Generic Programming of All Kinds

Type Constructors and GADTs in Sum-of-Products Style

Alejandro Serrano
Dept. of Information and Computing Sciences
Utrecht University
The Netherlands
A.SerranoMena@uu.nl

Victor Cacciari Miraldo
Dept. of Information and Computing Sciences
Utrecht University
The Netherlands
V.CacciariMiraldo@uu.nl

Abstract

Datatype-generic programming is a widely used technique to define functions that work regularly over a class of datatypes. Examples include deriving serialization of data, equality or even functoriality. The *state-of-the-art* of generic programming still lacks handling GADTs, multiple type variables, and some other features. This paper exploits modern GHC extensions, including *TypeInType*, to handle arbitrary number of type variables, constraints, and existentials. We also provide an Agda model of our construction that does *not* require Russel's paradox, proving the construction is consistent.

CCS Concepts • **Software and its engineering → Functional languages**; **Data types and structures**;

Keywords Generic programming, Haskell

ACM Reference Format:
Alejandro Serrano and Victor Cacciari Miraldo. 2018. Generic Programming of All Kinds: Type Constructors and GADTs in Sum-of-Products Style. In *Proceedings of ACM SIGPLAN Conference on Programming Languages (Haskell'18).* ACM, New York, NY, USA, 19 pages.

1 Introduction

(Datatype)-generic programming is a technique to define functions by induction over the structure of a datatype. Simpler mechanisms, such as the **deriving** clause, have been present in Haskell for a long time [19], although restricted to a few generic operation such as equality. Over the years, many different approaches have been described to allow the definition of generic functions by the programmer (see [17, 26] for a comparison). Ultimately, GHC added special support via the *Data* [13, 22] and *Generic* [15] classes.

The built-in *Generic* uses a lightweight encoding. One of its key design aspects is to *not* represent recursion explicitly, as opposed to `regular` [23], `multirec` [35], and `generic-mrsop` [21]. Our approach is inspired by the same lightweight philosophy, but we extend it much further, enabling the programmer to employ generic programming techniques to much more expressive datatypes.

For instance, *Generic* only supports representing ground types, that is, types of kind $*$. It does provide a second type-class, *Generic₁*, for type constructors of kind $k \to *$, but that is as far as it goes. Our techniques are able to represent types of arbitrary kinds by using some of the more modern features of the Haskell language.

Our work builds upon many recent extensions to the Haskell language, which have been implemented in GHC. The list includes datatype promotion, kind polymorphism [36], the *Constraint* kind [4], and *TypeInType* [31, 32]. Regarding the latter, we show that our construction does *not* require the $* : *$ axiom by attaching a model in Agda (Appendix A). Nevertheless, using these recent additions we drastically expand the amount of Haskell datatypes we can represent generically compared to other approaches. Take for example the following datatype for simple well-typed expressions:

```
data Expr :: * → * where
  Lit :: a → Expr a
  IsZ :: Expr Int → Expr Bool
  If  :: Expr Bool → Expr a → Expr a → Expr a
```

Internally, GHC enforces a specific type in a constructor – as *Bool* in the constructor *Eq* above – by using equality constraints. Thus the previous declaration is internally translated to the following form:

```
data Expr :: * → * where
  Lit :: a → Expr a
  IsZ :: a ~ Bool ⇒ Expr Int → Expr a
  If  :: Expr Bool → Expr a → Expr a → Expr a
```

Using the approach presented in this paper, this type is encoded using a list of lists of atoms. The outer list represents the choice of constructors, and the inner list represents the fields of such constructors.

Haskell'18, January 01–03, 2017, New York, NY, USA

```
type CodeExpr =
  '['[Explicit V0],
    '[Implicit (Kon (~) :@: V0 :@: Kon Bool),
      Explicit (Kon Expr :@: Kon Int)],
    '[Explicit (Kon Expr :@: Kon Bool),
      Explicit (Kon Expr :@: V0),
      Explicit (Kon Expr :@: V0)]]
```

The three constructors translate into three elements in the outer list. Each of the inner lists contain a list of fields. The type of each of their fields are represented using the applicative fragment of λ-calculus. Constant types and type constructors are brought in via *Kon*, application is represented by (:@:), and V_0, V_1, ... represent each of the type variables of the datatype. For example, the constraint $a \sim Bool$ is represented as the application of the type constructor $(\sim)$ to the first argument V_0 and the constant *Kon Bool*. Our encoding ensures that everything is *well-kinded*.

