Classes of Arbitrary Kind*

Alejandro Serrano and Victor Cacciari Miraldo

Department of Information and Computing Sciences, Utrecht University
{A.SerranoMena, V.CacciariMiraldo}@uu.nl

Abstract. The type class system in the Haskell Programming language provides a useful abstraction for a wide range of types, such as those that support comparison, serialization, ordering, between others. This system can be extended by the programmer by providing custom instances to one's custom types. Yet, this is often a monotonous task. Some notions, such as equality, are very regular regardless if it is being encoded for a ground type or a type constructor. In this paper we present a technique that unifies the treatment of ground types and type constructors whenever possible. This reduces code duplication and improves consistency. We discuss the encoding of several classes in this form, including the generic programming facility in GHC.

Keywords: Haskell · Type classes · Generic programming.

1 Introduction

Type classes [16] are a widely used abstraction provided by the Haskell programming language. In their simplest incarnation, a type class defines a set of *methods* which every *instance* must implement, for instance:

```
class Eq a where
  (≡) :: a → a → Bool
```

Where we are stating that a type a can be an instance of Eq as long as it implements $(\equiv)$. This is a very useful mechanism, as it allows a programmer to write polymorphic functions but impose some restrictions on the types. For instance, consider the type of the *nub* function below:

```
nub :: (Eq a) ⇒ [a] → [a]
```

It receives a list of arbitrary a's, as long we can compare these values for equality. It returns a list of the same type, but removes every duplicate element.

The base library comes pre-packaged with instances for built-in types, such as integers, Booleans and lists. But the programmer is also allowed to extend this set of instances. For example, here is a data type representing binary trees with values in the internal nodes:

```
data Tree a = Leaf | Node (Tree a) a (Tree a)
```

Its corresponding **instance** for the Eq class is defined as:

* Supported by NWO project 612.001.401.

```
instance Eq a ⇒ Eq (Tree a) where
  Leaf           ≡ Leaf           = True
  (Node ℓ1 x1 r1) ≡ (Node ℓ2 x2 r2) = x1 ≡ x2 ∧ ℓ1 ≡ ℓ2 ∧ r1 ≡ r2
  _              ≡ _              = False
```

This example highlights one of the important features of the type class system: the availability for an instance of *Tree* a, in this case, depends on the availability of an instance for the type a. In other words, in order to compare a *Tree* a for equality, we must also be able to compare values of type a, since these are the elements in the nodes of our tree. That way, we know that equality is defined for *Tree Bool* — since *Bool* is a member of the *Eq* type class — but not for *Tree* (*Int* → *Bool*) — since functions cannot be compared for equality.

Our *Tree* data type is also an instance of the *Functor* class, which states that we can change a *Tree* a into a *Tree* b as long as we can change one a into one b. We do so by applying this transformation uniformly in the nodes of the tree. In Haskell, we would say:

```
class Functor f where
  fmap :: (a → b) → f a → f b
instance Functor Tree where
  fmap _ Leaf           = Leaf
  fmap f (Node ℓ x r) = Node (fmap f ℓ) (f x) (fmap f r)
```

There is a subtle difference between *Eq* and *Functor* that is central to our work. The *Eq* class receives ground types, such as *Int* and *Tree Bool*; whereas the *Functor* receives type constructors, such as *Tree*. In Haskell we distinguish between those by the means of *kinds*. The intuition is that a kind is the "type of types". By convention, ground types have kind $\star$. Therefore, our *Tree* type is of kind $\star \to \star$, that is, given a ground type as an argument, it produces a ground type. The *kind system* prevents us from writing nonsensical statements such as *Functor Int*, since *Functor* expects something of kind $\star \to \star$ and *Int* has kind $\star$.

There are several extensions to the type class mechanism, such as functional dependencies [7], and associated types [4]. For the purpose of this paper, though, we shall consider only the simpler version described above, with the occasional appearance of multi-parameter type classes [11].

1.1 Concepts of arbitrary kind

We can declare that lists support equality provided that their elements do so:

```
instance (Eq a) ⇒ Eq [a] where ...
```

However, lists support an even stronger notion of comparison: if we provide a comparison function between elements of types a and b, we can uniformly lift this operation to act over $[a]$ and $[b]$. This concept is captured by the *Eq1* type class, which can be found in GHC's base library:

```
class Eq1 (f :: ⋆ → ⋆) where
  liftEq :: (a → b → Bool) → f a → f b → Bool
```

Following the same line of thought, we might be inclined to derive a similar notion for the *Either a b* type. This is a type constructor with *two* arguments that can be compared in a similar way, but this time we would need two comparison functions in order to compare a *Either a b* with a *Either c d*. Alas, we need to introduce yet another type class, since *Eq1* only works for types of kind $\star \to \star$:

```
class Eq2 (f :: ⋆ → ⋆ → ⋆) where
  liftEq2 :: (a → b → Bool) → (c → d → Bool)
          → f a c → f b d → Bool

instance Eq2 Either where
  liftEq2 p _ (Left  x) (Left  y) = p x y
  liftEq2 _ q (Right x) (Right y) = q x y
  liftEq2 _ _ _         _         = False
```

The notion of lifting equality makes sense for arbitrary kinds: *Eq*, *Eq1*, *Eq2*, and so on. Where *Eq* is seen as the *Eq0* member of this sequence, where we do not take any equality function as argument because the type in question has no type parameters.

