The Right Kind of Generic Programming

José Pedro Magalhães
Department of Computer Science, University of Oxford
jpm@cs.ox.ac.uk

Abstract

Haskell is known for its strong, static type system. A good type system classifies values, constraining the valid terms of the language and preventing many common programming errors. The Glasgow Haskell Compiler (GHC) has further extended the type language of Haskell, adding support for type-level computation and explicit type equality constraints.

Type-level programming is used to classify values, but types themselves have remained notoriously unclassified. Recently, however, the kind-level language was extended with support for user-defined kinds and kind polymorphism. In this paper we revisit generic programming approaches that rely heavily on type-level computation, and analyse how to improve them with the extended kind language. For instructive purposes, we list a series of advantages given by the new features, and also a number of shortcomings that prevent desirable use patterns.

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

Keywords datatype-generic programming, Haskell, kind polymorphism, datakinds, Agda

1. Introduction

Datatype-generic programming (Gibbons 2007) approaches in Haskell have evolved from early pre-processor approaches like PolyP (Jansson and Jeuring 1997) and Generic Haskell (Löh 2004) to the large number of embedded libraries seen nowadays (Rodriguez Yakushev et al. 2008). Most of the recent libraries, such as Regular (Van Noort et al. 2008), Multirec (Rodriguez Yakushev et al. 2009), Instant Generics (Chakravarty et al. 2009; Van Noort et al. 2010), and the framework for generic programming in GHC (Magalhães et al. 2010a), rely heavily on advanced type-level features such as Generalised Algebraic Data Types (GADTs, Schrijvers et al. 2009) and type families (Sulzmann et al. 2007). However, all the approaches share a lack of kind discipline; for instance, types used internally for representing other datatypes all live in the kind $\star$, even though conceptually they define a separate class of types. The lack of proper kinds when doing complex type-level programming results in simple type mistakes that go unnoticed, especially during library development, but appear later, normally through painful debugging and unhelpful error messages. Moreover, most approaches have difficulties supporting datatypes of different kinds, often resorting to code duplication for accommodating each kind separately. Visible examples of this are the $dataCast_1$/$dataCast_2$ functions in the Scrap Your Boilerplate approach (SYB, Lämmel and Peyton Jones 2003, 2004), or the *Generic*/*Generic*$_1$ classes of Magalhães et al. (2010a).

This is all bound to change, though, with the recent addition of kind polymorphism and user-defined kinds to GHC (Yorgey et al. 2012). The new features, available as a "technology preview" in GHC version 7.4, and due to be fully supported in version 7.6, bring a whole new range of programming techniques within reach. In this paper we explore the potential of these new features for improving generic programming, giving "the right kind" annotations to some existing approaches, and analysing in detail the improvements achieved. Naturally, as soon as new features are introduced into the wild, programmers begin using them, exploring dark corners, and demanding new and more complex extensions. We will play our role in that task, pointing out how current limitations hinder further innovations, and providing insight for future developments in the language.

Specifically, the contributions of this paper are the following:

- Multiple examples of the usefulness of kind polymorphism and user-defined kinds for generic programming in Haskell, by redefining existing implementations to make use of the new features.
- Pointing out not only the advantages of the new features, but also their shortcomings, identifying key limitations and providing insight for future developments.
- Defining a new library for generic programming in Haskell that supports datatypes with arbitrary number of parameters, mutually-recursive datatypes, and composition. This library had been previously defined in the dependently-typed language Agda (Norell 2007), but is now expressible in Haskell.

We introduce the important concepts as they appear, but due to the exploratory nature of this paper with regards to GHC extensions and generic programming libraries, we assume some familiarity with both. The reader is referred to related work for in-depth coverage of particular extensions or libraries.

In the remainder of this paper we first introduce kind polymorphism and user-defined kinds through datatype promotion (Section 2). We then use the new features for improving the Regular (Section 3) and Multirec (Section 4) libraries, and analyse how to improve SYB given a kind-polymorphic Typeable (Section 5). In Section 6 we discuss how to improve the built-in support for generic programming in GHC by formulating a Haskell encoding of the Agda generic programming approach of Löh and Magalhães (2011), made possible by the new extensions. Finally, we discuss future work and conclude in Section 7.

WGP'12, September 9, 2012, Copenhagen, Denmark.

Notation

In this paper we will often use the same name for constructors, types, and kinds. To reduce the potential for confusion, constructors are coloured in blue (like *Just*), types and type classes in orange (like *Maybe Int*), and kinds in green (like $\star$). A black and white version is available from `http://dreixel.net/research/pdf/trkgp_nocolor.pdf`.

2. User kinds and kind polymorphism

Yorgey et al. (2012) introduce two new features in GHC: user-defined kinds (through datatype promotion) and kind polymorphism. We introduce the two features in this section by means of example uses, and refer the reader to the original paper for implementation details.

2.1 Datatype promotion

Consider the datatype of Peano-style natural numbers:

$$\textbf{data}\ \mathit{Nat} = \mathit{Ze} \mid \mathit{Su}\ \mathit{Nat}$$

Through datatype promotion, enabled with the flag `-XDataKinds`, this declaration gives rise not only to a type *Nat* inhabited by the expressions *Ze* and *Su n* for every $n :: \mathit{Nat}$, but also to a kind *Nat* inhabited by the types *Ze* and *Su v* for every $v :: \mathit{Nat}$.

Only types of kind $\star$ can contain values; types of other kinds are always uninhabited. Nonetheless, we can use the new *Nat* kind to define a type of length-indexed lists, or vectors:

$$\begin{aligned}&\textbf{data}\ \mathit{Vec} :: \star \to \mathit{Nat} \to \star\ \textbf{where}\\ &\quad \mathit{Nil}\ \ :: \mathit{Vec}\ \alpha\ \mathit{Ze}\\ &\quad \mathit{Cons} :: \alpha \to \mathit{Vec}\ \alpha\ v \to \mathit{Vec}\ \alpha\ (\mathit{Su}\ v)\end{aligned}$$

This vector type can be used to prevent a usual runtime error, that of asking for the first element of an empty list. Since vectors keep track of their length, we can define a function that returns the first element of a non-empty vector:

$$\begin{aligned}&\mathit{headVec} :: \mathit{Vec}\ \alpha\ (\mathit{Su}\ v) \to \alpha\\ &\mathit{headVec}\ (\mathit{Cons}\ a\ t) = a\end{aligned}$$

This function cannot fail, as it does not operate on empty vectors; a call of the form *headVec Nil* does not typecheck.

We could define *Vec* without `-XDataKinds`; the usual trick is to define empty datatypes that stand for a new kind and use them carefully. So we could define:

$$\begin{aligned}&\textbf{data}\ \mathit{Ze}'\\ &\textbf{data}\ \mathit{Su}'\ v\\ &\textbf{data}\ \mathit{Vec}' :: \star \to \star \to \star\ \textbf{where}\\ &\quad \mathit{Nil}'\ \ :: \mathit{Vec}'\ \alpha\ \mathit{Ze}'\\ &\quad \mathit{Cons}' :: \alpha \to \mathit{Vec}'\ \alpha\ v \to \mathit{Vec}'\ \alpha\ (\mathit{Su}'\ v)\end{aligned}$$

Note the difference in the kind of *Vec′*: both its arguments are of kind $\star$, which means that *Vec′ Int Int* would not trigger a kind error. However, we always expect the second argument to *Vec′* to be either *Ze′* or *Su′*; `-XDataKinds` allows us to make that requirement explicit, and have the compiler enforce it for us, statically, just like the compiler enforces properties on the structure of values by means of their types.

Not all datatypes are currently promoted by GHC; only datatypes with a kind of the form $\star \to \ldots \to \star \to \star$ are promoted. In particular, higher-kinded datatypes and datatypes whose kinds involve promoted types are not promoted. Additionally, datatypes whose constructors are kind polymorphic, involve constraints, or use existential quantification are also not promoted.

2.2 Kind polymorphism

Consider the type of standard Haskell lists:

$$\textbf{data}\ [\alpha] = [] \mid \alpha : [\alpha]$$

With `-XDataKinds` this type gets promoted too! What is, then, the kind of promoted lists? The type of lists is polymorphic, so the kind of promoted lists is polymorphic too, but at the kind level. So for all possible kinds κ, the term $[\kappa]$ denotes a valid kind.

