷ O
⟦ Z     ⟧ r o = ⊥
⟦ U     ⟧ r o = ⊤
⟦ F ⊕ G ⟧ r o = ⟦ F ⟧ r o + ⟦ G ⟧ r o
⟦ F ⊗ G ⟧ r o = ⟦ F ⟧ r o × ⟦ G ⟧ r o
⟦ F ◎ G ⟧ r o = (⟦ F ⟧ ∘ ⟦ G ⟧) r o
```

We interpret Z as the empty type ⊥, and U as the singleton type ⊤ with constructor tt.

Each of the three constructors for sum, product and composition is interpreted as the corresponding concept on (lifted) Agda types. For sums, we use the _+_ type with constructors inl and inr. For products, we use the _×_ type with constructor _,_. For composition, we use ordinary function composition _∘_ on the interpretations.

With this part of the universe in place we can already encode some simple, non-recursive datatypes. Consider the type of boolean values Bool with constructors true and false:

```
data Bool : Set where
  true  : Bool
  false : Bool
```

It is a single, non-indexed datatype, which takes no parameters and is not recursive. We can encode it as an indexed functor using a type with zero inhabitants for I and a type with one inhabitant for O:

```
'BoolF' : ⊥ ▶ ⊤
'BoolF' = U ⊕ U
```

To convert between the type of booleans and its representation, we use two functions:

```
fromBool : ∀ {r o} → Bool → ⟦ 'BoolF' ⟧ r o
fromBool true  = inl tt
fromBool false = inr tt

toBool : ∀ {r o} → ⟦ 'BoolF' ⟧ r o → Bool
toBool (inl tt) = true
toBool (inr tt) = false
```

2.2 Isomorphisms

Being in a dependently-typed language, we can also provide the proof that fromBool and toBool indeed form an isomorphism:

```
isoBool₁ : ∀ {b r o} → toBool {r} {o} (fromBool b) ≡ b
isoBool₁ {true}  = refl
isoBool₁ {false} = refl

isoBool₂ : ∀ {b r o} → fromBool {r} {o} (toBool b) ≡ b
isoBool₂ {inl tt} = refl
isoBool₂ {inr tt} = refl
```

For convenience, we wrap all the above in a single record, which we reuse later:

```
infix 3 _≃_
record _≃_ (A B : Set) : Set where
  field
    from : A → B
    to   : B → A
    iso₁ : ∀ {x} → to (from x) ≡ x
    iso₂ : ∀ {x} → from (to x) ≡ x
```

The particular instantiation for Bool simply uses the functions we have defined before:

```
isoBool : (r : Indexed ⊥) (o : ⊤) → Bool ≃ ⟦ 'BoolF' ⟧ r o
isoBool r o = record { from = fromBool {r} {o}
                     ; to   = toBool   {r} {o}
                     ; iso₁ = isoBool₁
                     ; iso₂ = isoBool₂ }
```

Isomorphisms are useful when we want to view a type as a different, but equivalent type. We can use isomorphisms to extend our universe: we may not be able to directly encode a type as an indexed functor, but it is sufficient if we can encode an isomorphic type:

```
Iso : ∀ {I O} → (C : I ▶ O) → (D : I ▷ O)
    → ((r : Indexed I) → (o : O) → D r o ≃ ⟦ C ⟧ r o) → I ▶ O
```

The code Iso captures an isomorphism between the interpretation of a code C and an indexed functor D. This indexed functor D is used in the interpretation of the code Iso:

```
⟦ Iso C D e ⟧ r o = D r o
```

Having Iso has two advantages. Firstly, it allows us to have actual user-defined Agda datatypes in the image of our interpretation function. For example, we can give a code for the Agda datatype Bool now:

```
'Bool' : ⊥ ▶ ⊤
'Bool' = Iso 'BoolF' (λ _ _ → Bool) isoBool
```

Secondly, Iso makes it possible to reuse types that we previously defined inside more complex definitions, while preserving the original code for further generic operations—in Section 2.7 we show an example of this technique by reusing the code for natural numbers inside the definition of a simple language.

2.3 Adding fixed points

We cannot yet describe recursive datatypes in our universe of indexed functors. The standard way to treat recursive datatypes generically is to add a fixed-point operator that computes the fixed point of a type described by a code.

Indexed functors, however, have the nice property of being closed under fixed points: if we have an indexed functor where the recursive calls come from the same index set as the output, i.e. a functor O ▷ O, then the fixed point will be of type ⊥ ▷ O. If we consider a functor with parameters of type I + O ▷ O, with input indices being either parameters from I, or recursive calls from O, we can obtain a fixed point of type I ▷ O.

Since the fixed point of indexed functors is another indexed functor, it is possible and convenient to just add fixed points as another constructor to our type of codes:

```
Fix : ∀ {I O} → (I + O) ▶ O → I ▶ O
```

As indicated above, the code Fix transforms a code with I + O input indices and O output indices into a code of type I ▶ O.

Of course, we also need a way to actually access inputs:

```
I    : ∀ {I O} → I          → I ▶ O
```

Using I, we can select a particular input index (which, in the light of Fix, might be either a parameter or a recursive call).

We use a datatype μ with a single constructor ⟨_⟩ to generate the interpretation of Fix F from that of F. We know that F : (I + O) ▶ O, so the first argument to ⟦ F ⟧ needs to discriminate between parameters and recursive occurrences. We use r | μ F r for this purpose, i.e. for parameters we use r : I → Set, whereas recursive occurrences are interpreted with μ F r : O → Set:

```
_|_ : ∀ {I J} → Indexed I → Indexed J → Indexed (I + J)
(r | s) (inl i) = r i
(r | s) (inr j) = s j

mutual
  data μ {I O : Set} (F : (I + O) ▶ O)
         (r : Indexed I) (o : O) : Set where
    ⟨_⟩ : ⟦ F ⟧ (r | μ F r) o → μ F r o

  ...
⟦ Fix F ⟧ r o = μ F r o
⟦ I i   ⟧ r o = r i
```

The interpretation of I uniformly invokes r for every input index i.

As an example of a datatype with parameters and recursion, we show how to encode parametric lists. We start by encoding the base functor of lists:

```
infixr 6 _::_
data [_] (A : Set) : Set where
  []  : [A]
  _::_ : A → [A] → [A]
'ListF' : (⊤ + ⊤) ▶ ⊤
'ListF' = U ⊕ (I (inl tt) ⊗ I (inr tt))
```

The arguments of _▶_ reflect that we are defining one type (output index ⊤) with two inputs (input index ⊤ + ⊤), where one is a parameter and one is representing the recursive call.

We use a product to encode the arguments to the _::_ constructor. The first argument is an occurrence of the first (and only) parameter, and the second is a recursive occurrence of the first (and only) type being defined (namely list).

Using Fix, we can now close the recursive gap in the representation of lists:

```
'List' : ⊤ ▶ ⊤
'List' = Fix 'ListF'
```

We can confirm that our representation is isomorphic to the original type by providing conversion functions:

```
fromList : ∀ {r o} → [r o] → ⟦ 'List' ⟧ r o
fromList []        = ⟨ inl tt ⟩
fromList {o = tt} (x :: xs) = ⟨ inr (x , fromList xs) ⟩

toList : ∀ {r o} → ⟦ 'List' ⟧ r o → [r o]
toList ⟨ inl tt ⟩       = []
toList {o = tt} ⟨ inr (x , xs) ⟩ = x :: toList xs
```

As before, we can show that the conversion functions really form an isomorphism—this looks a bit technical, because we have to explicitly pass some implicit arguments in order to satisfy Agda's typechecker, but the isomorphism is entirely trivial:

```
cong≡ : {A : Set} {B : Set} {x y : A} (f : A → B) →
        x ≡ y → f x ≡ f y
cong≡ f refl = refl

isoList₁ : {r : Indexed ⊤} {o : ⊤} {l : [r o]} →
           toList {r} {o} (fromList l) ≡ l
isoList₁ {r} {tt} {h :: t} =
  cong≡ ((λ x → h :: x)) (isoList₁ {r} {tt} {t})
isoList₁ {r} {tt} {[]}     = refl

isoList₂ : {r : Indexed ⊤} {o : ⊤} {l : ⟦ 'List' ⟧ r o} →
           fromList (toList l) ≡ l
isoList₂ {r} {tt} {⟨ inl tt ⟩}     = refl
isoList₂ {r} {tt} {⟨ inr (h , t) ⟩} =
  cong≡ ((λ x → ⟨ inr (h , x) ⟩)) (isoList₂ {r} {tt} {t})

isoList : {r : Indexed ⊤} {o : ⊤} → [r o] ≃ ⟦ 'List' ⟧ r o
isoList {r} = record {from = fromList
                      ; to   = toList
                      ; iso₁ = isoList₁ {r}; iso₂ = isoList₂}
```

We will refrain from showing further proofs of isomorphisms in this paper.

We now redefine 'List' to include the isomorphism between List and Fix 'ListF':

```
'List' : ⊤ ▶ ⊤
'List' = Iso (Fix 'ListF') (λ f t → [f t]) (λ r o → isoList)
```

This code makes it possible for us to use actual Agda lists in generic operations, and explicit applications of the conversion functions toList and fromList are no longer necessary.

Having Fix in the universe (as opposed to using it externally) has the advantage that codes become more reusable. For example, we can reuse the code for lists we have just defined while defining a code for rose trees (Section 2.6), which are the instance of the common situation where a fixed point is used as the first argument of a composition. We thus can represent datatypes in our universe that involve several applications of Fix.

2.4 Mapping for indexed functors

We make the claim that the interpretation of our codes are indexed functors, but we have yet to show that we can define a map operation for them.

We are working with indexed sets rather than sets, so we have to look at arrows between indexed sets, which are index-preserving functions:

```
_⇉_ : ∀ {I} → Indexed I → Indexed I → Set
r ⇉ s = ∀ i → r i → s i
```

As usual, map lifts such an indexed-preserving function r ⇉ s between two indexed sets r and s to an indexed-preserving function ⟦ C ⟧ r ⇉ ⟦ C ⟧ s on the interpretation of a code C:

```
map : {I O : Set} {r s : Indexed I} →
      (C : I ▶ O) → (r ⇉ s) → (⟦ C ⟧ r ⇉ ⟦ C ⟧ s)
```

Note that by choosing I = O = ⊤, the indexed sets become isomorphic to sets, and the index-preserving function become isomorphic to functions between two sets, i.e. we specialize to the well-known Haskell-like setting of functors of type Set → Set.

Let us look at the implementation of map:

```
map Z       f o ()
map U       f o tt  = tt
map (I i) f o x   = f i x
```

If we expand the type of map, we see that it takes four explicit arguments: The first two are the code and the indexed-preserving function f. The function returned by map can be expanded to ∀ o → ⟦ C ⟧ r o → ⟦ C ⟧ s o, explaining the remaining two arguments: an arbitrary output index o, and an element of the interpreted code.

The interpretation of code Z has no inhabitants, so we do not have to give an implementation. For U, we receive a value of type ⊤, which must be tt, and in particular does not contain any elements, so we return tt unchanged. On I we get an element x corresponding to input index i, which we supply to the function f at index i.

For sums and products, we keep the structure, pushing the map inside. Composition is handled with a nested map, and for fixed points we adapt the function argument to take into account the two different types of indices: using an auxiliary _‖_ operator to merge natural transformations, left-tagged indices (parameters) are mapped with f, whereas right-tagged indices (recursive occurrences) are mapped recursively:

```
map (F ⊕ G) f o (inl x)  = inl (map F f o x)
map (F ⊕ G) f o (inr y)  = inr (map G f o y)
map (F ⊗ G) f o (x , y)  = map F f o x , map G f o y
map (F ⊙ G) f o x        = map F (map G f) o x
map (Fix F)   f o ⟨ x ⟩  = ⟨ map F (f ‖ map (Fix F) f) o x ⟩
```

Finally, mapping over isomorphisms requires judicious use of the conversion functions:

```
map {r = r} {s = s} (Iso C D e) f o x with (e r o , e s o)
... | ep₁ , ep₂ = _≃_.to ep₂ (map C f o (_≃_.from ep₁ x))
```

As an example, let us look at the specific instance of map on lists. We can obtain that simply by specializing the types:

```
↑ : ∀ {A B : Set} → (A → B) → (const A ⇉ const B)
↑ f i x = f x

mapList : ∀ {A B} → (A → B) → [A] → [B]
mapList f = map 'List' (↑ f) tt
```

We are in the situation described above, where both index sets are instantiated to ⊤. The lifting operator ↑ witnesses one half of the isomorphism between A → B and const A ⇉ const B. Note that we do not need to apply any conversion functions, since 'List' contains the isomorphism between lists and their representation. We discuss how to prove the functor laws for map in Section 4.

For now, we can confirm that mapList works as expected on a simple example:

```
mapListExample : mapList suc (1 :: 2 :: []) ≡ (2 :: 3 :: [])
mapListExample = refl
```

In this example, we use the standard type for naturals ℕ with constructors zero and suc, and the propositional equality type _≡_ with constructor refl.

2.5 Recursion schemes

Equipped with a mapping function for indexed functors, we can define basic recursive morphisms in a conventional fashion:

```
id⇉ : {I : Set} {r : Indexed I} → r ⇉ r
id⇉ i = id

cata : {I O : Set} {r : Indexed I} {s : Indexed O} →
       (C : (I + O) ▶ O) →
       (⟦ C ⟧ (r ‖ s) ⇉ s) → ⟦ Fix C ⟧ r ⇉ s
cata C φ o ⟨ x ⟩ = φ o (map C (id⇉ ‖ cata C φ) o x)
```

The catamorphism on indexed functors is not much different from the standard functorial catamorphism. The important difference is the handling of left and right indices differently: since we wish to traverse over the structure only, we apply the identity on indexed sets id⇉ to parameters, and recursively apply cata to right-tagged indices.

As an example, let us consider lists again and see how the foldr function can be expressed in terms of the generic catamorphism:

```
_▽_ : {I J R : Set} → (I → R) → (J → R) → (I + J) → R
(r ▽ s) (inl i) = r i
(r ▽ s) (inr j) = s j

up : {F : Indexed ⊤} → F tt → (i : ⊤) → F i
up x tt = x

T : Set → Indexed ⊤
T A tt = A

foldr : {A R : Set} → (A → R → R) → R → [A] → R
foldr {A} {R} c n xs =
  cata {r = T A} {s = T R} 'ListF' φ tt (fromList xs)
    where φ = up (const n ▽ uncurry c)
```

The function foldr invokes cata. The parameters are instantiated with T A, i.e. there is a single parameter and it is A, and the recursive slots are instantiated with T R, i.e. there is a single recursive slot and it will be transformed into an R. We invoke the catamorphism on 'ListF', which means we have to manually apply the coercion fromList on the final argument to get from the user-defined list type to the isomorphic structural representation. Ultimately we are interested in the single output index tt that lists provide.

This leaves the algebra φ. The generic catamorphism expects something of type ⟦'ListF' ⟧ (r ‖ s) ⇉ s, which can be reduced in this context to (i : ⊤) → ⊤ + A × R → R i. On the other hand, foldr takes the nil- and cons- component separately, which we can join together with _▽_ to obtain something of type ⊤ + A × R → R. It turns out that up provides the desired generalization of the type.

We can define the length of a list using foldr and check that it works as expected:

```
length : ∀ {A} → [A] → ℕ
length = foldr (const suc) zero

lengthExample : length (1 :: 0 :: []) ≡ 2
lengthExample = refl
```

Many other recursive patterns can be defined similarly. We show the definitions of ana and hylo:

```
ana : {I O : Set} {r : Indexed I} {s : Indexed O} →
      (C : (I + O) ▶ O) →
      (s ⇉ ⟦ C ⟧ (r ∣ s)) → s ⇉ ⟦ Fix C ⟧ r
ana C ψ    o x = ⟨ map C (id⇉ ∥ ana C ψ) o (ψ o x) ⟩

hylo : {I O : Set} {r : Indexed I} {s t : Indexed O} →
       (C : (I + O) ▶ O) →
       (⟦ C ⟧ (r ∣ t) ⇉ t) → (s ⇉ ⟦ C ⟧ (r ∣ s)) → s ⇉ t
hylo C φ ψ o x = φ o (map C (id⇉ ∥ hylo C φ ψ) o (ψ o x))
```

In our code bundle we also provide para- and apomorphisms.

2.6 Using composition

To show how to use composition we encode the type of rose trees:

```
data Rose (A : Set) : Set where
  fork : A → [Rose A] → Rose A
```

The second argument to fork, of type [Rose A], is encoded using composition:

```
'RoseF' : (⊤ + ⊤) ▶ ⊤
'RoseF' = I (inl tt) ⊗ ('List' ⊙ I (inr tt))

'Rose' : ⊤ ▶ ⊤
'Rose' = Fix 'RoseF'
```

Note that the first argument of composition here is 'List', not 'ListF'. We thus really make use of the fact that fixed points are part of our universe and can appear everywhere within a code. We also show that codes can be reused in the definitions of other codes.

The conversion functions for rose trees are:

```
fromRose : {r : Indexed ⊤} {o : ⊤} → Rose (r o) →
           ⟦ 'Rose' ⟧ r o
fromRose {o = tt} (fork x xs) =
  ⟨ x , map 'List' (λ i → fromRose) tt (fromList xs) ⟩

toRose : {r : Indexed ⊤} {o : ⊤} → ⟦ 'Rose' ⟧ r o →
         Rose (r o)
toRose {o = tt} ⟨ x , xs ⟩ =
  fork x (toList (map 'List' (λ i → toRose) tt xs))
```

The use of composition in the code implies the use of map in the conversion functions, since we have to map the conversion over the elements of the list. This also means that to provide the isomorphism proofs for Rose we need to have proofs for the behavior of map. We describe these in detail in Section 4.

2.7 Parametrized families of datatypes

To explore the full power of abstraction of indexed functors, we show an example with a family of mutually recursive parametrized datatypes. Our family represents the Abstract Syntax Tree (AST) of a simple language:

```
mutual
  data Expr (A : Set) : Set where
    econst : ℕ                → Expr A
    add    : Expr A → Expr A → Expr A
    evar   : A                → Expr A
    elet   : Decl A → Expr A → Expr A
  data Decl (A : Set) : Set where
    assign : A          → Expr A → Decl A
    seq    : Decl A → Decl A → Decl A
```

In our AST, an expression can be a natural number constant, the addition of two expressions, a variable, or a let declaration. Declarations are either an assignment of an expression to a variable, or a sequence of declarations.

We can easily encode each of the datatypes as indexed functors. We start by defining a type synonym for the output indices, for convenience:

```
AST : Set
AST = ⊤ + ⊤

expr : AST
expr = inl tt

decl : AST
decl = inr tt
```

Since we are defining two datatypes, we use a type with two inhabitants, namely ⊤ + ⊤. Note that any other two-element type such as Bool or Fin 2 would also do. We define expr and decl as shorthands for each of the indices. We can now encode each of the types:

```
'ExprF' : (⊤ + AST) ▶ AST
'ExprF' =   ?₀
          ⊕ I (inr expr) ⊗ I (inr expr)
          ⊕ I (inl tt)
          ⊕ I (inr decl) ⊗ I (inr expr)

'DeclF' : (⊤ + AST) ▶ AST
'DeclF' =   I (inl tt)    ⊗ I (inr expr)
          ⊕ I (inr decl) ⊗ I (inr decl)
```

Our codes have type (⊤ + AST) ▶ AST, since we have one parameter (the type of the variables) and two datatypes in the family. For expressions, we want to reuse the 'ℕ' code for ℕ we have defined previously. However, 'ℕ' has type ⊥ ▶ ⊤, which is not compatible with the current code, so we cannot simply enter 'ℕ' in the ?₀ hole.

We need some form of re-indexing operation to plug indexed functors within each other. Therefore we add the following operation to our universe:

```
_↗_↘_ : ∀ {I I' O O'}          → I' ▶ O' →
        (I' → I) → (O → O') → I ▶ O
```

Now we can fill the hole ?₀ with the expression 'ℕ' ↗ (λ ()) ↘ const tt. The interpretation of this new code is relatively simple. For the input, we compose the re-indexing function with r, and for the output we apply the function to the output index:

```
⟦ F ↗ f ↘ g ⟧ r o = ⟦ F ⟧ (r ∘ f) (g o)
```

Mapping over a re-indexed code is also straightforward:

```
map (F ↗ g ↘ h) f o x = map F (f ∘ g) (h o) x
```

Finally, we can now join the two codes for expressions and declarations into a single code for the whole family. For this we will need an additional code to specify that we are defining one particular output index:

```
! : ∀ {I O} → O → I ▶ O
```

The code ! is parameterized by a particular output index. Its interpretation introduces the constraint that the argument index should be the same as the output index we select when interpreting:

```
⟦ ! o' ⟧ r o = o ≡ o'
```

Its usefulness becomes clear when combining the codes for the AST family:

```
'ASTF' : (⊤ + AST) ▶ AST
'ASTF' =   ! expr ⊗ 'ExprF'
         ⊕ ! decl ⊗ 'DeclF'

'AST' : ⊤ ▶ AST
'AST' = Fix 'ASTF'
```

In 'ASTF' (the code for the family before closing the fixed point), we encode either an expression or a declaration, each coupled with an equality proof that forces the output index to match the datatype we are defining. If we now select the expr index from the interpretation of 'ASTF', then ! decl yields an uninhabited type, whereas ! expr yields a trivial equality, thereby ensuring that only 'ExprF' corresponds to the structure in this case. If we select decl in turn, then only 'DeclF' contributes to the structure.

We can now define the conversion functions between the original datatypes and the representation:

```
mutual
  toExpr : {r : ⊤ → Set} → ⟦ 'AST' ⟧ r expr → Expr (r tt)
  toExpr ⟨ inl (refl , inl x) ⟩                    = econst x
  toExpr ⟨ inl (refl , inr (inl (x , y))) ⟩        = add (toExpr x)
                                                         (toExpr y)
  toExpr ⟨ inl (refl , inr (inr (inl x))) ⟩        = evar x
  toExpr ⟨ inl (refl , inr (inr (inr (d , e)))) ⟩  = elet (toDecl d)
                                                         (toExpr e)
  toExpr ⟨ inr (() , _) ⟩
  toDecl : {r : ⊤ → Set} → ⟦ 'AST' ⟧ r decl → Decl (r tt)
  toDecl ⟨ inl (() , _) ⟩
  toDecl ⟨ inr (refl , inl (x , e)) ⟩     = assign x (toExpr e)
  toDecl ⟨ inr (refl , inr (d₁ , d₂)) ⟩   = seq (toDecl d₁)
                                                (toDecl d₂)
```

The important difference from the previous examples is that now we have absurd patterns. For instance, in toExpr we have to produce an Expr, so the generic value starting with inr is impossible, since there is no inhabitant of the type expr ≡ decl. Dually, in the conversion from the original type into the generic type, these proofs have to be supplied:

```
mutual
  fromExpr : {r : ⊤ → Set} → Expr (r tt) →
             ⟦ 'AST' ⟧ r expr
  fromExpr (econst x) = ⟨ inl (refl , inl x) ⟩
  fromExpr (add x y)  =
    ⟨ inl (refl , inr (inl (fromExpr x , fromExpr y))) ⟩
  fromExpr (evar x)   = ⟨ inl (refl , inr (inr (inl x))) ⟩
  fromExpr (elet d e) =
    ⟨ inl (refl , inr (inr (inr (fromDecl d , fromExpr e)))) ⟩
  fromDecl : {r : ⊤ → Set} → Decl (r tt) →
             ⟦ 'AST' ⟧ r decl
  fromDecl (assign x e) = ⟨ inr (refl , inl (x , fromExpr e)) ⟩
  fromDecl (seq d₁ d₂)  =
    ⟨ inr (refl , inr (fromDecl d₁ , fromDecl d₂)) ⟩
```

At this stage the proofs are trivial to produce (refl), since we know exactly the type of the index.

2.8 Arbitrarily indexed datatypes

The index types of a functor need not be finite types. Consider the following datatype describing lists of a fixed length (vectors):