We can witness the same pattern for the *Monoid* and *Alternative* type classes, provided in the base library:

```
class Monoid (m :: ⋆) where
  mempty  :: a
  mappend :: a → a → a

class Alternative (f :: ⋆ → ⋆) where
  empty :: f a
  (⟨|⟩)  :: f a → f a → f a
```

A monoid is a type embodied with a way to combine two elements, *mappend*, which satisfies associativity and for which *mempty* acts as neutral element. An instance of *Alternative* for a type constructor *C* essentially states that *C a* can be used as a monoid, regardless of the element *a* contained in the structure.

The relationship between *Monoid* and *Alternative* is different from that between *Eq* and *Eq1*: in the latter case we lift a growing sequence of functions, whereas in the former the type has the same shape regardless of the kind at play. Nevertheless, the programmer has to write a new class for every different kind, even though this might be a very regular notion.

1.2 Contributions

The main contribution of this paper is a pattern to define "once and for all" type classes to encompass notions such as *Eq* and *Monoid* which extend to arbitrary kinds. We borrow a technique from recent developmens in generic programming [14], enabling us to represent type applications uniformly (Section 2) and showcase its usage in the definition of type classes (Section 3). As it turns out, the ability of handling type applications on the term level has far more use than solely generic programming.

We also discuss an extension of the generic programming mechanism in GHC [8] to represent types of arbitrary kinds (Section 4). Our approach can be seen as a translation from the techniques introduced by Serrano and Miraldo [14] to the world of pattern functors.

2 Representing Type Application

The core issue is the inability to represent a type that is applied to n type variables. For the *Eq* case, we ultimately want write something in the lines of:

$$\begin{aligned}&\textbf{class } \mathit{Eq}^n\ (f :: \star \to \ldots \to \star) \textbf{ where}\\ &\quad \mathit{liftEq}^n\ :: (a_1 \to b_1 \to \mathit{Bool}) \to \ldots \to (a_n \to b_n \to \mathit{Bool})\\ &\qquad\quad \to f\ a_1\ \ldots\ a_n \to f\ b_1\ \ldots\ b_n \to \mathit{Bool}\end{aligned}$$

Yet, simple Haskell without any extension does not allow us to talk about a type variable f applied to "as many arguments as it needs". In fact, we require some of the later developments on the Haskell language to be able to uniformly talk about types of arbitrary kinds. These developments include data type promotion [21] and partial support for dependent types [18].

The key idea is to split a type application such as $f\ a\ b\ c$ in two parts: the *head* f, and the *list of types* $\langle a, b, c\rangle$. Afterwards, we define an operator $(:@@:)$ which applies a list of types to a head. For example, we should have:

$$f :@@: \langle a, b, c\rangle \approx f\, a\, b\, c$$

where the $\approx$ denotes isomorphism.

Naturally, we want to rule our incorrect applications of the $(:@@:)$ operator. For example, $\mathit{Int} :@@: \langle \mathit{Int}\rangle$. should be flagged as wrong, since *Int* is not a type constructor, and thus cannot take arguments. In a similar fashion, $\mathit{Tree} :@@: \langle \mathit{Tree}\rangle$ should not be allowed, because *Tree* needs a ground type as argument. The required information to know whether a list of types can be applied to a head must be derived from the *kind* of the head. The solution is to add an index to the type Γ that keeps track of the *kind* of such environment. The definition is written as a Generalized Algebraic Data Type, which means that we give the type of each constructor explicitly:

$$\begin{aligned}&\textbf{infixr } 5 :\&:\\ &\textbf{data } \Gamma\ k \textbf{ where}\\ &\quad \epsilon \quad :: \quad \Gamma\ (\star)\\ &\quad (:\&:) :: k \to \Gamma\ ks \to \Gamma\ (k \to ks)\end{aligned}$$

An empty list of types is represented by ϵ. If we apply such a list of types with a goal of getting a ground type, this implies that the kind we started with was already $\star$, as reflected in the index of this constructor. The other possibility is to attach a type of kind k to a list of types with kind ks, represented by the constructor $(:\&:)$. Here are some examples of lists of types with different indices:

$$\begin{aligned}&\mathit{Int} :\&: \epsilon && :: \Gamma\ (\star \to \star)\\ &\mathit{Int} :\&: \mathit{Bool} :\&: \epsilon && :: \Gamma\ (\star \to \star \to \star)\\ &\mathit{Tree} :\&: \mathit{Bool} :\&: \epsilon && :: \Gamma\ ((\star \to \star) \to \star \to \star)\end{aligned}$$

The next step is the definition of the $(:@@:)$ operator.

$$\begin{aligned}&\textbf{data } (f :: k) :@@: (\mathit{tys} :: \Gamma\ k) :: \star \textbf{ where}\\ &\quad \mathit{A0}\ :: f \to f :@@: \epsilon\\ &\quad \mathit{Arg} :: f\ t :@@: \mathit{ts} \to f :@@: (t :\&: \mathit{ts})\end{aligned}$$