Kind polymorphism is activated with the flag `-XPolyKinds`, which enables kind-polymorphic inference and checking. Consider the following datatype:

$$\textbf{data}\ \mathit{Proxy}\ \sigma = \mathit{Proxy}$$

Without `-XPolyKinds`, the kind of *Proxy* is $\star \to \star$; the kind of the argument σ is defaulted to $\star$, as the Haskell 98 language report states (Peyton Jones 2003). With `-XPolyKinds`, however, the kind of *Proxy* becomes $\forall\kappa.\ \kappa \to \star$. This extra generality gives us the ability to instantiate *Proxy* with types of different kinds. For instance, *Proxy Int*, *Proxy Maybe*, and *Proxy Ze* are all valid types. The usefulness of kind polymorphism will become clearer in the examples of the coming sections.

3. Improving the Regular library

Regular[1] is a simple library for generic programming in Haskell, originally developed to support a rewriting system (Van Noort et al. 2008). It has a fixed-point view on data: the generic representation is a pattern-functor, and a fixed-point operator ties the recursive knot. In the original formulation, this is used to ensure that rewriting meta-variables can only occur at recursive positions of the datatype. Its name derives from the concept of a regular datatype, which is a type that can be represented as a potentially recursive polynomial functor. This definition excludes exponentials (functions), or nested datatypes (Bird and Meertens 1998), among others, which Regular indeed does not support.

3.1 Defining the universe

The first step in improving Regular is to encode its representation types (such as sum, product, unit, etc.) as a universe. Universes are a well-known strategy for implementing generic programming in dependently-typed languages (Morris 2007); we find that they now apply naturally to Haskell with promotion. The basic idea is to define a datatype that encodes the structure of datatypes, and a function that explains how to view this structure as a datatype itself.

Essentially, we want to define a new kind encompassing the representation types of Regular. Imagine the following syntax for introducing kinds directly:

$$\begin{aligned}\textbf{kind}\ \mathit{Universe} = {} & \mathit{U} \mid \mathit{K}\ \star \mid \mathit{I}\\ & \mid \mathit{C}\ \mathit{MetaCon}\ \mathit{Universe}\\ & \mid \mathit{Universe} :+: \mathit{Universe}\\ & \mid \mathit{Universe} :\times: \mathit{Universe}\end{aligned}$$

In this way we would introduce a new kind *Universe*, inhabited by each of the six representation types of Regular. Note in particular how *K*, the type for representing constants, is parametrised over a type of kind $\star$.

Since the only way to define new kinds is through datatype promotion, we must define the universe as a datatype, and hope that it is promoted to the kind above. But how can we define *K* in the datatype version of the universe? There is no datatype argument that is promoted to kind $\star$; so we are forced to abstract over a parameter, which we will later instantiate to kind $\star$ (after promotion):

$$\begin{aligned}\textbf{data}\ \mathit{Universe}\ \mathit{star} = {} & \mathit{U} \mid \mathit{K}\ \mathit{star} \mid \mathit{I}\\ & \mid \mathit{C}\ \mathit{MetaCon}\ (\mathit{Universe}\ \mathit{star})\end{aligned}$$

[1] `http://hackage.haskell.org/package/regular`

```
  | (Universe star) :+: (Universe star)
  | (Universe star) :×: (Universe star)
```

This is slightly cumbersome, and leads us to formulate the first shortcoming of the current implementation of user kinds in GHC.

Shortcoming 1. *We have to define a new datatype, even when all we are interested in is the respective kind.*

From Shortcoming 1 it follows that if we want to have kinds such as $\star$ or $\star \to \star$ as arguments to user-defined kinds, we will have to first abstract from them as kind variables, and later instantiate them to the right kind. Alternatively, GHC could define a built-in datatype *Star* whose promotion is defined to be $\star$.

We will see this instantiation in the interpretation of the Regular universe in Section 3.2. For now, we should discuss the encoding of meta-information for constructors in the C representation type. This meta-information stores the constructor name, its fixity, and associativity, and is important when defining functions such as generic *read* or *show*. The current encoding of Regular generates one empty datatype for each constructor of the type being represented. We propose replacing this with an encoding of the meta-information at the kind level; *MetaCon* should be defined as follows:

```
data MetaCon = MC [Char] Fixity Bool
data Fixity = Prefix | Infix Assoc Int
data Assoc = LeftAssoc | RightAssoc | NotAssoc
```

The kind *MetaCon*, arising from the promotion of the datatype *MetaCon*, now contains all the meta-information available for a constructor; its first argument encodes the constructor name, the second its fixity, and the third controls whether record notation was used. We will see in Section 3.4 exactly how this new representation is an improvement over the current one.

3.2 Interpretation

Together with the Regular universe we must define an interpretation: a value witness for types of kind *Universe*. We define the interpretation for Regular as follows:

```
data ⟦ υ :: Universe ⋆ ⟧ (τ :: ⋆) where
  U' :: ⟦ U ⟧ τ
  K' :: α → ⟦ (K α) ⟧ τ
  I' :: τ → ⟦ I ⟧ τ
  C' :: (Constructor ν) ⇒ ⟦ α ⟧ τ → ⟦ (C (MC ν ϕ ρ) α) ⟧ τ
  L' :: ⟦ α ⟧ τ → ⟦ α :+: β ⟧ τ
  R' :: ⟦ β ⟧ τ → ⟦ α :+: β ⟧ τ
  ×' :: ⟦ α ⟧ τ → ⟦ β ⟧ τ → ⟦ α :×: β ⟧ τ
```

For notational convenience we call the interpretation datatype $[\![\_]\!]$, taking the universe argument υ inside the brackets; the argument for the recursive positions τ comes after the brackets. Because *Universe* is a datatype with constructors such as K, we cannot call the constructor corresponding to the interpretation of K simply K; we are forced to come up with a new name, so we prime all the constructor names. This is a consequence of Shortcoming 1.

The kind of υ is *Universe* $\star$; as promised, the argument to *Universe* is always $\star$. Ideally we would define a kind synonym for *Universe* $\star$, but unfortunately kind synonyms are not implemented yet.[2]

Shortcoming 2. *The lack of support for kind synonyms means we cannot abbreviate kinds.*

[2] On the other hand, we are now free to instantiate *Universe* to kinds other than $\star$; it remains to see whether there is any practical application for this.

The interpretation itself is mostly unsurprising: units have no arguments, constants take a value of the given constant type, and for a recursive position (I') we store the τ variable. This is the way Regular encodes recursion; see the example in Section 3.4 for clarification. Products take two arguments, and a sum can be built in two different ways. The meta-information for constructors introduces a class constraint on the (type-level) name of the constructor. This class provides the value-level operations on meta-information:

```
class Constructor (ν :: [Char]) where
  conName :: Proxy ν → String
  conFixity :: Proxy ν → Fixity
  conFixity = const Prefix
  conIsRecord :: Proxy ν → Bool
  conIsRecord = const False
```

We need to use the *Proxy* datatype (introduced in Section 2.2) because the type of the methods in the class needs to mention at least one of the class type variables, but we cannot take ν as an argument directly since its kind is not $\star$. We suspect that a datatype like *Proxy* is going to appear in most code using kind polymorphism, so it should probably be defined within a module that comes with GHC.

Note also that, at the time of writing, there is no real support for type-level *String*s in GHC: while lists are promoted, *String*s are lists of *Char*, and there is no syntax for using promoted characters. As a workaround, we can assign a unique number to each datatype and use that instead. Note however that type-level *Int*s are not supported either, so we would have to use type-level Peano-style naturals. Also, due to Shortcoming 2, we have to write [*Char*] instead of just *String*. We hope that Iavor Diatchki's work on type-level literals will soon allow us to define these types.[3]

Shortcoming 3. *We lack support for type level naturals and strings.*

3.3 Converting to and from user datatypes

We need a way to convert between user datatypes and their generic representation to provide a seamless user experience. We do not want users to ever have to see the generic representation; they should use their datatypes as usual, and the generic functions should automatically convert to the generic representation as necessary. For this task Regular uses a type class:

```
class Regular (α :: ⋆) where
  type PF α :: Universe ⋆
  from :: α → ⟦ PF α ⟧ α
  to   :: ⟦ PF α ⟧ α → α
```

Regular uses a shallow generic encoding: a value of type α is converted to a one-layer generic representation of type $[\![PF\ \alpha]\!]\ \alpha$; the values at the recursive positions (under I') are of type α, and not generic representations.