If a field represents a constraint in Haskell code, it is marked as *implicit* in the code. This is the case for the equality $a \sim Bool$ above. All other fields are marked as explicit.

1.1 Contributions

The main contribution of this paper is to provide a *single, unified* type class for generic programming, which supports algebraic data types and type constructors of *all kinds* (Section 4) by using a variant of the sum-of-products representation. Before delving into the most general framework, we explore the required background in Section 2, and provide a first extension of the sum-of-products style for type constructor with a single parameter in Section 3.

Since the introduction of Generalised Algebraic Data Types [34], *GADTs* for short, datatype constructors may have more complicated shapes than a mere list of fields:

- Each constructor may require one or more *constraints* to be satisfied by the types of their fields. The *Expr* datatype defined above is a prime example of this feature. In Section 5 we explore how to extend our base framework to account for these constraints.
- Constructors may introduce new type variables. These are called *existentials*, since once you pattern match you know that a type has been used, but not exactly which one. The support for existentials is discussed in Section 6.

In Appendix B we address how one could encode explicit recursion within our approach.

2 Preliminaries

Let us take a step back and take a tour of generic programming techniques. We will build up in complexity gradually, ultimately leading to our approach. We focus on the *Generic* line of work, whose main characteristic is the use of type-level information to represent the *shape* of datatypes.

Each generic programming library provides different *building blocks* for the representations. For example, *Generic* uses the following set of functors and combinators:[1]

```
data V1        p                           -- empty
data U1        p = U1                      -- unit
data K1 c      p = K1 c                    -- constant
data (f :+: g) p = L1 (f p) | R1 (g p)     -- sum
data (f :*: g) p = (f p) :*: (g p)         -- product
```

By combining these blocks we can describe the structure of any algebraic datatype. We encode the choice of constructors by sums, and the combination of fields of a constructor by products, or by the unit functor if there are none. In turn, each field is represented by a constant functor. To make things more concrete, here is a definition for binary trees:

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

The shape of this datatype is described as follows:

```
U1 :+: (K1 (Tree a) :*: K1 a :*: K1 (Tree a))
```

The type above is the *representation* of *Tree a*. Note that this representation refers to the type *Tree a* itself. Hence, we say that recursion is encoded *implicitly* here. Other approaches to generic programming use a specific combinator for marking recursive positions instead [23, 35].

The combinators presented above are enough to describe the structure of a datatype, but not metadata such as the names of the type and constructor. *Generic* includes an additional functor M_1 for that purpose, but we omit further discussion for the sake of conciseness.

In order to use generic operations, we must tie each datatype with its representation. This is done via the *Generic* type class. In addition, we ought to witness the isomorphism at the term level via a pair of functions *to* and *from*.

```
class Generic a where
  type Rep a :: * → *
  from :: a → Rep a p
  to   :: Rep a p → a
```

Fortunately, we do not have to write *Generic* instances by hand. GHC provides an extension to the **deriving** mechanism to create these instances. In fact, all generic programming libraries automate this step, making use of metaprogramming facilities such as Template Haskell [29]. In many cases, generic descriptions can also be derived from the compiler-generated one [18].

2.1 Generic operations

In order to define an operation generically, we create two dedicated *type classes*, one for ground types and one for

[1]In the actual library, K_1 has an additional type parameter i, which was used to distinguish recursive positions from non-recursive ones. In latest versions this distinction has been removed and i is always set to R, but the type parameter remains for backwards compatibility.

type constructors. Take the *size* function, which counts the number of constructors.

```
class Size a where
  size  :: a   → Integer
class GSize f where
  gsize :: f p → Integer
```

All we have to do is to write instances of the second class for each of the building blocks of datatypes. The type class mechanism is how we reflect the type level structure into a term level implementation.

```
instance (GSize f, GSize g) ⇒ GSize (f :*: g) where
  gsize (f :*: g) = gsize f + gsize g
instance (GSize f, GSize g) ⇒ GSize (f :+: g) where
  gsize (L1 f)   = gsize f
  gsize (R1 g)   = gsize g
instance (Size c) ⇒ GSize (K1 c) where
  gsize (K1 x)   = size x
```

Note how the instance of constants K_1 points back to the type class for ground types, *Size*. We need one such instance for each datatype; but now we can reuse the generic implementation if we first transform the value into its representation.

```
instance Size Int where
  size n = 1
instance Size (Tree a) where
  size t = gsize (from t)
```

By using the **default** keyword [15] available in GHC, the implementation of *size* in terms of *gsize* can be completely automated. In that case, only an empty **instance** declaration for *Size* is required. The recent **deriving via** proposal [3] provides another way to automatize the creation of such instances.