We abstain from using type families [13] since defining functions whose arguments are applied families is hard in practice. For instance, if we had defined $(:@@:)$ as follows:

$$\begin{aligned}&\textbf{type family}\ (f :: k) :@@:_{\mathsf{fam}} (tys :: \Gamma\ k) :: \star\ \textbf{where}\\ &\quad f :@@:_{\mathsf{fam}} \epsilon \qquad\qquad = f\\ &\quad f :@@:_{\mathsf{fam}} (a :\&: as) = (f\ a) :@@:_{\mathsf{fam}} as\end{aligned}$$

Then writing a simple function that is polymorphic on the *number*, such as:

$$\begin{aligned}&g :: f :@@:_{\mathsf{fam}} tys \to String\\ &g\ \_ = \text{“}Hello,\ PADLers!\text{”}\end{aligned}$$

Would be rejected by the compiler with the following error message:

```
Couldn't match type 'f :@: tys' with 'f0 :@: tys0'
Expected type: f :@: tys -> String
Actual type: f0 :@: tys0 -> String
NB: '(:@:)' is a non-injective type family
The type variables 'f0', 'tys0' are ambiguous
```

The problem is that type families are not necessarily injective. That is, the result of $f :@@:_{\mathsf{fam}} tys$ is not sufficient to fix the values of the type variables f and tys in the type of g. This is not a problem with the type checking algorithm; these three different choices of f and tys are all equal to the same type:

$$\begin{aligned}&Either :@@:_{\mathsf{fam}} (Int :\&: Bool :\&: \epsilon) \equiv Either\ Int\ Bool\\ &Either\ Int :@@:_{\mathsf{fam}} (Bool :\&: \epsilon) \quad\ \equiv Either\ Int\ Bool\\ &Either\ Int\ Bool :@@:_{\mathsf{fam}} \epsilon \qquad\ \equiv Either\ Int\ Bool\end{aligned}$$

Hence, we stick with GADTs, which allows us to write the g function above. However, when we *call* the function we need to wrap the argument with constructor $A0$ and Arg. The amount of Arg constructors expresses how many of the arguments go into the list of types. For example:

$$\begin{aligned}&g\ (A0\ (Left\ 3)) \qquad\qquad\ \rightsquigarrow f = Either\ Int\ Bool, tys = \epsilon\\ &g\ (Arg\ (Arg\ (A0\ (Left\ 3)))) \rightsquigarrow f = Either, tys = Int :\&: Bool :\&: \epsilon\end{aligned}$$

This need to be explicit about the amount of type variables is definitely cumbersome. In Section 3.2 we define a type class which allows us to convert easily between uses of $(:@@:)$ and $(:@@:_{\mathsf{fam}})$.

Better pattern matching. A related problem with the usage of $(:@@:)$ is the need of nested Arg constructors, both in building values and pattern matching over them. Fortunately, we can reduce the number of characters by using *pattern synonyms* [12].

For the purposes of this paper, it is enough to provide synonyms from nested sequences of Arg up to length 2:

$$\begin{aligned}&\textbf{pattern}\ A1\ x = Arg\ (A0\ x)\\ &\textbf{pattern}\ A2\ x = Arg\ (A1\ x)\end{aligned}$$

This means that we could have written the latest example using g as simply $g\ (A2\ (Left\ 3))$.

3 Arbitrary-kind Type Classes

With the necessary tools at hand, we are ready to discuss how to define type classes for notions which exist in arbitrary kinds. Being able to access the type application structure on the term level, with $(:@@:)$, is essential. In what follows, we look at this construction by a series of increasingly complex type classes which generalize the well-known *Show*, *Eq*, and *Functor*.

Generalizing Show. In its bare bones version, *Show* specifies how to turn a value into a string. One could define a simplistic[1] version of *Show* as:

```
class Show t where
  show :: t → String
```

To specify that we want *Show* to work on arbitrary kinds, we need to add a kind signature to its definition, and change the type of the *show* method to apply a list of types:[2]

```
class Show◇ (f :: k) where
  show◇ :: f :@@: tys → String
```

We can define an instance for integers by piggybacking on the usual version of *Show* provided in Haskell's base library:

```
instance Show◇ Int where
  show◇ (A0 n) = show n
```

Note that since we are dealing with values of the data type $(:@@:)$, we need to pattern match on *A0* to obtain the integer value itself.

If we try to write similar code for the *Maybe* type constructor, we will bump into an error. Consider the following *Show◇ Maybe* instance:

```
instance Show◇ Maybe where
  show◇ (Arg (A0 Nothing)) = "Nothing"
  show◇ (Arg (A0 (Just x))) = "Just (" ++ show◇ (A0 x) ++ ")"
```

The compiler complains with:

```
Could not deduce (Show◇ t) arising from a use of 'show◇'
```

The problem is in the call *show◇ (A0 x)* and stems from the fact that we have not provided any proof that the *contents* inside the *Just* constructor can be "shown". Let us recall the *Show* instance for *Maybe a*:

```
instance Show a ⇒ Show (Maybe a) where ...
```

Note how this instance *requires* a proof that *a* is also an instance of *Show*. We therefore need a mechanism to specify that the arguments of *Maybe* $:@@:$ *tys*, that is, *tys*, can be shown. That is, we need to specify some *constraint* over *tys*. This will allow us to call *show◇* with a list of types *tys* whenever every

[1] The actual Haskell definition contains functions to deal with operator precedence and efficient construction of *String*s.