Advantage 1. *The kind of the PF type family now clearly establishes that only representation types make up a valid representation.*

3.4 Example encoding

To illustrate the encoding of user datatypes we show how lists are represented in the new encoding of Regular:

```
instance Regular [α] where
  type PF [α] =  (C (MC ?1 (Infix RightAssoc ?3) False) U)
            :+: (C (MC ?2 Prefix False) ((K α) :×: I))
```

[3] `http://hackage.haskell.org/trac/ghc/wiki/TypeNats`

```
from []       = L' (C' U')
from (h : t)  = R' (C' (×' (K' h) (I' t)))
to (L' (C' U'))                  = []
to (R' (C' (×' (K' h) (I' t)))) = h : t
```

Due to the current lack of support for type-level strings and integers (Shortcoming 3), we cannot quite define $?_1$ (the type-level string `"[]"`), $?_2$ (the type-level string `":"`), and $?_3$ (the type-level integer 5).

We also need to give *Constructor* instances for each of the constructor representations:

```
instance Constructor ?1 where
  conName _ = "[]"
instance Constructor ?2 where
  conName _ = ":"
  conFixity _ = Infix RightAssoc 5
```

Note how these constructions are basically a value-level repetition of information that is already present in the types. We hope that recent work on automatic singleton generation (Eisenberg and Weirich 2012) can be used to automate this task.

3.5 Generic functions

Many generic functions can be defined in Regular. We start with the functorial map function, which can in turn be used to define standard recursive morphisms. As the function is indexed over the universe types, we define it using a type class:

```
class GMap (φ :: Universe ⋆) where
  gmap :: (α → β) → ⟦ φ ⟧ α → ⟦ φ ⟧ β
```

Given the kind of *GMap*, it is now clear that this class is used internally in Regular, and user datatypes are not to be instantiated to it.

Advantage 2. *We can only give instances of GMap for representation types.*

We define the generic function by providing an instance for each representation type. Units and constants have nothing to be mapped, whereas at the recursive position we apply the function being mapped. For constructors, sums, and products we simply recurse:

```
instance GMap U where
  gmap f U' = U'
instance GMap (K α) where
  gmap _ (K' a) = K' a
instance GMap I where
  gmap f (I' r) = I' (f r)
instance (GMap φ) ⇒ GMap (C γ φ) where
  gmap f (C' x) = C' (gmap f x)
instance (GMap φ, GMap ψ) ⇒ GMap (φ :+: ψ) where
  gmap f (L' a) = L' (gmap f a)
  gmap f (R' b) = R' (gmap f b)
instance (GMap α, GMap β) ⇒ GMap (α :×: β) where
  gmap f (×' a b) = ×' (gmap f a) (gmap f b)
```

Shortcoming 4. *GHC does not check for exhaustiveness of instances for a promoted datatype (e.g. warn if we forget the instance for U).*

The definition of catamorphism, using *gmap*, remains unchanged:

```
cata :: (Regular α, GMap (PF α))
     ⇒ (⟦ PF α ⟧ β → β) → α → β
cata f = f ∘ gmap (cata f) ∘ from
```

In fact, we can even go further than Shortcoming 4, and argue that the *GMap* (*PF* α) part of in the constraint of *cata* is unnecessary; GHC could check that the *GMap* class has all the necessary instances for the *Universe* $\star$ kind, and then automatically fill-in the appropriate class dictionaries.

Compared to the current implementation of Regular without user kinds, the main advantage of this new way of defining *gmap* is that we can use more precise kinds, preventing common mistakes. Furthermore, a function such as generic *show* is now easier to define given the presence of constructor meta-information at the type level. Consider the class definition for generic *show*, together with the interesting cases for *C*:

```
class GShow (φ :: Universe ⋆) where
  gshow :: (τ → String) → ⟦ φ ⟧ τ → String

instance (GShow φ) ⇒ GShow (C (MC ν Prefix ρ) φ) where
  gshow f (C' x) = "(" ++ conName (Proxy :: Proxy ν)
                   ++ " " ++ gshow f x ++ ")"
instance (GShow φ, GShow ψ)
  ⇒ GShow (C (MC ν (Infix α ι) ρ) (φ :×: ψ)) where
  gshow f (C' (×' x y)) = "(" ++ gshow f x ++ " "
                          ++ conName (Proxy :: Proxy ν)
                          ++ " " ++ gshow f y ++ ")"
```

The function being carried around (f) is used for the recursive positions, and can be ignored for the purposes of this example. What is important to notice is that we can give separate *instances* for prefix and infix constructors. A similar treatment can be given to constructors using record syntax, but we elided the selector meta-information for simplicity.

Advantage 3. *We can match meta-data properties at the type level, allowing generic functions to easily change their behaviour according to meta-data.*

4. Improving Multirec

Multirec (Rodriguez Yakushev et al. 2009) is a library for generic programming supporting families of mutually recursive datatypes. It can be seen as a generalisation of Regular, using indexed functors instead of plain functors for the generic representation. Like Regular, its current implementation [4] is not properly kinded, as the representation types are not encoded as a separate universe.

Fortunately we can improve Multirec in a way similar to Regular, by defining a datatype for the universe and using its promoted constructors as the representation types:

```
data Universe star ι = U | K star | I ι | (Universe star ι) :≫: ι
                     | (Universe star ι) :+: (Universe star ι)
                     | (Universe star ι) :×: (Universe star ι)
                     | (star → star) :∴: (Universe star ι)
```

There are a few important things to notice about Multirec's universe:

- Since Multirec represents datatypes as indexed functors, the universe is parametrised over an ι parameter, representing the family index. This parameter is used in the *I* case to state which type in the family we recurse into.
- A new representation type $\alpha :\gg: \iota$ is used to tag a particular branch α of the representation with an index ι, stating that said branch encodes the type of index ι.

[4] `http://hackage.haskell.org/package/multirec`

- Since version 0.6, Multirec has support for a limited form of composition, which allows representing datatypes that use container types, such as lists or *Maybe*. We encode this with $\phi :\cdot: \alpha$, where $\phi :: \star \to \star$ and α is a representation. Note that we rely on the promotion of the function type $star \to star$ to the kind $\star \to \star$.
- We omit the case for constructor meta-data for simplicity, as it is similar to Regular.

As with Regular, the interpretation is given as a GADT over the (promoted) universe. It is parametrised over a τ type that maps indices to their corresponding types, and a particular index ι; since we are encoding families of datatypes as a single datatype, we use ι to select which datatype we intend to obtain the interpretation for:

```
data [| υ :: Universe ⋆ ι_K |] (τ :: ι_K → ⋆) (ι :: ι_K) where
  U' :: [| U |] τ ι
  K' :: α → [| (K α) |] τ ι
  I' :: τ o → [| (I o) |] τ ι
  ≻' :: [| α |] τ ι → [| α :≻: ι |] τ ι
  L' :: [| α |] τ ι → [| α :+: β |] τ ι
  R' :: [| β |] τ ι → [| α :+: β |] τ ι
  ×' :: [| α |] τ ι → [| β |] τ ι → [| α :×: β |] τ ι
  C' :: φ ([| α |] τ ι) → [| φ :·: α |] τ ι
```

The mediation between representation types and user types is done with two type classes, one providing value conversion (*Fam*, for family), and the other (*El*, for element) assisting in the generation of family indices:

```
newtype I_0 ι = I_0 {unI_0 :: ι}

class Fam (φ :: ⋆ → ⋆) where
  type PF φ :: Universe ⋆ ⋆
  from :: φ ι → ι → [| PF φ |] I_0 ι
  to   :: φ ι → [| PF φ |] I_0 ι → ι

class El (φ :: ι_K → ⋆) (ι :: ι_K) where
  proof :: φ ι
```

The definitions of *Fam* and *El* are better understood through an example, so we show how to encode the following family of mutually recursive types:

```
data Zig  = Zig Zag | ZigEnd
data Zag = Zag [Zig]
```

The argument to *Zag* is a list of *Zig*s. The first step in encoding this family is to define a GADT to serve as witness for the presence of each of the types in the family:

```
data ZigZag ι where
  ZigZag_Zig :: ZigZag Zig
  ZigZag_Zag :: ZigZag Zag
```

We can now define a *Fam* instance for the *ZigZag* family:[5]

```
instance Fam ZigZag where
  type PF ZigZag =  (((I Zag) :+: U) :≻: Zig)
                :+: (([] :·: (I Zig)) :≻:  Zag)
  from ZigZag_Zig (Zig zag)  = L' (≻' (L' (I' (I_0 zag))))
  from ZigZag_Zig (ZigEnd)   = L' (≻' (R' U'))
  from ZigZag_Zag (Zag zigs) =
    R' (≻' (C' (map (I' ∘ I_0) zigs)))
  to ZigZag_Zig (L' (≻' (L' (I' (I_0 zag))))) = Zig zag
  to ZigZag_Zig (L' (≻' (R' U')))             = ZigEnd
  to ZigZag_Zag (R' (≻' (C' zigs)))           =
    Zag (map (λ(I' (I_0 x)) → x) zigs)
```

In the *Fam* class we use a lifted identity type I_0 as the τ argument to the interpretation. This simply means that at the recursive positions we have normal datatype values again; Multirec, like Regular, uses a shallow encoding of data. In the definition of the catamorphism (which we elide for brevity), for instance, this parameter is not fixed to a particular type, allowing a different return type per index, for instance.