```
infixr 5 _::_
data Vec (A : Set) : ℕ → Set where
  []  :                          Vec A zero
  _::_ : {n : ℕ} → A → Vec A n → Vec A (suc n)
```

The type Vec is indexed by the type of natural numbers ℕ. In fact, for a given type A, Vec A defines a family of sets: Vec A zero (which contains only the empty list), Vec A (suc zero) (all possible singleton lists), and so on. As such, we can see Vec as a code with one input parameter (the type A) and ℕ output parameters:

```
VecF : ℕ → (⊤ + ℕ) ▶ ℕ
VecF zero    = ! zero
VecF (suc n) = ! (suc n) ⊗ I (inl tt) ⊗ I (inr n)
```

Note, however, that we need to parameterize VecF by a natural number, since the code depends on the particular value of the index. In particular, a vector of length suc n is an element together with a vector of length n. Other than in the case for the abstract syntax trees, we cannot simply sum up the codes for all the different choices of output index, because there are infinitely many of them. Therefore, we need yet another code in our universe:

```
Σ : ∀ {I O} {C : ⊥ ▶ ⊤} →
    (⟦ C ⟧ (λ _ → ⊤) tt → I ▶ O) → I ▶ O
```

The code Σ introduces an existential datatype:

```
data ∃ {A : Set} (B : A → Set) : Set where
  some : ∀ {x} → B x → ∃ B

⟦ Σ f ⟧ r o = ∃ (λ i → ⟦ f i ⟧ r o)
```

Note that Σ is parameterized by a function f that takes values to codes. Arguments of f are supposed to be indices, but we would like them to be described by codes again, since that makes it easier to define generic functions over the universe. Therefore, we make a compromise and choose a code for a single unparameterized datatype ⊥ ▶ ⊤ rather than an arbitrary Set—for more discussion, see Section 2.10.

To create a value of type ⟦ Σ f ⟧ we need a specific witness i to obtain a code from f. When using a value of type ⟦ Σ f ⟧, we can access the index stored in the existential.

Here is the map function for Σ:

```
map (Σ g) f o (some {i} x) = some (map (g i) f o x)
```

Using Σ, we can finalize the encoding of Vec:

```
'Vec' : ⊤ ▶ ℕ
'Vec' = Fix (Σ {C = 'ℕ'} VecF)
```

We make use of the fact that we already have a code 'ℕ' for our index type of natural numbers.

Finally, we can provide the conversion functions. We need to pattern-match on the implicit natural number to be able to provide it as existential evidence in fromVec', and to be able to produce the right constructor in toVec':

```
fromVec : ∀ {n r} → Vec (r tt) n → ⟦ 'Vec' ⟧ r n
fromVec {n = zero}  []       = ⟨ some {x = zero} refl ⟩
fromVec {n = suc m} (h :: t) = ⟨ some {x = suc m}
                                      (refl , (h , fromVec t)) ⟩

toVec : ∀ {n r} → ⟦ 'Vec' ⟧ r n → Vec (r tt) n
toVec ⟨ some {zero}  refl ⟩          = []
toVec ⟨ some {suc n} (refl , (h , t)) ⟩ = h :: toVec t
```

2.9 Nested datatypes

Nested datatypes (Bird and Meertens 1998) can be encoded in Agda using indexed datatypes. Consider the type of perfect binary trees:

```
data Perfect (A : Set) : {n : ℕ} → Set where
  split : {n : ℕ} → Perfect A {n} × Perfect A {n} →
```

```
                Perfect A {suc n}
  leaf  : A → Perfect A {zero}
```

Perfect trees are indexed over the naturals. A perfect tree is either a leaf, which has depth zero, or a split-node, which has depth suc n and contains two subtrees of depth n each.

In Haskell, this type is typically encoded by changing the parameters of the type in the return type of the constructors: split would have return type Perfect (Pair A), for some suitable Pair type. We can define Perfect' as such a nested datatype in Agda, too:

```
data Pair (A : Set) : Set where
  pair : A → A → Pair A
data Perfect' : (A : Set) → Set₁ where
  split : (A : Set) → Perfect' (Pair A)
  leaf  : {A : Set} → A → Perfect' A
```

Now, Perfect' is isomorphic to a dependent pair of a natural number n and an element of Perfect {n}.

We can therefore reduce the problem of encoding Perfect' to the problem of encoding Perfect, which in turn can be done very similar as the encoding of vectors show in Section 2.8:

```
PerfectF : ℕ → (⊤ + ℕ) ▶ ℕ
PerfectF (zero)  = ! zero     ⊗ I (inl tt)
PerfectF (suc n) = ! (suc n) ⊗ I (inr n) ⊗ I (inr n)
'PerfectF' : (⊤ + ℕ) ▶ ℕ
'PerfectF' = Σ {C = 'ℕ'} PerfectF
'Perfect' : ⊤ ▶ ℕ
'Perfect' = Fix 'PerfectF'
```

We omit the embedding-projection pair as it provides no new insights.

2.10 Summary and discussion

At this point, we are at the end of discussing how various sorts of datatypes can be encoded in our universe, and we have presented all the constructors we need for our type of codes. For reference, we show the backbone of our approach in Figure 1: the universe, its interpretation, and the map function.

Naturally, there are several variations possible of our approach, and our universe is not the only useful spot in this part of the design space. We will now briefly discuss a number of choices we have made.

Perhaps most notably, we have avoided the inclusion of arbitrary constants of the form

```
K : ∀ {I O} → Set → I ▶ O
```

There are two main reasons why one might want to have constants in universes. One is to be able to refer to user-defined datatypes. We can do this via Iso, as long as the user-defined datatypes can be isomorphically represented by a code. The other reason is to be able to include abstract base types (say, floating point numbers), for which it is difficult to give a structural representation.

Adding constants, however, introduces problems as well. While map is trivial to define for constants—they are just ignored—most other functions, such as e.g. decidable equality, become impossible to define in the presence of arbitrary constants. Additional assumptions (such as that the constants being use admit decidable equality themselves) must usually be made, and it is impossible to predict in advance all the constraints necessary when defining a K code.

A similar problem guides our rather pragmatic choice when defining Σ. There are at least two other potential definitions for Σ:

```
Σ₁ : ∀ {I O} → Set → I ▶ O
```

```
data _▶_ : Set → Set → Set₁ where
  Z   : ∀ {I O} → I ▶ O
  U   : ∀ {I O} → I ▶ O
  I   : ∀ {I O} → I   → I ▶ O
  !   : ∀ {I O} → O → I ▶ O
  _⊕_ : ∀ {I O}   → I   ▶ O → I ▶ O → I ▶ O
  _⊗_ : ∀ {I O}   → I   ▶ O → I ▶ O → I ▶ O
  _◎_ : ∀ {I M O} → M ▶ O → I ▶ M → I ▶ O
  _↗_↘_ : ∀ {I I' O O'} → I' ▶ O' →
          (I' → I) → (O → O') → I ▶ O
  Fix : ∀ {I O} → (I + O) ▶ O → I ▶ O
  Σ   : ∀ {I O} → {C : ⊥ ▶ ⊤} →
          (⟦ C ⟧ (λ _ → ⊤) tt → I ▶ O) → I ▶ O
  Iso : ∀ {I O} → (C : I ▶ O) → (D : I ▷ O) →
          ((r : Indexed I) → (o : O) → D r o ≃ ⟦ C ⟧ r o) →
          I ▶ O

data μ {I O : Set} (F : (I + O) ▶ O)
       (r : Indexed I) (o : O) : Set where
  ⟨_⟩ : ⟦ F ⟧ (r | μ F r) o → μ F r o
⟦_⟧ : ∀ {I O} → I ▶ O → I ▷ O
⟦ Z       ⟧ r o = ⊥
⟦ U       ⟧ r o = ⊤
⟦ I i     ⟧ r o = r i
⟦ F ↗ f ↘ g ⟧ r o = ⟦ F ⟧ (r ∘ f) (g o)
⟦ F ⊕ G   ⟧ r o = ⟦ F ⟧ r o + ⟦ G ⟧ r o
⟦ F ⊗ G   ⟧ r o = ⟦ F ⟧ r o × ⟦ G ⟧ r o
⟦ F ◎ G   ⟧ r o = ⟦ F ⟧ (⟦ G ⟧ r) o
⟦ Fix F   ⟧ r o = μ F r o
⟦ ! o'    ⟧ r o = o ≡ o'
⟦ Σ f     ⟧ r o = ∃ (λ i → ⟦ f i ⟧ r o)
⟦ Iso C D e ⟧ r o = D r o