[2] We use the notation $f^{\diamond}$ to refer to the generalized version of f.

element of this list is also member of $Show^\diamond$. In order to define this constraint we require *ConstraintKinds* [1] GHC extension. In summary, this extension enables the manipulation of everything that appears before the $\Rightarrow$ arrow in a Haskell type as if it was a regular type. The only difference is in the kind: normal arguments to functions must be of a type of kind $\star$, implicit arguments, to the left of the $\Rightarrow$, must be of kind *Constraint*.

We will now define a constraint that depends on the types *tys* that are applied to a type constructor with $f :@@: tys$. This definition will proceed by induction on the structure of *tys*. For the specific $Show^\diamond$ case we define the following $AllShow^\diamond$ constraint. Note here that the syntax for an empty constraint is $()$ in GHC, and the conjunction of constraints is represented by tupling:

```
type family AllShow◇ (tys :: Γ k) :: Constraint where
  AllShow◇ ϵ          = ()
  AllShow◇ (t :&: ts) = (Show◇ t, AllShow◇ ts)
```

Finally we introduce this constraint in the type of $show^\diamond$:

```
class Show◇ (f :: k) where
  show◇ :: AllShow◇ tys ⇒ f :@@: tys → String
```

The $Show^\diamond$ instance for *Maybe* shown above is now accepted, since $AllShow^\diamond$ applied to a list of types of the form $t :\&: \epsilon$ reduces to a constraint $Show^\diamond\ t$.

Generalizing Eq. The generalized equality class is slightly more complicated; the number of arguments to the *liftEq* function changes between classes:

```
(≡)     ::                                          a      → b      → Bool
liftEq  :: (a → b → Bool)                        → f a    → f b    → Bool
liftEq2 :: (a → b → Bool) → (c → d → Bool)       → f a c  → f b d  → Bool
```

Hence, the number of arguments in this case is dictated by the kind of the type constructor f: one per appearance of $\star$. We will use a similar technique to $Show^\diamond$, and define a data type by induction on the structure of the type applications. This new data type, *Predicates*, will require a function $a \to b \to Bool$ for every pair of types obtained from the corresponding lists:

```
data Predicates (as :: Γ k) (bs :: Γ k) where
  Pϵ  :: Predicates ϵ ϵ
  P&  :: (a → b → Bool) → Predicates as bs
      → Predicates (a :&: as) (b :&: bs)
```

Using this data type we can chain as many predicates as we need:

```
P& f Pϵ          :: Predicates (a :&: ϵ)          (b :&: ϵ)
P& f (P& g Pϵ) :: Predicates (a :&: c :&: ϵ) (b :&: d :&: ϵ)
```

The final step to generalize the *Eq* notion is to use the explicit application operator $(:@@:)$ in the definition of the type class. The result in this case is:

```
class Eq◇ (f :: k) where
  eq◇ :: Predicates as bs → f :@@: as → f :@@: bs → Bool
```

Here are the instances for integers and the instance for *Either*:

```
instance Eq◇ Int where
  eq◇ Pϵ (A0 x) (A0 y) = x ≡ y

instance Eq◇ Either where
  eq◇ (P& l (P& r Pϵ)) (A2 (Left x))  (A2 (Left y))  = l x y
  eq◇ (P& l (P& r Pϵ)) (A2 (Right x)) (A2 (Right y)) = r x y
  eq◇ _                _              _              = False
```

3.1 You-Name-It-Functors

As the final example of our approach, we are going to generalize several notions of functoriality, including those present in GHC's base library. With access to the structure of type applications available through (:@:), we are able to unify the *Functor*, *Contravariant*, *Profunctor* and *Bifunctor* classes with many others. Recall the *Functor* type class, which describes how to lift a function into a container f:

```
class Functor f where
  fmap :: (a → b) → f a → f b
```

Many types, such as lists, *Maybe*, or binary trees, are examples of *Functor*s. But now take the following data type, which describes a logical predicate:

```
data Pred a = Pred (a → Bool)
```

It is not possible to write a *Functor* instance for this type. However, we can write one for *Contravariant*, a variation of *Functor* in which the function being lifted goes "in the opposite direction":

```
class Contravariant f where
  contramap :: (b → a) → f a → f b
instance Contravariant Pred where
  contramap f (Pred p) = Pred (p ∘ f)
```

These notions generalize to higher kinds. For example, the type *Either* a b behaves as a functor for both a and b. That is, if you give a function mapping a to c, if can be lifted to a function between *Either* a b to *Either* c b; and the same holds for the other type variable. We say that *Either* is a *bifunctor*. The type of functions, $a \to b$, is slightly trickier, because it works as a contravariant functor in the source type, and as a usual functor in the target type. The name for this kind of structure is a *profunctor*. Both structures exist in the base libraries:

```
class Bifunctor f where
  bimap :: (a → b) → (c → d) → f a c → f b d
class Profunctor f where
  dimap :: (b → a) → (c → d) → f a c → f b d
```

With our techniques, we can develop a *unified* type class for all of these notions. The recipe is analogous: we need to specify some information for each

type argument *tys* in $f :@@: tys$. Here we need to know whether each position maps in the "same" or "opposite" way – we call this the *variance* of that position. This is essentially a list of flags whose length must coincide with the number of type arguments to the type. We copy the approach of *Γ* and *Predicates*, and define a new data type indexed by a kind:

```
data Variance k where
  Vϵ ::                 Variance (⋆)
  ⇛  :: Variance ks → Variance (⋆ → ks)
  ⇚  :: Variance ks → Variance (⋆ → ks)
  ↷  :: Variance ks → Variance (k → ks)
```

For the sake of generality, we have also included a $\curvearrowright$ flag which states that a type variable is not used in any constructor of the data type, and thus we can skip it in the list of functions to be lifted. Using this *Variance* type, we can express the different ways in which *Functor*, *Contravariant*, and *Bifunctor* operate on their type variables as $\Rrightarrow V\epsilon$, $\Lleftarrow V\epsilon$, and $\Rrightarrow (\Rrightarrow V\epsilon)$, respectively.