The definition of *size* is very simple, other generic operations such as *show* or *parse* need to access the metadata and keep additional information around.

2.2 Sums of Products

The landscape of type-level programming in GHC changed radically after the introduction of *datatype promotion* [36], which is used by the `generics-sop` library [6] to guarantee the sum-of-products invariants.

Briefly, by promoting a datatype you can use its constructors as types, and the type being defined is promoted to a kind. For example, we can use a promoted Boolean value to encode whether a certain string has been validated or not.

```
newtype VString (v :: Bool) = VString String
validate :: VString False → VString True
```

In particular, `generics-sop` makes heavy use of promoted lists. Each datatype is described by a list of list of types, where the outer level should be thought as the choice between constructors, and each inner list represents the fields in that constructor. This structure corresponds to that of algebraic datatypes in Haskell, and it is called *sum of products*. We refer to the list of list of types which describe a datatype as the *code* of that datatype. For example, here is the code for *Tree a* defined above:[2]

```
'['[ ], '[Tree a, a, Tree a]]
```

In contrast, the representation using *Generic* is not strong enough to guarantee that shape; the compiler does not stop us from writing:

```
U1 :*: (K1 Int :+: Maybe)
```

that is, a product of sums instead of a sum of products. Furthermore, it uses a functor *Maybe* which is not part of the basic building blocks.

Unfortunately, a new problem arises: we cannot construct terms of the code directly, we first need to turn it into a ground type. The kind of the type level list is $[[*]]$ – where $*$ is the kind of ground types in Haskell – yet, we can only write terms of kind $*$. The bridge between the two worlds is given by the following two GADTs: *NS*, which interprets a list of elements as a choice, and *NP*, which requires a value for each element in the list, and thus encodes a product.

```
data NS :: (k → *) → [k] → * where
  Here  :: f k       → NS f (k ': ks)
  There :: NS f ks → NS f (k ': ks)
data NP :: (k → *) → [k] → * where
  Nil   ::                NP f '[ ]
  (:*)  :: f x → NP f xs → NP f (x ': xs)
```

Both *NS* and *NP* receive as first argument a type constructor of kind $k \to *$. Both *NS* and *NP* apply that constructor to the elements of the list: to one of them in *Here*, and to all of them in $(:*)$.

The most common combination of *NS* and *NP* is used to obtain the ground type representing a certain code c:

```
type SOP c = NS (NP I) c
```

The idea is that we choose one of the constructors in the outer list by using *NS*, and then apply *NP I* to ask for one value of every element for the chosen constructor. The argument to *NP* is the identity functor *I* defined as:

```
data I p = I p
```

As a result, we require for each field one value of exactly the type declared in the inner list. As we shall see, the ability to manipulate the inner lists is paramount to our approach.

Just like the built-in *Generic*, each datatype is tied to its code by a type class. In the `generics-sop` this class is also known as *Generic*, but we use a superscript to distinguish it.

[2] The quote sign serves to differentiate type level from term level when there is a risk of confusion. For example, [ ] is the name of the list *type constructor*, whereas '[ ] is the *promoted empty list*.

```
class Generic^sop a where
  type Code a :: [[*]]
  from :: a → SOP (Code a)
  to   :: SOP (Code a) → a
```

This approach to generic programming allows the definition of generic operations without resorting to the type class mechanism. By pattern matching on the *NS* and *NP* constructors we gain enough information about the shape of the datatype. For example, here is the definition of the generic *size* operation:

```
gsize :: (Generic^sop a, All_2 Size (Code a)) ⇒ a → Integer
gsize = goS ∘ from
  where goS (Here  x) = goP x
        goS (There x) = goS x
        goP Nil       = 0
        goP (x :* xs) = size x + goP xs
```

The only remarkable part of this implementation is the use of the All_2 type class to ensure that every type which appears in a field of the code has a corresponding *size* operation. We omit further details about All_2, the interested reader is referred to de Vries and Löh [6].

3 Generic Type Constructors

If we want to write functions such as *fmap*, from *Functor*, generically, we need to have knowledge about which fields of a type of kind $* \to *$ are, in fact, an occurence of its type parameter. In this section we look into how this has been done by *GHC.Generics* and how to translate this to the `generics-sop` style.