map : {I O : Set} {r s : Indexed I} (C : I ▶ O) →
      (r ⇉ s) → (⟦ C ⟧ r ⇉ ⟦ C ⟧ s)
map Z       f o ()
map U       f o x                = x
map (I i)   f o x                = f i x
map (F ⊕ G) f o (inl x)          = inl (map F f o x)
map (F ⊕ G) f o (inr y)          = inr (map G f o y)
map (F ⊗ G) f o (x , y)          = map F f o x , map G f o y
map (F ↗ g ↘ h) f o x            = map F (f ∘ g) (h o) x
map (F ◎ G) f o x                = map F (map G f) o x
map (! o')  f o x                = x
map (Σ g)   f o (some {i} x)     = some (map (g i) f o x)
map (Fix F) f o ⟨ x ⟩ = ⟨ map F (f ‖ map (Fix F) f) o x ⟩
map {r = r} {s = s} (Iso C D e) f o x with (e r o , e s o)
... | (ep₁ , ep₂) = _≃_.to ep₂ (map C f o (_≃_.from ep₁ x))
```

Figure 1. The universe, its interpretation, and mapping

```
Σ₂ : ∀ {I O I' O' s o'} {C : I' ▶ O'} →
     (⟦ C ⟧ s o' → I' ▶ O') → I ▶ O
```

In the first case, we allow an arbitrary Set as index type. This, however, leads to problems with decidable equality (Morris 2007, Section 3.3), because in order to compare two existential pairs for

equality we have to compare the indices. Restricting the indices to representable types guarantees we can easily compare them. The second variant is more general than our current Σ, abstracting from a code with any input and output indices. However, this makes the interpretration depend on additional parameters s and o', which we are unable to produce, in general. In our Σ we avoid this problem by setting I' to ⊥ and O' to ⊤, so that s is trivially λ () and o' is tt. Our Σ encodes indexing over a single unparametrised datatype.

In general, many constructs in our universe (such as reindexing) could be slightly generalized or more restricted, always balancing the ease of representing certain datatypes against the ease of defining certain datatype-generic functions. We believe that our universe serves rather well in practice.

Furthermore, it is interesting to note that several other libraries for datatype-generic programming with fixed points can be obtained from ours by instantiating the input and output indices appropriately. The `regular` library of Van Noort et al. (2008) considers functors defining single datatypes without parameters. This corresponds to instantiating the input index to ⊥ + ⊤ (no parameters, one recursive slot) and the output index to ⊤, and allowing fixed points only on the outside, of type ⊥ + ⊤ ▶ ⊤ → ⊥ ▶ ⊤.

Adding one parameter brings us to the realm of PolyP (Jansson and Jeuring 1997). PolyP works with fixed points of bifunctors, and the kind of the fixed-point operator used in PolyP corresponds exactly to the type ⊤ + ⊤ ▶ ⊤ → ⊤ ▶ ⊤, i.e. taking a bifunctor to a functor. Note also that PolyP furthermore allows a limited form of composition where the left operand is a fixed point of a bifunctor (i.e. of type ⊤ ▶ ⊤), and the right operand is a bifunctor (of type ⊤ + ⊤ ▶ ⊤).

Multirec (Rodriguez Yakushev et al. 2009) supports any finite number n of mutually recursive datatypes without parameters; that corresponds to fixed points of type ⊥ + Fin n ▶ Fin n → ⊥ ▶ Fin n.

Our library thus generalizes all of the above libraries. It allows arbitrarily many mutually-recursive datatypes with arbitrarily many parameters, and it allows non-finite index types. In the remainder of the paper we provide further evidence of the usefulness of the universe presented by showing a number of applications.

3. A zipper for indexed functors

Along with defining generic functions in our universe, we can also define type-indexed datatypes (Hinze et al. 2002): types defined generically in the universe of representations. A frequent example is the type of one-hole contexts, described for regular types by McBride (2001), and for mutually recursive datatypes by Rodriguez Yakushev et al. (2009), with application to the zipper (Huet 1997).

Here, we revisit the zipper in the context of our universe, starting with the type-indexed datatype of one-hole contexts, and proceeding to the navigation functions.

3.1 Generic contexts

The one-hole context of an indexed functor is the type of values where exactly one input position is replaced by a hole. The idea is that we can then split an indexed functor into an input value and its context. Later, we can identify a position in a datatype by keeping a stack of one-hole contexts that give us a path from the subtree in focus up to the root of the entire structure.

We define Ctx as another interpretation function for our universe: it takes a code and the input index indicating what kind of position we want to replace by a hole, and it returns an indexed functor again:

```
Ctx : {I O : Set} → I ▶ O → I → I ▷ O
Ctx Z       i r o = ⊥
Ctx U       i r o = ⊥
Ctx (! o')  i r o = ⊥
Ctx (I i')  i r o = i ≡ i'
```

For the void, unit, and tag types there are no possible holes, so the context is the empty datatype. For I, we have a hole if and only if the index for the hole matches the index we recurse on. If there is a hole, we want the context to be isomorphic to the unit type, otherwise it should be isomorphic to the empty type. An equality type of i and i' has the desired property.

For a re-indexed code we store proofs of the existence of the new indices together with the reindexed context:

```
Ctx (F ↗ f ↘ g) i r o = ∃ (λ i' → ∃ (λ o' →
                          f i' ≡ i × g o ≡ o' × Ctx F i' (r ∘ f) o'))
```

As described by McBride (2001), computing the one-hole context of a polynomial functor corresponds to computing the formal derivative. This correspondence motivates the definition for sum, product and composition. Notably, the context for a composition follows the chain rule:

```
Ctx (F ⊕ G) i r o = Ctx F i r o + Ctx G i r o
Ctx (F ⊗ G) i r o = Ctx F i r o × ⟦ G ⟧ r o + ⟦ F ⟧ r o × Ctx G i r o
Ctx (F ⊙ G) i r o = ∃ (λ m → Ctx F m (⟦ G ⟧ r) o × Ctx G i r m)
```

The context of a Σ f is the context of the resulting code f i', for some appropriate index i'. The context of an Iso is the context of the inner code:

```
Ctx (Σ f)       i r o = ∃ (λ i' → Ctx (f i') i r o)
Ctx (Iso C D e) i r o = Ctx C i r o
```

The context of a fixed point is more intricate. Previous zippers (such as that of the Multirec library) have not directly considered the context for a fixed point, since these were outside the universe. If we are interested in positions of an index i within a structure that is a fixed point, we must keep in mind that there can be many such positions, and they can be located deep down in the recursive structure. A Fix F is a layered tree of F structures. When we finally find an i, we must therefore be able to give an F-context for i for the layer in which the input actually occurs. Now F actually has more inputs than Fix F, so the original index i corresponds to the index inl i for F. We then need a path from the layer where the hole is back to the top. To store this path, we define a datatype of context-stacks Ctxs. This stack consists of yet more F-contexts, but each of the holes in these F-context must correspond to a recursive occurrence, i.e., an index marked by inr:

```
Ctx (Fix F) i r o = ∃ (λ m → Ctx F (inl i) (r | μ F r) m
                          × Ctxs F m r o)

data Ctxs {I O : Set} (F : (I + O) ▶ O) (i : O) (r : Indexed I)
     : Indexed O where
  empty : Ctxs F i r i
  push  : {m o : O} → Ctx F (inr i) (r | μ F r) m →
          Ctxs F m r o → Ctxs F i r o
```

Note that the stack of contexts Ctxs keeps track of two output indices, just like a single context Ctx. A value of type Ctxs F i r o denotes a stack of contexts for a code F, focused on a hole with type index i, on an expression of type index o. This stack is later reused in the higher-level navigation functions (Section 3.4).

3.2 Plugging holes in contexts

A basic operation on contexts is to replace the hole by a value of the correct type; this is called "plugging". Its type is unsurprising: given a code C, a context on C with hole of type index i, and a value of this same index, plug returns the plugged value as an interpretation of C:

```
plug : {I O : Set} {r : Indexed I} {i : I} {o : O} →
       (C : I ▶ O) → Ctx C i r o → r i → ⟦ C ⟧ r o
```

Plugging is not defined for the codes with an empty context type. For I, pattern-matching on the context gives us a proof that the value to plug has the right type, so we return it:

```
plug Z     ()   r
plug U     ()   r
plug (! o) ()   r
plug (I i) refl r = r
```

Re-indexing proceeds plugging recursively, after matching the equality proofs:

```
plug (F ↗ f ↘ g) (some (some (refl , refl , c))) r = plug F c r
```

Plugging on a sum proceeds recursively on the alternatives. Plugging on a product has two alternatives, depending on whether the hole lies on the first or on the second component. Plugging on a composition F ◎ G proceeds as follows: we obtain two contexts, an F-context c with a G-shaped hole, and a G-context d with a hole of the type that we want to plug in. So we plug r into d, and the resulting G is then plugged into c.