In contrast to the previous examples, to obtain the generalized version of the *Functor* we need an extra parameter: we must attach the corresponding *Variance*. Given a type constructor its variance is fixed, therefore, it can be uniquely determined. This is represented by a *functional dependency* [7]:

```
class Functor◇ (f :: k) (v :: Variance k) | f → v where ...
```

The next step is defining lists of functions that we will need in order to map a $f :@@: tys$ into a $f :@@: txs$. This is done in the same fashion as in the previous examples: by induction on the structure of *tys* and *txs*. This time, however, in addition to the list of types for the source and the target, we also use the *Variance* to know which is the direction in which the function should operate:

```
data Mappings (v :: Variance k) (as :: Γ k) (bs :: Γ k) where
  Mϵ  ::                Mappings Vϵ     ϵ          ϵ
  M⇛ :: (a → b) → Mappings     vs         as         bs
                → Mappings (⇛ vs) (a :&: as) (b :&: bs)
  M⇚ :: (b → a) → Mappings     vs         as         bs
                → Mappings (⇚ vs) (a :&: as) (b :&: bs)
  M↷ ::           Mappings     vs         as         bs
                → Mappings (↷ vs) (a :&: as) (b :&: bs)
```

We are now ready to write the definition of *Functor*$^\diamond$, by using a list of mappings as an argument to the generalize *fmap*$^\diamond$ function:

```
class Functor◇ (f :: k) (v :: Variance k) | f → v where
  fmap◇ :: Mappings v as bs → f :@@: as → f :@@: bs
```

What follows are two *Functor*$^\diamond$ instances for types of different kinds, namely, one for *Pred* and one for *Either*.

```
instance Functor◇ Pred (⇚ Vϵ) where
  fmap◇ (M⇚ f Mϵ) (A1 (Pred p)) = A1 (Pred (p ∘ f))
```

```
instance Functor◇ Either (⇛ (⇛ Vϵ)) where
  fmap◇ (M⇛ f (M⇛ _ Mϵ)) (A2 (Left x))  = A2 (Left (f x))
  fmap◇ (M⇛ _ (M⇛ g Mϵ)) (A2 (Right y)) = A2 (Right (g y))
```

Note how the $Functor^{\diamond}\ Pred\ (\Leftarrow\ V\epsilon)$ instance is equivalent to *Contravariant Pred* and similarly $Functor^{\diamond}\ Either\ (\Rightarrow\ (\Rightarrow\ V\epsilon))$ is equivalent to *Bifunctor Either*. In fact, as a final touch, we can regain the old type classes by giving names to certain instantiations of the *Variance* parameter in $Functor^{\diamond}$:

```
type Functor     f = Functor◇ f (⇛ Vϵ)
type Contravariant f = Functor◇ f (⇚ Vϵ)
type Bifunctor   f = Functor◇ f (⇛ (⇛ Vϵ))
type Profunctor  f = Functor◇ f (⇚ (⇛ Vϵ))
```

3.2 From Families to Data, and Back Again

The two ways of defining type application, $(:@@:_{\mathsf{fam}})$ and $(:@@:)$, have dual advantages. In the case of the type family, the caller of the function does not have to wrap the arguments manually, but the compiler rejects the function if no further information is given. When using a data type, the compiler accepts defining functions without further problem, at the expense for the caller having to introduce *A0* and *Arg* constructors.

The distinction does not have to be that sharp, though. It is simple to write a type class *Ravel* which converts between the data type encoding and the type family encoding:

```
class Ravel (t :: ⋆) (f :: k) (tys :: Γ k) | f tys → t where
  unravel :: f :@@: tys → f :@@:fam tys
  ravel   :: f :@@:fam tys → f :@@: tys
instance Ravel t t ϵ where
  unravel (A0 x) =     x
  ravel   x      = A0 x
instance Ravel t (f x) tys ⇒ Ravel t f (x :&: tys) where
  unravel (Arg x) = unravel x
  ravel   x       = Arg (ravel x)
```

This way, we can pattern match in elements using $(:@@:)$ when defining function – and thus working around the problems of ambiguity discussed above – but at the same time provide an external interface with simpler types by means of the *unravel* and *ravel* conversion functions.

An inmediate application of this type class is the definition of specialized versions of $fmap^{\diamond}$ for the different variances with a name. Here is the *dimap* function from the *Profunctor* type class:

```
dimap :: Profunctor f ⇒ (b → a) → (c → d) → f a c → f b d
dimap f g = unravel ∘ fmap◇ (M⇚ f (M⇛ g Mϵ)) ∘ ravel
```

We first map the given value of type $f\ a\ c$ into $f :@@: (a :\&: c :\&: \epsilon)$. At that point we can apply the generic operation $fmap^{\diamond}$, we need to wrap the operations

f and g into the *Mappings* type. We get a result of type $f :@@: (b :\&: d :\&: \epsilon)$, which we map back to $f\ b\ d$ by means of *unravel*.