We have seen that converting Multirec to make use of better kinds brings considerable changes to the library. However, the changes are mostly mechanical, cause no loss of functionality, and result in a properly kinded library. From the user's perspective there is no difference, as the representations are automatically generated. We have remained faithful to the current definition of the *Fam* class and used indices of kind $\star$, but with the current kind-polymorphic definition of *Universe* we are free to use indices of other kinds. In Section 6.2.1 we show a way to define a *Fam* class that allows more general index kinds.

5. Improving Typeable and SYB

SYB (Lämmel and Peyton Jones 2003, 2004), a library for generic programming with built-in support in GHC, uses runtime type-safe casting to support generic functions with ad hoc cases for specific datatypes.

5.1 Kind-polymorphic Typeable

SYB relies on the Typeable library for runtime type comparison and casting. Currently, Typeable consists of a group of classes, of the following form:

```
data TypeRep
class Typeable (α :: ⋆) where
  typeOf :: α → TypeRep
class Typeable_1 (φ :: ⋆ → ⋆) where
  typeOf_1 :: φ α → TypeRep
```

The datatype *TypeRep* gives a runtime representation of a datatype; we leave its definition abstract as its internals are not important for our discussion. A type of kind $\star \to \star$, such as *Maybe*, gets two instances:

```
instance Typeable_1 Maybe ...
instance (Typeable α) ⇒ Typeable (Maybe α) ...
```

There is a total of eight Typeable classes, from *Typeable* to $Typeable_7$. This is inconvenient not only because of the obvious duplication, but also because:

1. Types of arity higher than seven cannot be given a Typeable instance;
2. Types with arguments of higher ranks (such as $\star \to \star$) cannot be instantiated at all;
3. Types with arguments of user-defined kinds cannot be instantiated either.

Yorgey et al. (2012) already proposed a kind-polymorphic *Typeable* class as follows:

```
class Typeable (φ :: κ) where
  typeOf :: Proxy φ → TypeRep
```

[5] All of the representation code is currently generated automatically by Multirec using Template Haskell (Sheard and Peyton Jones 2002). This support can be retained when using data kinds, since these are already supported in Template Haskell (Eisenberg and Weirich 2012).

The construction of "towers" of Typeable instances for higher-kinded types like *Maybe* can then be condensed into a single, kind-polymorphic instance:

```
instance (Typeable (φ :: κ1 → κ2), Typeable (α :: κ1))
      ⇒ Typeable (φ α) ...
```

We further propose naming the *Typeable* method *typeRep*, instead of *typeOf*, so that we can retain *typeOf*, *typeOf1*, etc. for backwards compatibility:

```
typeOf :: ∀(α :: ⋆). Typeable α ⇒ α → TypeRep
typeOf _ = typeRep (Proxy :: Proxy α)

typeOf1 :: ∀(φ :: ⋆ → ⋆)(α :: ⋆). Typeable φ ⇒ φ α → TypeRep
typeOf1 _ = typeRep (Proxy :: Proxy φ)
```

Note that the code above uses scoped type variables to provide the right type to *Proxy*.

5.2 Kind-polymorphic SYB?

SYB also has a form of duplication to accommodate types of various kinds, which we might hope to remove by using kind polymorphism. This duplication is noticeable in the definition of the extension functions:

```
ext0 :: (Typeable (α :: ⋆), Typeable (β :: ⋆))
     ⇒ φ α → φ β → φ α
ext0 def ext = maybe def id (gcast ext)

ext1 :: (Data α, Typeable1 ψ)
     ⇒ φ α → (∀β. Data β ⇒ φ (ψ β)) → φ α
ext1 def ext = maybe def id (dataCast1 ext)
```

The function *ext1* is just a variant of *ext0* that is used when dealing with types of kind $\star \to \star$. There is also an *ext2* variant. The *dataCast1* function comes from the *Data* class:

```
class Data α where
  dataCast1 :: Typeable1 ψ
            ⇒ (∀β. Data β ⇒ φ (ψ β)) → Maybe (φ α)
  dataCast1 _ = Nothing
```

The default definition of *dataCast1* is suitable for types of kind $\star$, like *Int*. For types of kind $\star \to \star$, like *Maybe*, *gcast1* should be used instead; this function is very similar to *gcast* (used by *ext0*), but while *gcast* uses *typeOf*, *gcast1* uses *typeOf1*:

```
gcast :: (Typeable (α :: ⋆), Typeable β) ⇒ φ α → Maybe (φ β)
gcast x = r where
  r = if typeOf (getArg x) ≡ typeOf (getArg (fromJust r))
      then Just $ unsafeCoerce x
      else Nothing
  getArg :: φ α → α
  getArg = ⊥

gcast1 :: (Typeable1 (ψ :: ⋆ → ⋆), Typeable1 ψ')
       ⇒ φ (ψ α) → Maybe (φ (ψ' α))

gcast1 x = r where
  r = if typeOf1 (getArg x) ≡ typeOf1 (getArg (fromJust r))
      then Just $ unsafeCoerce x
      else Nothing
  getArg :: φ α → α
  getArg = ⊥
```

Note how the difference between *gcast* and *gcast1* is basically confined to their type; the implementation is equivalent, especially in the new Typeable setting where *typeOf* and *typeOf1* are both just *typeRep*.

5.2.1 Example usage—kind monomorphic

Defining a generic function in SYB and extending it with a type-specific case for a container type such as *Maybe* is currently done as follows:

```
newtype Result (α :: ⋆) = Result Int

example :: ∀α. (Data α, Typeable α) ⇒ Result α
example = general `ext1` maybe1 `ext0` maybe0
  where general :: Result α
        general = Result 0

        maybe0 :: Result (Maybe Int)
        maybe0 = Result 1

        maybe1 :: ∀χ. Result (Maybe χ)
        maybe1 = Result 2
```

The *example* function returns:

- *Result* 0 when its type is instantiated to *Result Int*;
- *Result* 1 when its type is instantiated to *Result* (*Maybe Int*);
- *Result* 2 when its type is instantiated to *Result* (*Maybe Char*).

We have to use *ext1* for *maybe1*; using *ext0* would result in an ambiguous variable χ (from the type of *maybe1*) during unification. This is because *general* `` `ext0` `` *maybe1* leads to the following unifications:

- ϕ with *Result*;
- α (from *ext0*) with α (from *example*);
- β with *Maybe* χ.

From *ext0* we get a *Typeable* β constraint, which translates into a *Typeable* (*Maybe* χ) constraint in this case. Using the instance *Typeable* $\alpha \Rightarrow$ *Typeable* (*Maybe* α) we get a *Typeable* χ constraint, which is *not* satisfied. Fixing this is not easy; since only *maybe1* knows χ, we would have to change the type signature of *example* to include the *Typeable* χ constraint, which in turn would require adding a dummy argument of type χ to *example*. Using *ext1* we avoid this problem, since we only get a *Typeable1 Maybe* constraint, which does not introduce any constraints on χ.