3.1 Type constructors using $Generic_1$

The $Generic_1$ type class is the counterpart of *Generic* for types with one parameter, such as *Maybe* or [ ]. The definition is pretty similar, except that $from_1$ and to_1 take as argument an instantiated version $f\ a$ of the type constructor f. The same instantiation is done in the generic representation.

```
class Generic_1 f where
  type Rep_1 f :: * → *
  from_1 :: f a → Rep f a
  to_1   :: Rep f a → f a
```

We now need to extend our set of building blocks, since in the case of type constructors we have an additional possibility for the fields, namely referring to the type variable a in f. Par_1 is used to represent the case in which the type variable appears "naked", that is, direcly as a:

```
data Par_1 p = Par_1 p
```

Par_1 is not enough to represent a type like *Tree a*, in which a also appears as part of a larger type – in this case *Tree a* again for the subtrees. We introduce another building block:

```
data Par_1   p = Par_1 p
data Rec_1 f p = Rec_1 (f p)
```

We can now give a representation of the *Tree* type constructor. In contrast, before we had defined a family of representations for *Tree a*, regardless of the a.

```
U_1 :+: (Rec_1 Tree :*: Par_1 :*: Rec_1 Tree)
```

Although type application is named Rec_1, and it is used when we have recursion in our type, is does not only model recursion. In fact, every application of a type constructor to the type variable has to be encoded by the means of Rec_1, even if this is not a recursive application.

Note that Rec_1 is not expressive enough to represent arbitrary recursion. If we want to represent non-regular datatypes, such as *Rose a* below:

```
data Rose a = Fork a [Rose a]
```

we need some extra machinery in the form of *functor composition*. We omit discussion of these non-regular datatypes in this section, but note that the framework in forecoming sections *does* support this shape of recursion.

Defining generic operations for $Generic_1$ is done as for *Generic*; one just need to add additional instances for the new Par_1 and Rec_1 functors. Here are the important pieces of the generic *fmap* declaration, taken from the `generic-deriving` package. The rest of the instances just apply *gfmap* recursively in every position.

```
class GFunctor f where
  gfmap :: (a → b) → f a → f b
instance GFunctor Par_1 where
  gfmap f (Par_1 a) = Par_1 (f a)
instance (Functor f) ⇒ GFunctor (Rec_1 f) where
  gfmap f (Rec_1 a) = Rec_1 (fmap f a)
```

Although $Generic_1$ works well for one type parameter, the general technique does not scale to more parameters. At the very least, we would need new Par_1 and Par_2 types which refer to each of the type variables.

```
data Par_1 a b = Par_1 a
data Par_2 a b = Par_2 b
```

By doing so, the kind of the representation can no longer be $* \to *$, we need at least $* \to * \to *$ to accomodate the two type parameter. Unfortunately, this means that none of V_1, U_1, $(:+:)$, and $(:*:)$ can be used, since they all create or operate on types of kind $* \to *$. We could build a completely different set of primitive building blocks for two-parameter types, but the problem would repeat again once we consider three parameters. We will address this issue in Section 4.

3.2 Type constructors in sum-of-products style

The key point to extend a generic framework to handle type constructors is to introduce marks for those places where the type parameter ought to appear. In the case of *Generic*$_1$, it was only a matter of adding new *Par*$_1$ and *Rec*$_1$ types.

The approach taken by `generics-sop` is to describe a datatype by a list of list of types. Ultimately, the elements of the nested lists are ground types, of kind ∗. This precludes us from using the same form of codes directly, since we cannot add an indicators for variables or recursion. Instead, we introduce *atoms*, which describe the choice we have for each of our fields:

```
data Atom = Var | Rec (∗ → ∗) | Kon (∗)
```

A code is no longer represented by [[∗]], but rather [[*Atom*]]. The *NS* and *NP* types which interpret those codes are still valid; the nested list structure is still there. But we also need to interpret each of the atoms into a value of kind ∗. For that we introduce yet another layer, which we call *NA*:

```
data NA :: ∗ → Atom → ∗ where
  V :: a   → NA a Var
  R :: f a → NA a (Rec f)
  K :: k   → NA a (Kon k)
```

Each of the constructors in *NA* closely matches the definition of *Par*$_1$, *Rec*$_1$, and *K*$_1$ in the *Generic*$_1$ framework. The code for our running example, *Tree*, reads as follows:

```
'['[], '[Rec Tree, Var, Rec Tree]]
```