4 Generics for Arbitrary Kinds

As a final note on the applicability of $(:@@:)$, we would like to show how it immediatly helps in providing support for more types in the already existing `GHC.Generics` framework. In this section, we provide a direct translation of the work of Serrano and Miraldo [14] reusing most of the machinery already available in GHC instead of relying on complicated type level constructions, as in the original work. These extensions are available as part of the `kind-generics` library in Hackage.

Haskell compilers provide facilities to automatically write instances for some common classes, including *Eq* and *Show*. This mechanism is syntactically lightweight, the programmer is just required to add a **deriving** clause at the end of a data type definition:

```
data Either a b = Left a | Right b deriving (Eq, Show)
data Pair   a b = Pair a b         deriving (Eq, Show)
```

GHC extends this facility with a mechanism to write functions which depend solely on the structure of the data type, but follow the same algorithm otherwise. This style is called *generic programming*.

The core idea of generic programming is simple: map data types to a uniform *representation*. Every function which operates on this representation is by construction generic over the data type. There are several ways to obtain the uniform representation in a typed setting; the main ones being codes [15, 9, 14], and pattern functors [10, 20, 8]. The latter is the one used by GHC, and the one we extend in this section.

In the pattern functor approach, the structure of a data type is expressed by a combination of the following functors, which act as building blocks:

```
data U1       a = U1
data K1 p     a = K1 a
data (f :*: g) a = f a :*: g a
data (f :+: g) a = L1 (f a) | R1 (g a)
```

Let us look at the representation of the aforementioned *Either* and *Pair*:

```
Rep (Either a b) = K1 R a :+: K1 R b
Rep (Pair   a b) = K1 R a :*: K1 R b
```

The *Either* type provides a choice between two constructors. This fact is represented by using the *coproduct* functor $(:+:)$. The *Pair* type, on the other hand, requires two pieces of information. The *product* functor $(:*:)$ encodes this infor-

mation. In both cases, each field is represented by the *constant* functor *K1 R*.[3] The remaining block, *U1*, represents a constructor with no fields attached, like the empty list [].

Besides the structure of a type, we also need a map between *values* in the original type and its representation. The *Generic* type class bridges this gap:

```
class Generic a where
  type Rep a :: ⋆ → ⋆
  from :: a → Rep a x
  to   :: Rep a x → a
```

To define an instance of *Generic*, you need three pieces of information: (1) the representation *Rep*, which is a functor composed of the building blocks described above, (2) how to turn a value of type a into its representation – this is the function *from* –, and (3) how to map back – encoded as *to*. Writing these instances is mechanical, and GHC automates them by providing a *DeriveGeneric* extension.

Due to space limitations, we gloss over how to define functions which work on the generic representations. The interested reader is referred to the work of Magalhães *et al.* [8], and the documentation of `GHC.Generics`.

The first step towards generalizing the *Generic* type class is to extend its building blocks for the representations. In the case of the original *Rep*, we encode a type of kind $\star$ by a functor of kind $\star \to \star$; now we are going to encode a type of kind k by a representation of kind $\Gamma\ k \to \star$. The use of a list of types Γ here is essential, because otherwise we cannot express arbitrary kinds. This approach deviates from the one taken by GHC, in which type constructors of kind $\star \to \star$ are also represented by functors of kind $\star \to \star$. In fact, all we need to do is define a new *K1* since the other combinators are readily compatible.

We solve this problem by introducing a separate data type to encode the structure of the type of a field [17, 14]. This is nothing more than the applicative fragment of λ-calculus, in which references to type variables are encoded using de Bruijn indices:

```
data TyVar d k where
  VZ ::                 TyVar (x → xs) x
  VS :: TyVar xs k → TyVar (x → xs) k
data Atom d k where
  Var  :: TyVar d k → Atom d k
  Kon  :: k         → Atom d k
  (:@:) :: Atom d (k1 → k2) → Atom d k1 → Atom d k2
```

A value of *Atom d k* represents a type of kind k in an environment with type variables described by the kind d. *TyVar* refers to a type variable in the kind d using Peano numerals, starting from the left-most variable. For example, the type of the single field of the *Right* constructor is represented as *Var* (*VS VZ*).

[3] The first argument to *K1* was used in the past to encode some properties of the field. However, it has fallen into disuse, and GHC always sets its value to *R* in any automatically-derived representation.

By itself, *Atom d k* only *describes* the shape of a type. In order to obtain an actual type, we need to *interpret* it with known values for all the type variables. If one thinks of *Atom*s as expressions with variables, such as $x^2 + x + 1$, interpreting the *Atom* boils down to obtaining the value of the expression given a value for each variable, like $x \mapsto 5$. This interpretation can be defined by recursion on the structure of the *Atom*:

```
type family Ty (t :: Atom d k) (tys :: Γ d) :: k where
  Ty (Var VZ)     (t :&: ts) = t
  Ty (Var (VS v)) (t :&: ts) = Ty (Var v) ts
  Ty (Kon x)      tys        = x
  Ty (f :@: x)    tys        = (Ty f tys) (Ty x tys)
```

Note that by construction an *Atom d k* is always interpreted to a type of kind *k*.