5.2.2 Example usage—kind polymorphic

In a kind-polymorphic setting, we can remove this duplication by providing a single function for casting and for extending a function with an ad hoc case:

```
gcast :: ∀(α :: κ1)(β :: κ2)(φ :: κ1 → ⋆)(ψ :: κ2 → ⋆).
         (Typeable α, Typeable β) ⇒ φ α → Maybe (ψ β)
gcast x = r where
  r = if typeRep (getArg x) ≡ typeRep (getArg (fromJust r))
      then Just $ unsafeCoerce x
      else Nothing
  getArg :: ∀φ α. φ α → Proxy α
  getArg = ⊥

ext :: ∀(α :: κ1)(β :: κ2)(φ :: κ1 → ⋆)(ψ :: κ2 → ⋆).
       (Typeable α, Typeable β) ⇒ φ α → ψ β → φ α
ext def = maybe def id ∘ gcast
```

Both *gcast* and *ext* now have a much more general type; generalising α and β's kind also requires generalising the kind of the containers, ϕ and ψ.

We can redefine *example* using the new *ext*:

```
newtype Result (α :: κ) = Result Int

example :: ∀α. (Typeable α) ⇒ Result α
```

```
example = general `ext` maybe0 `ext` maybe1
  where general :: Result α
        general = Result 0
        maybe0 :: Result (Maybe Int)
        maybe0 = Result 1
        maybe1 :: Result Maybe
        maybe1 = Result 2
```

Note that:

1. There is no more duplication; we use a single extension function (*ext*), which uses a single cast function (*gcast*);
2. The case for $maybe_1$ no longer requires quantification over a type variable.
3. The order in which we perform extension is no longer relevant.

Unfortunately the example presented is not general enough. In *example* we are never looking at the input value, and simply return a result that depends on the input type. Most generic functions, however, do inspect their input; this means the inputs have a type of kind $\star$, so we cannot use the trick shown above to handle types of different kinds in a similar fashion. For instance, the generic show function has a special case for $[\alpha]$; we cannot express this case as a function of type $[] \to String$,[6] only as a function of type $\forall\alpha.\ [\alpha] \to String$. In this scenario we are back to the problem described at the end of Section 5.2.1, namely ambiguity of type variable α during unification.

This issue arises also from the way *Typeable* instances are given. Even in the kind-polymorphic setting, we need two *Typeable* instances for lists:

```
instance Typeable [] where ...
instance (Typeable α) ⇒ Typeable [α] where ...
```

This follows from the design of the Typeable library; we want to be able to check that the container type of $[Int]$ and $[Char]$ is the same (using $typeOf_1$), but the fully applied types are distinct (using *typeOf*). The second instance above is the root of our problem: it introduces a $Typeable\ \alpha$ constraint that we cannot adequately propagate from the generic function definition through to *ext*.

There is a good reason to want the current behaviour of Typeable to persist. Again, generic show is a good example: we want to define it by giving a general case, a special case for lists, and an even more special case for lists of characters. At the moment, however, it is not clear how to remove the duplication in SYB using kind polymorphism alone.

6. Improving generic programming in GHC

Having seen the potential of an improved kind system in the implementation of the generic programming approaches discussed so far, we turn our attention to the latest GHC addition for generic programming, implemented in the `GHC.Generics` module, described by Magalhães et al. (2010a). This approach, which we call Deriving,[7] uses an implementation strategy similar to Regular and Multirec, but without using functors as representation types. Recursion is handled without specific abstraction of recursive positions, resulting in a more flexible universe (which allows encoding more user datatypes), but less structured (unable to support the definition of the recursive morphisms, for instance).

[6] This type is even ill-kinded; we could try $Proxy\ [] \to String$, but that will not work either because the *Proxy* type does not store a value, only its type.

[7] Even though it does not really allow defining new derivable type classes; it allows defining classes with a generic default, which results in a very similar usage style to standalone **deriving**.

We could follow the same strategy used for Regular (Section 3) or Multirec (Section 4) to improve Deriving,[8] but instead we decide to tackle a long standing limitation with Deriving. Unlike Regular and Multirec, Deriving does support parametrised datatypes, allowing for the (generic) definition of the standard list *map* on the arguments, for instance. However, it only supports abstracting over one parameter (much like Regular only supports abstracting over one recursive position). So while we can express *map* for lists and *Maybe*, for instance, we cannot express the *bimap* for *Either*. The classes that witness the isomorphism between user datatypes and their representation in Deriving currently look like this:

```
class Generic α where
  type Rep α :: ⋆ → ⋆
  from :: α → (Rep α) χ
  to   :: (Rep α) χ → α

class Generic1 ϕ where
  type Rep1 ϕ :: ⋆ → ⋆
  from1 :: ϕ α → Rep1 ϕ α
  to1   :: Rep1 ϕ α → ϕ α
```

The *Generic* class is used to encode user types of kind $\star$, while $Generic_1$ is used for user types of kind $\star \to \star$. The repetition is evident; while the universe types themselves are not repeated (we encode datatypes without parameters by creating a fake parameter χ that is never used), datatype representation requires potentially two instances, even though they are rather similar. In this section we aim at removing this duplication, while at the same time defining a representation that allows any number of datatype parameters.

6.1 First attempt

A legitimate first approach to this task would be to use the same strategy as proposed by Hesselink (2009) to support parameters in Multirec. A user datatype is associated not only with its generic representation, but also the list of types it is parametrised over:

```
class Generic (α :: ⋆) where
  type Rep α :: Universe ⋆
  type Es  α :: [⋆]
  from :: α → ⟦Rep α⟧ (Es α)
  to   :: ⟦Rep α⟧ (Es α) → α
```

We show a simplified universe with a type for encoding parameters, *P*, that simply takes the index (as a natural number) of the parameter we are interested in:

```
data Universe star = U | K star | P Nat
                   | (Universe star) :+: (Universe star)
                   | (Universe star) :×: (Universe star)
```

We can define a suitable interpretation for this universe given a list of parameters. In the *P* case we store the *n*-th parameter from this list:

```
data ⟦υ :: Universe ⋆⟧ (τ :: [⋆]) :: ⋆ where
  U' :: ⟦U⟧ τ
  K' :: α → ⟦(K α)⟧ τ
  P' :: Nth τ ν → ⟦(P ν)⟧ τ
  L' :: ⟦α⟧ τ → ⟦α :+: β⟧ τ
  R' :: ⟦β⟧ τ → ⟦α :+: β⟧ τ
  ×' :: ⟦α⟧ τ → ⟦β⟧ τ → ⟦α :×: β⟧ τ
```

For this we need a type-level lookup function on lists:

[8] As described in `http://hackage.haskell.org/trac/ghc/wiki/Commentary/Compiler/GenericDeriving?version=51#Kindpolymorphicoverhaul`

```
type family N_th (τ :: [⋆]) (ν :: Nat) :: ⋆
type instance N_th (α : β) Ze     = α
type instance N_th (α : β) (Su ν) = N_th β ν
```

We must be careful never to ask for an index outside the bonds of the list, else we will get a type error. Ideally, τ should have kind $Vec \star \kappa$, where κ is the number of parameters in the type (Vec was defined in Section 2.1), but Vec is not promotable due to its kind, and because it is a GADT.

Shortcoming 5. *GHC does not support GADT promotion.*

In any case, the universe given here will not work. Note, for instance, that we have omitted a case for composition; the original Deriving approach supports composition, as it is clear that datatypes with one parameter can be composed. With multiple parameters, however, composition is not always possible. For example, what should be the result of composing $Either$ with $Maybe$? Furthermore, our type K for recursion is not expressive enough to allow defining generic map, for instance. Consider a family of datatypes, all sharing a single parameter. While mapping a function over the parameters, when we encounter another datatype with K we have to keep mapping inside this new type. Unfortunately K does not keep any information about the structure of its argument (namely if they are parametrised, and what their parameters are), so we cannot properly define map. The bottom line is that generalising our representation to an arbitrary number of parameters also requires an explicit abstraction over recursion, so that we can keep track of which parameters go where. Additionally, if we want to be able to compose the represented types, we either need to simplify composition to a special case (like Multirec does), or we need to be able to specify how each parameter connects to each input.

6.2 Indexed functors in Haskell

Fortunately, generalising the representation types to indexed functors, as shown by Löh and Magalhães (2011), allows flexible parametrisation, mutual recursion through fixed points, and generalised composition to coexist. The resulting library, which we call Indexed, requires a number of dependently-typed programming techniques that remained out of reach for practical Haskell programming so far. However, with the new extensions to the kind language, we can encode a significant portion of the Indexed universe in Haskell.

We do not provide a detailed explanation for the universe and its interpretation; we refer the reader to Löh and Magalhães (2011) for more details. Here we focus mostly on how to bring the approach to Haskell, and what challenges arise.