```
plug (F ⊕ G) (inl c)       r = inl (plug F c r)
plug (F ⊕ G) (inr c)       r = inr (plug G c r)
plug (F ⊗ G) (inl (c , g)) r = plug F c r , g
plug (F ⊗ G) (inr (f , c)) r = f          , plug G c r
plug (F ◎ G) (some (c , d)) r = plug F c (plug G d r)
```

Plugging into a fixed-point structure is somewhat similar to the case of composition, only that instead of two layers, we now deal with an arbitrary number of layers given by the stack. We can plug our r into the first context c, and then we unwind the stack using an auxiliary function unw. Once the stack is empty we are at the top and done. Otherwise, we proceed recursively upwards, plugging each level as we go:

```
plug {r = s} {o = o} (Fix F) (some {m} (c , cs)) r =
  unw m cs ⟨ plug F c r ⟩ where
    unw : ∀ m → Ctxs F m s o → ⟦ Fix F ⟧ s m → ⟦ Fix F ⟧ s o
    unw .o empty          x = x
    unw m (push {o} c cs) x = unw o cs ⟨ plug F c x ⟩
```

Finally, plugging on Σ proceeds recursively, using the code associated with the index packed in the context. For isomorphisms we proceed recursively on the new code and apply the to conversion function to the resulting value:

```
plug (Σ f) (some {i} c) r = some (plug (f i) c r)
plug {r = s} {o = o} (Iso C D e) x r with e s o
plug {o = o} (Iso C D e) x r | ep = _≃_.to ep (plug C x r)
```

3.3 Primitive navigation functions: first and next

With plug we can basically move up in the zipper: after plugging a hole we are left with a value of the parent type. To move down, we need to be able to split a value into its first child and the rest. This is the task of first:

```
first : {I O : Set} {r : Indexed I} {o : O} {R : Set} →
        (C : I ▶ O) → ((i : I) → r i → Ctx C i r o → Maybe R) →
        ⟦ C ⟧ r o → Maybe R
```

We write first in continuation-passing style. One should read it as a function taking a value and returning a context with a hole at the first (i.e. leftmost) possible position, the value previously at that position and its index. These are the three arguments to the continuation function. Since not all values have children, we might not be able to return a new context, so we wrap the result in a Maybe. Note that first (and not the caller) picks the index of the hole according to the first input that it can find.

There are no values of void type, so that case is impossible. For unit and tag types, there are no elements, so the split fails:

```
first Z     k ()
first U     k x = nothing
first (! o) k x = nothing
```

For I there is exactly one child, which we return by invoking the continuation:

```
first (I i) k x = k i x refl
```

For re-indexing and sums we proceed recursively, after adapting the continuation function to the new indices and context appropriately:

```
first (F ↗ f ↘ g) k x =
  first F (λ i' r c → k (f i') r (some (some (refl , (refl , c))))) x
first (F ⊕ G) k (inl x) = first F (λ i r c → k i r (inl c)) x
first (F ⊕ G) k (inr x) = first G (λ i r c → k i r (inr c)) x
```

On a product we have a choice. We first try the first component, and only in case of failure (through plusMaybe) we try the second:

```
first (F ⊗ G) k (x , y) =
  plusMaybe (first F (λ i r c → k i r (inl (c , y))) x)
            (first G (λ i r c → k i r (inr (x , c))) y)
```

Composition follows the nested structure: we first split the outer structure, and if that is successful, we call first again on the obtained inner structure:

```
first (F ◎ G) k x = first F (λ m s c →
                      first G (λ i r d → k i r (some (c , d))) s) x
```

Fixed points require more care. We use two mutually-recursive functions to handle the possibility of having to navigate deeper into recursive structures until we find an element, building a stack of contexts as we go. Note that the type of input indices changes once we go inside a fixed point. If we obtain a split on an inl-marked value, then that is an input index of the outer structure, so we are done. If we get a split on an inr-marked value, we have hit a recursive occurrence. We then descend into that by calling fstFix again, and retain the current layer on the context stack:

```
first {I} {O} {r} {o} {R} (Fix F) k x = fstFix x empty where
  mutual
    fstFix : {m : O} → μ F r m → Ctxs F m r o → Maybe R
    fstFix ⟨ x ⟩ cs = first F (contFix cs) x

    contFix : {m : O} → Ctxs F m r o → (i : I + O) →
              (r | μ F r) i → Ctx F i (r | μ F r) m → Maybe R
    contFix cs (inl i) r c = k i r (some (c , cs))
    contFix cs (inr i) r c = fstFix r (push c cs)
```

As usual, splitting on a Σ proceeds recursively, and on isomorphisms we apply conversion functions as necessary:

```
first (Σ f) k (some {i'} y) =
  first (f i') (λ i r c → k i r (some c)) y
first {r = r} {o = o} (Iso C D e) k x with e r o
first (Iso C D e) k x | ep = first C k (_≃_.from ep x)
```

Another primitive navigation function is next, which, given a current context and an element which fits in the context, tries to move the context to the next element to the right, producing a new context and an element of a compatible (and possibly different) type:

```
next : {I O : Set} {r : Indexed I} {o : O} {R : Set} →
       (C : I ▶ O) →
       ((i : I) → r i → Ctx C i r o → Maybe R) →
       {i : I} → Ctx C i r o → r i → Maybe R
```

Its implementation is similar to that of first, so we omit it.

3.4 Derived navigation

Given the primitives plug, first, and next, we are ready to define high-level navigation functions, entirely hiding the context from the user.

We are going to define a zipper data structure that enables the user to navigate through a structure defined by a fixed point on the outside. We can then efficiently navigate to all the recursive positions in that structure. Several variations of this approach—such as a zipper that also allows navigating to parameter positions—are possible.

While we are traversing a structure, we keep the current state in a datatype that we call a *location*. It contains the subtree that is currently in focus, and a path up to the root of the complete tree. The path is a stack of one-hole contexts, and we reuse the Ctxs type from Section 3.1 to hold the stack:

```
data Loc {I O : Set} (F : (I + O) ▶ O)
         (r : Indexed I) (o : O) : Set where
  loc : {o' : O} → ⟦ Fix F ⟧ r o' → Ctxs F o' r o → Loc F r o
```

The high-level navigation functions all have the same type:

```
Nav : Set₁
Nav = ∀ {I O} {F : (I + O) ▶ O} {r : Indexed I} {o : O}
    → Loc F r o → Maybe (Loc F r o)
```

Given a location, we might be able to move to a new location, keeping the same code, interpretation for recursive positions, and output type index. We need to allow for failure since, for instance, it is not possible to move down when there are no children.

Moving down corresponds to splitting the context using first:

```
down : Nav
down {I} {O} {F} {r} {o} (loc {i'} ⟨ x ⟩ cs) = first F f x
  where f : (i : I + O) → (r ⌊ μ F r ⌋) i →
            Ctx F i (r ⌊ μ F r ⌋) i' → Maybe (Loc F r o)
        f (inl i) r d = nothing
        f (inr i) r d = just (loc r (push d cs))
```

The continuation function f expresses the behavior for the different kinds of input positions: we do not descend into parameters, and on recursive calls we build a new location by returning the tree in focus and pushing the new layer on the stack.

Moving up corresponds to plugging in the current context. It fails if there is no context, meaning we are already at the root:

```
up : Nav
up (loc x empty)          = nothing
up {F = F} (loc x (push c cs)) = just (loc ⟨ plug F c x ⟩ cs)
```

Moving to the right corresponds to getting the next child. We process the results of next as in down:

```
right : Nav
right (loc x empty) = nothing
right {I} {O} {F} {r} {o} (loc x (push {m} c cs)) =
  next F f c x
  where f : (i : I + O) → (r ⌊ μ F r ⌋) i →
            Ctx F i (r ⌊ μ F r ⌋) m → Maybe (Loc F r o)
        f (inl i) r d = nothing
        f (inr i) r d = just (loc r (push d cs))
```

The functions presented so far allow reaching every position on the datatype. Other navigation functions, such as to move left, can be added in a similar way.

Finally, we provide operations to start and stop navigating a structure.