With these ingredients, we can encode the missing building block. In Haskell, it is required that fields in a data type have a type with kind $\star$. This is visible in the definition of *F*, where *t* must describe a type with that specific kind:

```
data F (t :: Atom d (⋆)) (x :: Γ d) = F (Ty t x)
```

The representations of *Either* and *Pair* look as follows:

```
Rep◇ Either = F (Var VZ) :+: F (Var (VS VZ))
Rep◇ Pair   = F (Var VZ) :*: F (Var (VS VZ))
```

Note the change in the arguments to *Rep*◇. We are no longer defining a family of representations for every possible choice of type arguments to *Either* and *Pair*, as we did in the case of *Rep*. Here we encode precisely the polymorphic structure of the data types.

Finally, we can wrap it all together with a type class that declares the isomorphism between values and their generic representation.

```
class Generic◇ (f :: k) where
  type Rep◇ f :: Γ k → ⋆
  from◇ ::          f :@@: x → Rep◇ f x
  to◇   :: Γs x → Rep◇ f x → f :@@: x
```

One part of the isomorphism, *from*◇, has a straightforward type. The converse operation is harder to define though. The problem is that we need to match over the structure of the list of types, but this information is apparent from the kind itself. The solution is to use a *singleton* [5], that is, to introduce a new data type that completely mimics the shape of the type level information:

```
data Γs (tys :: Γ k) where
  ϵs  ::          Γs ϵ
  &s :: Γs ts → Γs (t :&: ts)
```

By inspecting this singleton value, the compiler gains enough information about the shape of *tys* to know that the result is well-formed.

We can finally give the *Generic*$^\diamond$ instance for *Either*:

```
instance Generic◇ Either where
  type Rep Either = F (Var VZ) :+: F (Var (VS VZ))
  from◇ (A2 (Left x))  = InL (F x)
  from◇ (A2 (Right y)) = InR (F y)
  to◇ (&s (&s ϵs)) (InL (F x)) = A2 (Left x)
  to◇ (&s (&s ϵs)) (InR (F y)) = A2 (Right y)
```

4.1 Representing Constraints and Existentials

One advantage of this new representation is that it becomes simple to describe more complicated structures for data types. In particular, it enables us to represent constructors with constraints and existentials, key ingredients of Generalized Algebraic Data Types [19] available in Haskell and OCaml, among others.

The case of constraints is very similar to that of regular fields. As we have discussed several times already, a constraint is seen by GHC as a regular type of a specific kind *Constraint*. Since our language of types, *Atom*, works regardless of the kind it represents, we can still use it in this new scenario:

```
data (:⟹:) (c :: Atom d Constraint) (f :: Γ d → ⋆) (x :: Γ d) where
  C :: Ty c x ⇒ f x → (c :⟹: f) x
```

The definition of $(:\Longrightarrow:)$ wraps a representation f with an additional *implicit* parameter. This means that by merely pattern matching on the constructor we make this information available. For example, here is the usual definition of the *Equality* data type and its *Generic*$^\diamond$ instance:

```
data Equals a b where
  Refl :: a ∼ b ⇒ Equals a b
instance Generic◇ Equals where
  type Rep◇ Equals = (Kon (∼) :@: Var VZ :@: Var (VS VZ)) :⟹: U1
  from◇ (A2 Refl) = C U1
  to◇ (&s (&s ϵs)) (C U1) = A2 Refl
```

Introducing existentials requires more involved types, however.

```
data E k (f :: Γ (k → d) → ⋆) (x :: Γ d) where
  E :: forall k (t :: k) d (f :: Γ (k → d) → ⋆) (x :: Γ d)
    .  f (t :&: x) → E k f x
```

The *E* type above provides us with the required kind $\Gamma\ d \to \star$ given another representation with the kind $\Gamma\ (k \to d) \to \star$. In other words, the argument to *E* is another representation where we have an *additional* type variable available in the environment. This new type variable refers to the existential introduced.

The following data type, *Exists*, keeps a value of any type we want, but this type is not visible as an index to the type. Using *E* we can describe its representation in this generic framework:

```
data Exists where
  Exists :: a → Exists
instance Generic◇ X where
  type Rep◇ Exists = E (⋆) (F (Var VZ))
  from◇ (A0 (Exists x)) = E (F x)
  to◇ ϵs (E (F x)) = A0 (Exists x)
```

In conclusion, the introduction of the $(:@@:)$ construction provides a basis for more expressive generic programming. By defining only one new building block, F, we are readily able to represent types of arbitrary kind directly. Once these foundations are laid down, we can accommodate descriptions of types using constraints and existentials in our generic programming framework.

5 Related Work

Weirich and Casinghino [17] discuss how to encode arity and data type-generic operations in Agda. There are two main differences between that work and ours. First, the language of implementation is Agda, a language with full dependent types, as opposed to Haskell. Second, whereas their goal is to define arity-generic operations such as *map*, our goal is to describe notions which exist regardless of the arity, such as $Eq^\diamond$ and $Functor^\diamond$. As a result, we are less concerned about the implementation of the instances.