6.2.1 Universe

The Indexed universe can be seen as a generalisation of Multirec's universe, with the important difference that we now distinguish between *input indices* (ι) and *output indices* (o):

```
data Code ι o = Z
              | U
              | I ι
              | ! (Code ι o) o
              | (Code ι o) :+: (Code ι o)
              | (Code ι o) :×: (Code ι o)
              | ∀ν. (Code ν o) :·: (Code ι ν)
              | ∀ι'. I_X (ι' → ι) (Code ι' o)
              | ∀o'. O_X (o → o') (Code ι o')
              | μ (Code (Sum ι o) o)
data Sum α β = L α | R β
```

Conceptually, input indices stand for parameters, and output indices stand for the datatypes being defined. So a family of two datatypes with three parameters will typically use a type with three inhabitants for ι and a type with two inhabitants for o.

A code Z stands for empty datatypes, while U is used for constructors without arguments. I picks a particular index, while $!$ tags a particular representation with a specific index (similarly to Multirec's $:\!\!\gg\!:$). Sums and products are unsurprising, but composition now requires that the codes being composed have a matching input and output type. Note the existential quantification of the ν type variable; we are relying on promotion of existential quantification! The codes I_X and O_X stand for input and output reindexing, respectively, and are used mostly to allow composition of codes of distinct indices. Finally we have the fixed point code μ; here we take a code where the input indices are a disjoint sum between a parameter index ι and a output type index o, and return a $Code\ \iota\ o$, by closing the recursive positions. For this we also rely on a disjoint sum type Sum.

6.2.2 Interpretation

The interpretation helps clarify the universe, especially in regards to the fixed point operator. We have given all the interpretations so far as a GADT, but for Indexed we need to use a data family:[9]

```
data family ([[ γ :: Code ι o ]]) :: (ι → ⋆) → (o → ⋆)
```

The kind of the interpretation states that given a universe code and a mapping of input indices to concrete types, we can produce a map of output indices to concrete types. We start with the interpretation of units, sums, products, and composition, which are unsurprising:

```
data instance [[ U ]]       τ o = U'
data instance [[ α :+: β ]] τ o = L' ([[ α ]] τ o)
                                | R' ([[ β ]] τ o)
data instance [[ α :×: β ]] τ o = ×' ([[ α ]] τ o) ([[ β ]] τ o)
data instance [[ α :·: β ]] τ o = C' {un[[]]_C :: [[ α ]] ([[ β ]] τ) o}
```

Note how we nest the interpretation for composition; this works only because the kind of $:\!\cdot\!:$ is expressive enough.

An input index is obtained by using the map τ, while tagging is done with an equality constraint as in Multirec:

```
data instance [[ (I ι) ]]     τ o = I' {un[[]]_I :: τ ι}
data instance [[ (! α o') ]] τ o where
  !' :: [[ α ]] τ o → [[ (! α o) ]] τ o
```

For input reindexing we need type-level composition, which we express using a newtype. Output reindexing is easier since we can just apply the transformation to o:

```
newtype FComp φ ψ α = FComp {unFComp :: φ (ψ α)}
data instance [[ (I_X ψ α) ]] τ o = I'_X ([[ α ]] (FComp τ ψ) o)
data instance [[ (O_X φ α) ]] τ o = O'_X ([[ α ]] τ (φ o))
```

We are left with the fixed-point case, which we interpret by applying τ on the left (to parameters) and recursively interpreting on the right (recursive positions). For this we need an auxiliary datatype Sum_1:

```
data instance [[ (μ φ) ]] τ o where
  μ' :: [[ φ ]] (Sum_1 τ ([[ (μ φ) ]] τ)) o → [[ (μ φ) ]] τ o
data Sum_1 (α :: κ_1 → ⋆) (β :: κ_2 → ⋆) :: Sum κ_1 κ_2 → ⋆ where
  L_1 :: α ι → Sum_1 α β (L ι)
  R_1 :: β ι → Sum_1 α β (R ι)
```

[9] Data families are more liberal when matching variable kinds; we need this for the $\mu\ \phi$ case.

6.2.3 Mapping indexed functors

Indexed functors support a map operation, which in turn can be used to define recursive morphisms (Meijer et al. 1991). We define the indexed map function by giving instances to each representation type by means of a type class:

```
infixr 7 :→:
type (φ :→: ψ) = ∀ι. φ ι → ψ ι
class Map (γ :: Code ι_κ o_κ) where
  imap :: (φ :→: ψ) → (⟦γ⟧ φ :→: ⟦γ⟧ ψ)
```

Our maps are index preserving, which we represent as a type synonym $(:\!\to\!:)$.

Units, sums, and products are standard:

```
instance Map U where
  imap _ U' = U'
instance (Map α, Map β) ⇒ Map (α :+: β) where
  imap f (L' x) = L' (imap f x)
  imap f (R' x) = R' (imap f x)
instance (Map α, Map β) ⇒ Map (α :×: β) where
  imap f (×' x y) = ×' (imap f x) (imap f y)
```

For composition we nest the call to *imap*:

```
instance (Map α, Map β) ⇒ Map (α :·: β) where
  imap f (C' x) = C' (imap (imap f) x)
```

For an input index we simply apply the function. Tagging proceeds recursively:

```
instance Map (I ι) where
  imap f (I' x) = I' (f x)
instance (Map α) ⇒ Map (! α o) where
  imap f (!' x) = !' (imap f x)
```

Output reindexing proceeds recursively without problems, as the type of *imap* does not mention the output index. Input reindexing, however, requires lifting the mapping function through the composition:

```
instance (Map α) ⇒ Map (O_X φ α) where
  imap f (O'_X a) = O'_X (imap f a)
instance (Map α) ⇒ Map (I_X ψ α) where
  imap f (I'_X a) = I'_X (imap (FComp ∘ f ∘ unFComp) a)
```

Finally, for fixed points we apply f to parameters (on the left), and recursively map recursive occurrences (on the right). We use an auxiliary function $(\|)$ for this purpose:

```
instance (Map γ) ⇒ Map (μ γ) where
  imap f (μ' x) = μ' (imap (f ‖ imap f) x)
(‖) :: (φ :→: φ') → (ψ :→: ψ')
    → (Sum_I φ ψ) :→: (Sum_I φ' ψ')
(f ‖ _) (L_I x) = L_I (f x)
(_ ‖ g) (R_I x) = R_I (g x)
```

Having defined map, we can now encode standard recursive morphisms, such as ana-, cata-, and hylomorphisms:

```
ana :: (Map γ) ⇒ (ψ :→: ⟦γ⟧ (Sum_I φ ψ))
               → (ψ :→: ⟦(μ γ)⟧ φ)
ana g x = μ' (imap (id ‖ ana g) (g x))

cata :: (Map γ) ⇒ (⟦γ⟧ (Sum_I φ ψ) :→: ψ)
                → (⟦(μ γ)⟧ φ :→: ψ)
cata f (μ' x) = f (imap (id ‖ cata f) x)

hylo :: (Map γ) ⇒ (⟦γ⟧ (Sum_I φ ρ) :→: ρ)
                → (ψ :→: ⟦γ⟧ (Sum_I φ ψ)) → (ψ :→: ρ)
hylo f g x = f (imap (id ‖ hylo f g) (g x))
```

6.2.4 Converting to and from user datatypes

We have some choices regarding how to encapsulate the conversion between user datatypes and the generic representation. One way to do it is to use a strategy similar to Multirec's, using the datatypes themselves as indices:

```
class Fam (φ :: ⋆ → ⋆) where
  type PF φ  :: Code ⋆ ⋆
  type Rec φ :: ⋆ → ⋆
  from :: φ α → α → ⟦PF φ⟧ (Rec φ) α
  to   :: φ α → ⟦PF φ⟧ (Rec φ) α → α
```

In this class definition, the type family *Rec* defines how to handle the recursive positions of the datatype. Since we set the input and output index kinds to $\star$, we are forced to define new datatypes to encode input indices (just like in Multirec). Although this encoding is a valid choice, it would be preferable not to force the indices to be of kind $\star$.