```
enter : ∀ {I O} {F : (I + O) ▶ O} {r : Indexed I} {o : O} →
        ⟦ Fix F ⟧ r o → Loc F r o
enter x = loc x empty
leave : ∀ {I O} {F : (I + O) ▶ O} {r : Indexed I} {o : O} →
        Loc F r o → Maybe (⟦ Fix F ⟧ r o)
leave (loc x empty)      = just x
leave (loc x (push h t)) = up (loc x (push h t)) >>= leave
```

To enter we create a location with an empty context, and to leave we move up until the context is empty. We use `_>>=_ : {A B : Set} → Maybe A → (A → Maybe B) → Maybe B` as the monadic bind to combine Maybe operations.

It is also useful to be able to manipulate the focus of the zipper. The function update applies a type-preserving function to the current focus:

```
update : ∀ {I O} {F : (I + O) ▶ O} {r : Indexed I} →
         ⟦ Fix F ⟧ r ⇉ ⟦ Fix F ⟧ r → Loc F r ⇉ Loc F r
update f (loc x i) = loc (f _ x) i
```

3.5 Examples

We now show how to use the zipper on the Rose datatype of Section 2.6. The representation type of rose trees is non-trivial since it uses composition with lists (and therefore contains an internal fixed point). Let us define an example tree:

```
treeB : Rose ℕ → Rose ℕ
treeB t = fork 5
            (fork 4 [] :: (fork 3 [] :: (fork 2 [] :: (fork 1
              (t :: []) :: []))))
tree : Rose ℕ
tree = treeB (fork 0 [])
```

Our example tree has a node 5 with children numbered 4 through 1. The last child has one child of its own, labelled 0.

Now we define a function that navigates through this tree by entering, going down (into the child labelled 4), moving right three times (to get to the rightmost child), descending down once more (to reach the child labelled 0), and finally increments this label:

```
navTree : Rose ℕ → Maybe (Rose ℕ)
navTree t = down (enter (fromRose t)) >>=
            right >>= right >>= right >>= down >>=
            just ∘ (update f _) >>=
            leave >>= just ∘ toRose where
  f : (i : ⊤) → ⟦ 'Rose' ⟧ (const ℕ) i → ⟦ 'Rose' ⟧ (const ℕ) i
  f tt = map 'Rose' (↑ suc) tt
```

Since the navigation functions return Maybe, but other functions (e.g. enter and fromRose) do not, combining these functions requires care. However, it is easy to define a small combinator language that overcomes this problem and simplifies writing traversals with the zipper.

We can check that our traversal behaves as expected:

```
navTreeExample : navTree tree ≡ just (treeB (fork 1 []))
navTreeExample = refl
```

We have shown how to define a zipper for our universe of indexed functors. In particular, we show a type of one-hole contexts for fixed points, using a stack of contexts that is normally used only for the higher-level navigation functions. Even though our zipper

is more complex than that of Multirec, it operates through all the codes in our universe, meaning, for instance, that we can now zip through indexed datatypes.

4. Functor laws

The functorial map of Figure 1 obeys the usual functor laws, which in the setting of indexed sets and index-preserving functions take the following form:

```
map_id : {I O : Set} {r : Indexed I} (C : I ▶ O) →
         map {r = r} C id⇉ ≐ id⇉
map_∘ : {I O : Set} {r s t : Indexed I}
        (C : I ▶ O) (f : s ⇉ t) (g : r ⇉ s) →
        map C (f ∘⇉ g) ≐ map C f ∘⇉ map C g
```

Note that $\_\doteq\_$ encodes pointwise equality and $\_\circ_{\rightrightarrows}\_$ denotes composition of index-preserving functions. We cannot use the standard propositional equality here because Agda's propositional equality on functions amounts to intensional equality, and we cannot prove these functions to be intensionally equal, except by postulating an extensionality axiom.

The first step to prove the laws is to define a congruence lemma stating that mapping equivalent functions results in equivalent maps:

```
map_cong : {I O : Set} {r s : Indexed I} {f g : r ⇉ s}
           (C : I ▶ O) → f ≐ g → map C f ≐ map C g
map_cong Z          ip i ()
map_cong U          ip i x = refl
map_cong (I i')     ip i x = ip i' x
map_cong (F ↗ f ↘ g) ip i x = map_cong F (ip ∘ f) (g i) x
```

For I we use the proof of equality of the functions. For re-indexing we need to adapt the indices appropriately. Sums, products, and composition proceed recursively and rely on congruence of the constructors:

```
map_cong (F ⊕ G) ip i (inl x) = cong≡ inl (map_cong F ip i x)
map_cong (F ⊕ G) ip i (inr x) = cong≡ inr (map_cong G ip i x)
map_cong (F ⊗ G) ip i (x , y) = cong≡₂ _,_ (map_cong F ip i x)
                                           (map_cong G ip i y)
map_cong (F ⊙ G) ip i x       = map_cong F (map_cong G ip) i x
map_cong (! o')  ip i x       = refl
```

For Σ, fixed points, and isomorphisms, the proof also proceeds recursively and by resorting to congruence of equality as necessary:

```
map_cong (Σ g)  ip i (some x) =
  cong≡ some (map_cong (g _) ip i x)
map_cong (Fix F) ip i ⟨ x ⟩   =
  cong≡ ⟨_⟩ (map_cong F (∥_cong ip (map_cong (Fix F) ip)) i x)
map_cong {r = r} {s = s} (Iso C D e) ip i x =
  cong≡ (_≃_.to (e s i)) (map_cong C ip i (_≃_.from (e r i) x))
```

For fixed points, we use a lemma regarding congruence of the $\_\|\_$ operator:

```
∥_cong : {I J : Set} {r u : Indexed I} {s v : Indexed J}
         {f₁ f₂ : r ⇉ u} {g₁ g₂ : s ⇉ v} →
         f₁ ≐ f₂ → g₁ ≐ g₂ → f₁ ∥ g₁ ≐ f₂ ∥ g₂
```

We are now able to prove $\mathsf{map}_{\mathsf{id}}$ and $\mathsf{map}_{\circ}$. The code is similar to that for $\mathsf{map}_{\mathsf{cong}}$, and can be seen in the code bundle. Note that the proofs establishing that the conversion functions really constitute isomorphisms are essential for the definition of the Iso case.

5. Generic decidable equality

Generic functions can be defined by instantiating standard recursion patterns such as the catamorphism defined in Section 2.5, or directly, as a type-indexed computation by pattern-matching on the code. Here we show how to define a semi-decidable equality for our universe; in case the compared elements are equal, we return a proof of the equality. Otherwise, no proof is returned—we omit the proof of difference for brevity, but it can be found in the code bundle.

The type of decidable equality is:

```
deqt : {I O : Set} {r : Indexed I} (C : I ▶ O) →
       SemiDec r → SemiDec (⟦ C ⟧ r)
```

Note that to compute the equality SemiDec (⟦ C ⟧ r) we need the equality on the respective recursive indexed functors (SemiDec r). The type constructor SemiDec matches the expected type of semi-decidable equality: for all possible indices, given two indexed functors we either return proof of their equality or fail:

```
SemiDec : ∀ {I} → Indexed I → Set
SemiDec r = ∀ i → (x y : r i) → Maybe (x ≡ y)
```

The type constructor $\_\nparallel\_$ is a variation on $\_\|\_$ adapted to the type of SemiDec, necessary for the Fix alternative:

```
infixr 5 _∦_
_∦_ : {I J : Set} {r : Indexed I} {s : Indexed J} →
      SemiDec r → SemiDec s → SemiDec (r | s)
(f ∦ g) (inl x) = f x
(f ∦ g) (inr y) = g y
```

Decidable equality is impossible on Z, trivial on U, and dispatches to the supplied function f on the selected index i for I:

```
deqt Z   f o () y
deqt U   f o tt tt = just refl
deqt (I i) f o x  y = f i x y
```

Re-indexing proceeds recursively on the arguments, after adjusting the recursive equality and changing the output index:

```
deqt (F ↗ f ↘ g) h o x y = deqt F (h ∘ f) (g o) x y
```

Sums are only equal when both components have the same constructor. In that case, we recursively compute the equality, and apply congruence to the resulting proof with the respective constructor:

```
deqt (F ⊕ G) f o (inl x) (inr y) = nothing
deqt (F ⊕ G) f o (inr x) (inl y) = nothing
deqt (F ⊕ G) f o (inl x) (inl y) =
  mapMaybe (cong≡ inl) (deqt F f o x y)
deqt (F ⊕ G) f o (inr x) (inr y) =
  mapMaybe (cong≡ inr) (deqt G f o x y)
```

The product case follows similarly, using a congruence lifted to two arguments:

```
deqt (F ⊗ G) f o (x₁ , x₂) (y₁ , y₂) =
    deqt F f o x₁ y₁ >>=
  λ l → deqt G f o x₂ y₂ >>=
  λ r → just (cong≡₂ _,_ l r)
```

A composition F ⊙ G represents a code F containing Gs at the recursive positions. Equality on this composition is the equality on F using the equality on G for the recursive positions. Tagging is trivial after pattern-matching:

```
deqt (F ⊙ G) f o  x    y    = deqt F (deqt G f) o x y
deqt (! o)   f .o refl refl = just refl
```

For Σ, we first check if the witnesses (the first components of the dependent pair) are equal. Here, we make essential use of the fact that the type of the witnesses is representable within the universe, so we can reuse the decidable equality function we are just defining. If the witnesses are equal, we proceed to compare the elements, using the code produced by the index, and apply congruence with some to the resulting proof:

```
deqt (Σ {C = C} g) f o (some {i₁} x) (some {i₂} y)
  with deqt {r = λ _ → _} C (λ ()) tt i₁ i₂
deqt (Σ g) f o (some {i} x) (some y) | nothing = nothing
deqt (Σ g) f o (some {i} x) (some y) | just refl =
  mapMaybe (cong≡ some) (deqt (g i) f o x y)
```

Equality for fixed points uses the auxiliary operator _⋔_ described above to apply f to parameters and deqt to recursive calls:

```
deqt (Fix F) f o ⟨ x ⟩ ⟨ y ⟩ =
  mapMaybe (cong≡ ⟨_⟩) (deqt F (f ⋔ deqt (Fix F) f) o x y)
```

Finally, equality for Iso uses the proof of equivalence of the isomorphism. We omit that case here as it is lengthy and unsurprising.

We are now ready to test our decidable equality function:

```
deqtℕ : (m n : ℕ) → Maybe (m ≡ n)
deqtℕ = deqt {r = (λ ())}'ℕ' (λ ()) tt
```

The type of natural numbers has no input indices, therefore we supply the absurd function λ () as the argument for the equality on the inputs.

```
deqtExample₁ : deqtℕ (suc zero) (suc zero) ≡ just refl
deqtExample₁ = refl

deqtExample₂ : deqtℕ (suc zero) zero ≡ nothing
deqtExample₂ = refl
```

For lists of naturals, we supply the equality on naturals as the argument for the equality on the list inputs:

```
deqt[] : {A : Set} → ((x y : A) → Maybe (x ≡ y)) →
         (l₁ l₂ : [A]) → Maybe (l₁ ≡ l₂)
deqt[] {A} f x y = deqt {r = const A}'ListE' g tt x y
  where g : (i : ⊤) → ((x y : const A i) → Maybe (x ≡ y))
        g tt = f

l₁ : [ℕ]
l₁ = zero :: suc zero :: []
l₂ : [ℕ]
l₂ = zero :: zero :: []

deqtExample₃ : deqt[] deqtℕ l₁ l₁ ≡ just refl
deqtExample₃ = refl

deqtExample₄ : deqt[] deqtℕ l₁ l₂ ≡ nothing
deqtExample₄ = refl
```

6. Conclusion

In this paper we have explained a universe of indexed functors for dependently typed datatype-generic programming in Agda which is both intuitive, in the sense that its codes map naturally to datatype features, and powerful, since it supports a wide range of datatypes and allows defining almost all the datatype-generic behavior we encounter.

The key features of our approach are: support for parameterized datatypes and recursive positions in a uniform way, fixed points as part of the universe, support for general composition, and the possibility to incorporate isomorphisms between datatypes into the universe. These features make it possible to reuse codes once defined, and to make normal user-defined Agda datatypes available for use in datatype-generic programs.

Along the way we have seen how proofs of correctness easily integrate with indexed functors, both in the universe and in the generic functions defined. Furthermore, we have shown a generalisation of the zipper operation to our universe, allowing for efficient and type-safe generic traversals.

Options for future work include generalizing from the zipper to dissection (McBride 2008) as well as refining the universe further, for example by allowing generalization over arity and datatype kind, both present in the work of Weirich and Casinghino (2010).

The original goal that inspired this work was to overcome the limitations of the Multirec library (Rodriguez Yakushev et al. 2009), which can handle mutually recursive families of datatypes, but does not support parameterized datatypes or composition. While we have overcome these limitations, we have done so by moving from Haskell to Agda. While Agda clearly has many advantages for generic programming, Haskell is currently still superior when it comes to writing practical code. We hope that we can summon up the usual established trickery of encoding dependent types back in Haskell, and together with the latest improvements in the treatment of type equalities in GHC 7 obtain a reasonably elegant encoding of indexed functors in Haskell, bringing the power of this approach to a wider audience.

Acknowledgments

This work has been partially funded by the Portuguese Foundation for Science and Technology (FCT) via the SFRH/BD/35999/2007 grant. We thank Pierre-Evariste Dagand, Johan Jeuring, Steven Keuchel, and the anonymous reviewers for the helpful feedback. Andres Löh particularly thanks Conor McBride for several enlightening discussions on the topic of indexed containers and functors.

References

T. Altenkirch, C. McBride, and P. Morris. Generic programming with dependent types. In *DGP'07*, pages 209–257. Springer-Verlag, 2007.

R. Bird and L. Meertens. Nested datatypes. In J. Jeuring, editor, *MPC'98*, volume 1422 of *LNCS*, pages 52–67. Springer, 1998.

J. Chapman, P.-E. Dagand, C. McBride, and P. Morris. The gentle art of levitation. In *ICFP'10*, pages 3–14. ACM, 2010.

R. Hinze, J. Jeuring, and A. Löh. Type-indexed data types. In *MPC'02*, volume 2386 of *LNCS*, pages 148–174. Springer, 2002.

G. Huet. The zipper. *JFP*, 7(5):549–554, 1997.

P. Jansson and J. Jeuring. PolyP—a polytypic programming language extension. In *POPL'97*, pages 470–482. ACM, 1997.

C. McBride. The derivative of a regular type is its type of one-hole contexts, 2001. Unpublished manuscript.

C. McBride. Clowns to the left of jokers to the right (pearl): dissecting data structures. In *POPL'08*, pages 287–295, 2008.

P. Morris. *Constructing Universes for Generic Programming*. PhD thesis, The University of Nottingham, Nov. 2007.

T. van Noort, A. Rodriguez Yakushev, S. Holdermans, J. Jeuring, and B. Heeren. A lightweight approach to datatype-generic rewriting. In *WGP '08*, pages 13–24. ACM, 2008.

U. Norell. Dependently typed programming in Agda. In P. Koopman, R. Plasmeijer, and D. Swierstra, editors, *AFP'08*, volume 5832 of *LNCS*, pages 230–266. Springer, 2009.

A. Rodriguez Yakushev, S. Holdermans, A. Löh, and J. Jeuring. Generic programming with fixed points for mutually recursive datatypes. In *ICFP'09*, pages 233–244. ACM, 2009.

S. Weirich and C. Casinghino. Arity-generic datatype-generic programming. In *PLPV'10*, pages 15–26. ACM, 2010.