The work of Hinze [6] tackles poly-kinded generic programming from a different point of view. There generic functions defined over representations of the $\star$ are automatically lifted to data types of higher kinds. It is an interesting avenue to investigate how much of his method can be translated into our setting.

Quantified class constraints [2] allow the programmer to define type classes for constructors, such as *Show1*, by quantification over the constraints for ground types. However, the ability to define the class for arbitrary kinds is still missing. We foresee that a combination of $(:@@:)$ with quantified constraints is possible.

Our representation of types as a type constructor and a list of arguments resembles the applicative fragment of the spine calculus of [3]. In contrast we do not impose any restriction about the shape of the head.

6 Conclusion

In this paper we have presented a way to encode the generalities between types of possibly different kinds by means of polymorphic type classes. The key ingredient being the definition of the $(:@@:)$ data type, which turns n-ary applications into the application of one head and a list of types.

We have discussed generalizations of the well-known *Show*, *Eq*, and *Functor* type classes. In the latter case, we have also discussed how to represent the variance of a type variable as part of the type class. We have also shown how one could extend the *Generic* framework present in GHC to work over types of arbitrary kind with minimal fuss.

References

1. Bolingbroke, M.: Constraint Kinds (2011), http://blog.omega-prime.co.uk/?p=127
2. Bottu, G.J., Karachalias, G., Schrijvers, T., Oliveira, B.C.d.S., Wadler, P.: Quantified class constraints. In: Proc. of the 10th International Symp. on Haskell. Haskell 2017, ACM (2017)
3. Cervesato, I., Pfenning, F.: A linear spine calculus. Journal of Logic and Computation **13**(5), 639–688 (2003)
4. Chakravarty, M.M.T., Keller, G., Peyton Jones, S., Marlow, S.: Associated types with class. In: Proc. of the 32nd Symp. on Principles of Programming Languages. POPL '05, ACM (2005)
5. Eisenberg, R.A., Weirich, S.: Dependently typed programming with singletons. In: Proc. of the 2012 Haskell Symp. Haskell '12, ACM (2012)
6. Hinze, R.: Polytypic values possess polykinded types. In: Procs. of the 5th Intl. Conf. on Mathematics of Program Construction. MPC '00, Springer-Verlag (2000)
7. Jones, M.P.: Type classes with functional dependencies. In: 9th European Symp. on Programming Languages and Systems. ESOP '00, Springer-Verlag (2000)
8. Magalhães, J.P., Dijkstra, A., Jeuring, J., Löh, A.: A generic deriving mechanism for haskell. In: Proc. of the 3rd Symp. on Haskell. Haskell '10, ACM (2010)
9. Miraldo, V.C., Serrano, A.: Sums of products for mutually recursive datatypes: The appropriationist's view on generic programming. In: Proc. of the 3rd Intl. Workshop on Type-Driven Development. TyDe 2018, ACM (2018)
10. Noort, T.v., Rodriguez, A., Holdermans, S., Jeuring, J., Heeren, B.: A lightweight approach to datatype-generic rewriting. In: Proc. of the Workshop on Generic Programming. WGP '08, ACM (2008)
11. Peyton Jones, S., Jones, M., Meijer, E.: Type classes: an exploration of the design space. In: Haskell workshop. Amsterdam (1997)
12. Pickering, M., Érdi, G., Peyton Jones, S., Eisenberg, R.A.: Pattern synonyms. In: Proc. of the 9th Intl. Symp. on Haskell. Haskell 2016, ACM (2016)
13. Schrijvers, T., Peyton Jones, S., Chakravarty, M., Sulzmann, M.: Type checking with open type functions. In: Proc. of the 13th Intl. Conf. on Functional Programming. ICFP '08, ACM (2008)
14. Serrano, A., Miraldo, V.C.: Generic programming of all kinds. In: Proc. of the 11th Symp. on Haskell. Haskell 2018, ACM (2018)
15. de Vries, E., Löh, A.: True sums of products. In: Proc. of the 10th Workshop on Generic Programming. WGP '14, ACM (2014)
16. Wadler, P., Blott, S.: How to make ad-hoc polymorphism less ad hoc. In: Proc. of the 16th Symp. on Principles of Programming Languages. POPL '89, ACM (1989)
17. Weirich, S., Casinghino, C.: Arity-generic datatype-generic programming. In: Proc. of the 4th Workshop on Programming Languages Meets Program Verification. PLPV '10, ACM (2010)
18. Weirich, S., Voizard, A., de Amorim, P.H.A., Eisenberg, R.A.: A specification for dependent types in haskell. Proc. ACM Program. Lang. **1**(ICFP) (2017)
19. Xi, H., Chen, C., Chen, G.: Guarded recursive datatype constructors. In: Proc. of the 30th Symp. on Principles of Programming Languages. POPL '03, ACM (2003)
20. Yakushev, A.R., Holdermans, S., Löh, A., Jeuring, J.: Generic programming with fixed points for mutually recursive datatypes. In: Proc. of the 14th Intl. Conf. on Functional Programming. ICFP '09, ACM (2009)
21. Yorgey, B.A., Weirich, S., Cretin, J., Peyton Jones, S., Vytiniotis, D., Magalhães, J.P.: Giving haskell a promotion. In: Proc. of the 8th Workshop on Types in Language Design and Implementation. TLDI '12, ACM (2012)