We can conceive a more general *Fam* class like the following:

```
class Fam (o :: o_κ) where
  type PF o :: Code ι_κ o_κ
  type Ix   o :: ι_κ → ⋆
  type Ox   o :: ⋆
  from :: Ox o → ⟦PF o⟧ (Ix o) o
  to   :: ⟦PF o⟧ (Ix o) o → Ox o
```

This definition, however, is rejected by GHC; it assumes that the kind variable ι_κ in the kinds of *PF* and *Ix* are unrelated, and is unable to match them. In fact, for our purposes, ι_κ behaves much like an *associated kind variable*; the input index type can be determined solely by the output index type (as long as we commit to defining a new index type per family and using its promotion as the output index kind).

Shortcoming 6. *We lack a mechanism for treating kind variables independently of types, allowing, among other things, for the definition of kind families (kind-level computations that return a kind).*

We plan to explore further alternatives to encapsulating datatype encodings in the Haskell version of Indexed (see Section 7). For now, we propose using the Multirec-like encoding given before, but for the rest of this paper we will not use the *Fam* class.

6.2.5 Example encodings

To further illustrate the use of Indexed in Haskell we provide a number of example datatype encodings.

Lists We start with the standard Haskell list type. Lists have one parameter, and are a single type, so we use the singleton type $()$ as both input and output index:

```
type List_F = (U :+: ((I (L ())) :×: (I (R ()))))
                :: Code (Sum () ()) ())
type List_C = (μ List_F :: Code () ())
```

The list functor $List_F$ encodes the representation type of lists as a choice between a unit (the empty list constructor) and a product of an element (parameter on the left) and another list (recursive position on the right). We then tie the recursive knot with the fixed-point operator in the definition of the $List_C$ representation type. We give kind annotations to the type synonyms for clarity only; note the use of the promoted singleton kind $()$.

The conversion functions are mostly standard, apart from the fact that we have to instantiate the τ parameter of the interpretation. Recall that its kind is $() \to \star$ for the list setting, and it should map the index to the list parameter. For this we use a *Const* datatype that ignores the index and returns the parameter:

```
data Const α β = Const {unConst :: α}

fromList :: [α] → ⟦ List_C ⟧ (Const α) ()
fromList []       = μ' (L' U')
fromList (x : xs) = μ' (R' (×' (I' (L_1 (Const x)))
                               (I' (R_1 (fromList xs)))))

toList :: ⟦ List_C ⟧ (Const α) () → [α]
toList (μ' (L' U'))                                          = []
toList (μ' (R' (×' (I' (L_1 (Const x))) (I' (R_1 xs))))) =
  x : toList xs
```

Due to the presence of fixed points in the universe, the conversion functions are now necessarily directly recursive. It remains to see if this has a negative impact on performance, and if a shallow embedding for Indexed can be defined.

Rose trees Rose trees branch to an arbitrary number of subtrees at each level, using lists at the recursive position:

```
data Rose α = Fork α [Rose α]
```

We can encode rose trees in Indexed using composition and the code for lists defined earlier in Section 6.2.5:

```
type Rose_F = ((I (L ())) :×: (List_C :·: (I (R ()))))
                :: Code (Sum () ()) ())
type Rose_C = (μ Rose_F :: Code () ())
```

The rose tree functor $Rose_F$ defines rose trees as a product between an element and a composition of lists with recursive positions (more rose trees). We take the fixed point with $Rose_C$, defining a single type with one input parameter, again using the singleton type as the index.

To convert from a rose tree we need to recursively convert user lists into representation lists. We first use *fromList*, and then recursively map *fromRose* to the list parameters:

```
fromRose :: Rose α → ⟦ Rose_C ⟧ (Const α) ()
fromRose (Fork a as) =
  μ' (×' (I' (L_1 (Const a))) (C'
    (imap (I' ∘ R_1 ∘ fromRose ∘ unConst) (fromList as))))
```

The conversion in the opposite direction proceeds entirely symmetrically. We define an auxiliary function unR_1 to avoid explicit pattern matching while mapping:

```
toRose :: ⟦ Rose_C ⟧ (Const α) () → Rose α
toRose (μ' (×' (I' (L_1 (Const a))) (C' as))) =
  Fork a (toList (imap (Const ∘ toRose ∘ unR_1 ∘ un⟦⟧_1) as))
unR_1 :: Sum_1 α β (R ι) → β ι
unR_1 (R_1 x) = x
```

Abstract Syntax Tree So far we have only seen single datatypes, but Indexed, like Multirec, also supports families of datatypes. Consider the following family of datatypes encoding a simplified Abstract Syntax Tree (AST):

```
data Expr α = Var α
            | Let (Decl α) (Expr α)
data Decl α = Assign α (Expr α)
            | Seq [Decl α]
```

The two types defined, *Expr* and *Decl*, are mutually recursive, share a parameter α, and *Decl* makes use of lists. We use $()$ again as the input index, and we define a type AST_I with two elements to use as output index:

```
data AST_I = Expr_I | Decl_I
```

The encoding of $Expr_F$ is then straightforward:

```
type Expr_F = ((I (L ())) :+: ((I (R Decl_I)) :×: (I (R Expr_I))))
                :: Code (Sum () AST_I) AST_I)
```

For encoding $Decl_F$ we need to use reindexing. Recall the kind of composition:

$$(:\cdot:) :: \forall \nu_\kappa.\ Code\ \nu_\kappa\ o_\kappa \to Code\ \iota_\kappa\ \nu_\kappa \to Code\ \iota_\kappa\ o_\kappa$$

In the *Decl* case, $\iota_\kappa = ()$, and $o_\kappa = AST_I$. This implies that we need to reindex the output of $List_C$ from $()$ to AST_I. We thus create a new code $List_{\uparrow AST}$ as the reindexing of $List_C$ to the kind $Code\ ()\ AST_I$, and use the new code in the definition of the code for *Decl*:

```
type List_↑AST = (O_X List_↑AST_o List_C :: Code () AST_I)
type Decl_F = (((I (L ())) :×: (I (R Expr_I))) :+:
                (List_↑AST :·: (I (R Expr_I))))
                  :: Code (Sum () AST_I) AST_I)
```

The reindexing is performed by a type family:

```
type family List_↑AST_o :: AST_I → ()
```

Unfortunately we cannot give any meaningful instances for this type family. To be able to match on the indices on the left we would need to declare it as follows:

```
type family List_↑AST_o (o :: AST_I) :: ()
```

With this definition, however, we cannot use it as argument to O_X, because GHC requires type family applications to be fully saturated. Although GHC's core language does support unsaturated type families (Weirich et al. 2011), there has been no attempt to lift this restriction in source Haskell code. One could think that (in this case in particular) we only need some form of operator K of type $(\alpha :: \kappa_1) \to (\beta :: \kappa_2) \to (\alpha :: \kappa_1)$, but we can only define such an operator when $\kappa_1 = \star$. In that case we can define the operator as a datatype. However, in the general case κ_1 is not necessarily $\star$, and as such we would need to define K as a type synonym, in which case we run into the restriction that type synonyms need to be fully applied.

Shortcoming 7. *Promotion of function types is only fully useful in the presence of unsaturated type family applications, but these are currently not allowed.*

6.2.6 Another look at reindexing

Given that we cannot really use reindexing as we have defined it in the universe, we turn to exploring alternative definitions that do not use functions. We could, for instance, try a more direct encoding of the reindexing by means of a list of tuples:

```
data Code ι o = ...
              | ∀ι'. I_X [(ι',ι)] (Code ι' o)
              | ∀o'. O_X [(o,o')] (Code ι o')
```

In this encoding we would hope to state the reindexing by explicitly listing the mapping between the indices. Lists and tuples get promoted to kinds, so I_X and O_X can be promoted. However, in the interpretation function we would need to lookup the reindexing in the map. For this we would need type functions:

```
type family Lookup (α :: [(κ,ι_κ)]) (ι :: κ) :: ι_κ
```

Unfortunately we cannot define *Lookup*; since type families are not allowed any overlapping, the following code is not accepted by GHC:

type instance $Lookup\ ((\alpha,\tau) : \sigma)\ \alpha = \tau$
type instance $Lookup\ ((\alpha,\tau) : \sigma)\ \beta = Lookup\ \sigma\ \beta$

Pattern-matching axioms[10] have been proposed to allow a limited form of overlapping in type families. Were these to be implemented, we could try to implement reindexing as an explicit relation between indices.

We are then left without a way to do reindexing in this approach. This limits the usefulness of Indexed, since to encode *Decl* we would need to re-encode lists at a different kind. Even if automated, this sort of duplication is clearly undesirable.