We can now define the $Generic_1^{sop}$ type class which ties each datatype to its code. We also define a *SOP*$_1$ type synonym for the nested application of the interpretation functors *NS*, *NP*, and *NA*:

```
type SOP₁ c a = NS (NP (NA a)) c
class Generic₁ˢᵒᵖ (f :: ∗ → ∗) where
  type Code₁ f :: [[Atom]]
  from₁ :: f a → SOP₁ (Code₁ f) a
  to₁   :: SOP₁ (Code₁ f) a → f a
```

Up to this point we have omitted the implementation of the functions which witness the isomorphism between a regular datatype and its generic representation. It is instructive to consider how it looks for the case of *Tree* seen as a type constructor:

```
instance Generic₁ˢᵒᵖ Tree where
  ...
  from Leaf = Here  Nil
  from (Node l x r)
    = There $ Here $ R l :∗ V x :∗ R r :∗ Nil
```

After a sequence of *There* and *Here* indicating which constructor we are working with, we find a (:∗)-separated list of fields, finished by *Nil*. In each case we need to mark whether that field arises from an application of a type constructor to the parameter (with *R*), for an ocurrence of the type parameter (with *V*), or simply from a constant type (with *K*, not shown in this example).

```
gfmap :: forall f a b .
         (Generic₁ˢᵒᵖ f, AllRec₂ Functor (Code₁ f))
       ⇒ (a → b) → f a → f b
gfmap f = to ∘ goS ∘ from
  where
    goS :: AllRec₂ Functor xs
        ⇒ NS (NP (NA a)) xs → NS (NP (NA b)) xs
    goS (Here  x) = Here  (goP x)
    goS (There x) = There (goS x)

    goP :: AllRec Functor xs
        ⇒ NP (NA a) xs → NP (NA b) xs
    goP Nil          = Nil
    goP (R x :∗ xs)  = (R $ fmap f x) :∗ goP xs
    goP (V x :∗ xs)  = V (f x)        :∗ goP xs
    goP (K x :∗ xs)  = K x            :∗ goP xs
type family AllRec₂ c xs :: Constraint where
  AllRec₂ c '[]          = ()
  AllRec₂ c (x ': xs)    = (AllRec c x, AllRec₂ c xs)
type family AllRec  c xs :: Constraint where
  AllRec c '[]             = ()
  AllRec c (Rec x ': xs) = (c x, AllRec c xs)
  AllRec c (x     ': xs) = AllRec c xs
```

Figure 1. Generic *fmap* using $Generic_1^{sop}$

Armed with our new $Generic_1^{sop}$, we can implement a generic version of *fmap*, given in Figure 1. The code is a bit more complex than *gsize*, though. The types involved in *goS*, *goP*, and *goA* are too complex to be inferred, hence, we must help the compiler by annotating the local declarations.

We cannot use the same type family *All*$_2$ that we were using, because we need to treat *Rec* positions differently from the rest. The solution is to define a more specific *AllRec*$_2$ type family, which only applies the a constraint *c* only over those positions. Unfortunately, we have not yet found a way to implement *AllRec*$_2$ in terms of *All*$_2$.

4 Generics of All Kinds

The extension of $Generic^{sop}$ to $Generic_1^{sop}$ was done in three steps. First, we changed the language of codes from lists of lists of *types*, to lists of lists of *atoms*. By doing so, we were able to refer to type parameters via *Var* and encode recursion via *Rec*. Next, we introduced an interpretation functor *NA* for atoms. Finally, we defined the $Generic_1^{sop}$ type class to tie each datatype to its code. In this section we follow the same three steps, but this time we go further. The resulting generic type-class supports types of arbitrary kinds.

The first step is representing each of the types of the fields, that is, estabilishing our new language of atoms. In Section 3 we discussed the problem of nested recursion, as

```
type Kind = (∗)
data TyVar (ζ :: Kind) (k :: Kind) :: (∗) where
  VZ ::                 TyVar (x → xs) x
  VS :: TyVar xs k → TyVar (x → xs) k
data Atom (ζ :: Kind) (k :: Kind) :: (∗) where
  Var   :: TyVar ζ k → Atom ζ k
  Kon   :: k           → Atom ζ k
  (:@:) :: Atom ζ (ℓ → k) → Atom ζ ℓ → Atom ζ k
```

Figure 2. Definition of *Atom*

in *Rose*. Here this problem is magnified; for example, in the following datatype of kind $* \to (* \to *) \to * \to *$:

```
newtype ReaderT r m a = Reader (r → m a)
```

the second type variable is applied to the third one. We cannot foresee all possible combinations of recursion and application, and thus extending the possibilities of the *Rec* constructor from *Atom* is a dead end.