7. Future work and conclusion

After exploring the potential of the improved kind language for improving different generic programming libraries, we now turn our attention to the shortcomings encountered, and what future work can be done to address them.

Type-level literals We plan to release new versions of the Regular and Multirec libraries as described in this paper; we will do so whenever type-level integers and strings are available in GHC, so that we have a proper way to encode datatype meta-information.

Improving SYB While Typeable can be easily improved with the use of kind polymorphism, we have not been able to come up with a simplification for SYB's *gcast*/*dataCast*$_1$/*dataCast*$_2$ duplication. More research is necessary to identify how to solve this problem.

Arity-generic functions Similarly to SYB, there is duplication on standard functions such as *liftM*/*liftM*$_2$ and *zipWith*/*zipWith*$_3$. Sheard (2006) studied this problem in Ωmega, a language with support for singleton types for dependent type emulation. Weirich and Casinghino (2010) name these "arity-generic" functions, and give unified definitions for them in Agda. It remains to be seen if these can now be encoded in Haskell.

Reindexing We have run into trouble with reindexing in our Haskell encoding of Indexed (Section 6.2). We plan to continue exploring alternative ways to encode this functionality, or necessary changes to the language to support our encoding.

Performance We have not yet analysed the performance of the rewritten libraries in the style of Magalhães et al. (2010b). We have good hopes that runtime performance does not deteriorate simply by the use of promotion, but we plan to confirm this by benchmarking. Furthermore, it would be interesting to see if the new kinds can be used for optimisation purposes.

Kind polymorphism and user-defined kinds have a tremendous potential for improving existing code, especially in generic programming libraries. Furthermore, the new extensions also enable developing programs that were previously impossible or uncanny to express. With new features come new limitations, but the promotion mechanism is in its early stages, and is likely to be improved to reflect programmers' needs and desires. We see these changes as an exciting new direction for Haskell, and look forward to future programs with less duplication, more expressive types, and stronger type-level guarantees.

Acknowledgments

This work has been funded by EPSRC grant number EP/J010995/1. Andres Löh has previously developed an encoding of Indexed in Haskell (without user kinds), and provided insight for the developments in this paper. Steven Keuchel also contributed to the definition of example encodings in Indexed. Richard A. Eisenberg, Thomas Harper, Thomas van Noort, and the anonymous reviewers provided valuable feedback on a draft version of this paper.

[10] `http://hackage.haskell.org/trac/ghc/wiki/NewAxioms`

References

Richard Bird and Lambert Meertens. Nested datatypes. In Johan Jeuring, editor, *Mathematics of Program Construction: 4th International Conference*, volume 1422 of *Lecture Notes in Computer Science*, pages 52–67. Springer, 1998.

Manuel M. T. Chakravarty, Gabriel C. Ditu, and Roman Leshchinskiy. Instant generics: Fast and easy, 2009. Available at `http://www.cse.unsw.edu.au/~chak/papers/CDL09.html`.

Richard A. Eisenberg and Stephanie Weirich. Dependently typed programming with singletons, 2012. To appear in Proceedings of the 2012 ACM SIGPLAN Haskell Symposium.

Jeremy Gibbons. Datatype-generic programming. In Roland Backhouse, Jeremy Gibbons, Ralf Hinze, and Johan Jeuring, editors, *Spring School on Datatype-Generic Programming*, volume 4719 of *Lecture Notes in Computer Science*. Springer-Verlag, 2007. doi:10.1007/978-3-540-76786-2.

Erik Hesselink. Generic programming with fixed points for parametrized datatypes. Master's thesis, Universiteit Utrecht, 2009. `https://github.com/hesselink/thesis/`.

Patrik Jansson and Johan Jeuring. PolyP—a polytypic programming language extension. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 470–482. ACM, 1997. doi:10.1145/263699.263763.

Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. In *Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, pages 26–37. ACM, 2003. doi:10.1145/604174.604179.

Ralf Lämmel and Simon Peyton Jones. Scrap more boilerplate: reflection, zips, and generalised casts. In *Proceedings of the 9th ACM SIGPLAN International Conference on Functional Programming*, pages 244–255. ACM, 2004. doi:10.1145/1016850.1016883.

Andres Löh. *Exploring Generic Haskell*. PhD thesis, Universiteit Utrecht, 2004. `http://igitur-archive.library.uu.nl/dissertations/2004-1130-111344`.

Andres Löh and José Pedro Magalhães. Generic programming with indexed functors. In *Proceedings of the 7th ACM SIGPLAN Workshop on Generic Programming*, pages 1–12. ACM, 2011. doi:10.1145/2036918.2036920.

José Pedro Magalhães, Atze Dijkstra, Johan Jeuring, and Andres Löh. A generic deriving mechanism for Haskell. In *Proceedings of the 3rd ACM Haskell Symposium on Haskell*, pages 37–48. ACM, 2010a. doi:10.1145/1863523.1863529.

José Pedro Magalhães, Stefan Holdermans, Johan Jeuring, and Andres Löh. Optimizing generics is easy! In *Proceedings of the 2010 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, pages 33–42. ACM, 2010b. doi:10.1145/1706356.1706366.

Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In John Hughes, editor, *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 124–144. Springer, 1991. doi:10.1007/3540543961_7.

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

Thomas van Noort, Alexey Rodriguez Yakushev, Stefan Holdermans, Johan Jeuring, and Bastiaan Heeren. A lightweight approach to datatype-generic rewriting. In *Proceedings of the ACM SIGPLAN Workshop on Generic Programming*, pages 13–24. ACM, 2008. doi:10.1145/1411318.1411321.

Thomas van van Noort, Alexey Rodriguez Yakushev, Stefan Holdermans, Johan Jeuring, Bastiaan Heeren, and José Pedro Magalhães. A lightweight approach to datatype-generic rewriting. *Journal of Functional Programming*, 20(Special Issue 3–4):375–413, 2010. doi:10.1017/S0956796810000183.

Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, September 2007.

Simon Peyton Jones, editor. *Haskell 98, Language and Libraries. The Revised Report*. Cambridge University Press, 2003. doi:10.1017/S0956796803000315. Journal of Functional Programming Special Issue 13(1).

Alexey Rodriguez Yakushev, Johan Jeuring, Patrik Jansson, Alex Gerdes, Oleg Kiselyov, and Bruno C.d.S. Oliveira. Comparing libraries for generic programming in Haskell. In *Proceedings of the 1st ACM SIGPLAN Symposium on Haskell*, pages 111–122. ACM, 2008. doi:10.1145/1411286.1411301.

Alexey Rodriguez Yakushev, Stefan Holdermans, Andres Löh, and Johan Jeuring. Generic programming with fixed points for mutually recursive datatypes. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, pages 233–244. ACM, 2009. doi:10.1145/1596550.1596585.

Tom Schrijvers, Simon Peyton Jones, Martin Sulzmann, and Dimitrios Vytiniotis. Complete and decidable type inference for GADTs. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, pages 341–352. ACM, 2009. doi:10.1145/1596550.1596599.

Tim Sheard. Generic programming programming in Ωmega. In Roland Backhouse, Jeremy Gibbons, Ralf Hinze, and Johan Jeuring, editors, *Datatype-Generic Programming*, volume 4719 of *Lecture Notes in Computer Science*, pages 258–284. Springer, 2006. doi:10.1007/978-3-540-76786-2_5.

Tim Sheard and Simon Peyton Jones. Template meta-programming for Haskell. *ACM SIGPLAN Notices*, 37:60–75, December 2002. doi:10.1145/636517.636528.

Martin Sulzmann, Manuel M. T. Chakravarty, Simon Peyton Jones, and Kevin Donnelly. System F with type equality coercions. In *Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, pages 53–66. ACM, 2007. doi:10.1145/1190315.1190324.

Stephanie Weirich and Chris Casinghino. Arity-generic datatype-generic programming. In *Proceedings of the 4th ACM SIGPLAN Workshop on Programming Languages meets Program Verification*, pages 15–26. ACM, 2010. doi:10.1145/1707790.1707799.

Stephanie Weirich, Dimitrios Vytiniotis, Simon Peyton Jones, and Steve Zdancewic. Generative type abstraction and type-level computation. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Panguages*, pages 227–240. ACM, 2011. doi:10.1145/1926385.1926411.

Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. Giving Haskell a promotion. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, pages 53–66. ACM, 2012. doi:10.1145/2103786.2103795.