Looking at the Haskell Report [19, section 4.1.2], we see that a type follows closely the applicative fragment of the λ-calculus: it is either a type variable, a type constructor, or an application. We use well-known techniques to represent the structure of these types as terms:

1. We use de Bruijn indices to refer to variables.
2. We prevent ill-kinded types such as *Kon Int* :@: *Var Z* by keeping track of the kind of the type being described as an index [2].

In Figure 2 we define two GADTs, *TyVar* and *Atom*, to represent types. For the sake of clarity, we also define a synonym *Kind* for $(*)$, which we used whenever the instantiation of a type argument represents a kind as opposed to a regular type. Both *TyVar* and *Atom* receive two *Kind*-indices: the first one ζ represents the kind of the datatype we are describing, and the second one k gives the kind of the type represented by the *Atom* itself. The latter index is the one preventing ill-kinded expressions such as *Kon Int* :@: *Var Z*.

The *TyVar* represents a de Bruijn index into the datatype ζ. We represent this index as a Peano numeral using *VZ* and *VS*, and ensure that references are never out of bounds. For simplicity, we also introduce some synonyms in order to make the descriptions of types in the rest of the paper a bit more concise:

```
type V₀ = Var VZ
type V₁ = Var (VS VZ)
type V₂ = Var (VS (VS VZ))
```

A datatype of kind ζ is now encoded as a list of list of the atoms defined above, given that they form a type of kind $*$. We define a type synonym *DataType* to refer to such lists:

```
type DataType ζ = [[Atom ζ (∗)]]
```

At this point we need the *Atom* datatype to be promoted, in order to use it in the list defining the code. Since *Atom* is a GADT, we are required to enable the *TypeInType* extension to promote it. However, we are not using the $* : *$ judgement in our construction, we discuss this matter in Section 4.3.

As an example, here are the codes corresponding to [ ], *Either*, and *Rose*. The fact that *Either* has two type parameter can be observed by the usage of both V_0 and V_1. The non-regular recursion pattern in *Rose* is translated to iterated uses of the application (:@:).

```
type ListCode   = '['[ ], '[V₀, Kon [ ] :@: V₀]]
type EitherCode = '['[V₀], '[V₁]]
type RoseCode   = '['[V₀, Kon [ ] :@: (Kon Rose :@: V₀)]]
```

Interpreting Atoms. In order to interpret this new language, none of the previously defined *NS* and *NP* require any modifications. On the other hand, the interpretation of atoms requires some type engineering. Recall the definition of *NA* given in Section 3:

```
data NA :: ∗ → Atom → ∗ where
  V :: a → NA a Var
  ...
```

The first argument of kind $*$ represents the type of the argument of the functor we are interpreting. In the current setting, we might have an arbitray number of arguments, and these might be of arbitrary kinds other than $*$. This notion is not new: to interpret a term possibly containing variables we need a *context* — commonly represented by the Greek letter Γ – which assigns a type to each of the variables. Other than the special shape of the index, the datatype for contexts looks very much like an heterogeneous list.

```
data Γ (ζ :: Kind) where
  ϵ     ::               Γ (∗)
  (:&:) :: k → Γ ks → Γ (k → ks)
```

For example, *Int* :&: *Maybe* :&: *Char* :&: ϵ is a well-formed context of kind $\Gamma\ (* \to (* \to *) \to * \to *)$.

With the introduction of contexts, we can write a definitive version of *NA*. In contrast to the previous iteration, we do not have a different constructor for each possible value of *Atom*. It is not possible to build interpretations for *Atom* in a compositional way: for example, it is not possible to define the interpretation *Kon* [ ] :@: *Kon Int* by composing interpretations of *Kon* [ ] and *Kon Int* because the notion of a value of something of a kind different from $*$, like [ ], does not make sense in Haskell. Instead, we define a type family *Ty* which computes the type of a field given a context.

```
type family Ty ζ (α :: Γ ζ) (t :: Atom ζ k) :: k where
  Ty (k → ks) (t :&: α) (Var VZ)      = t
  Ty (k → ks) (t :&: α) (Var (VS v)) = Ty ks α (Var v)
  Ty ζ α (Kon t)   = t
  Ty ζ α (f :@: x) = (Ty ζ α f) (Ty ζ α x)
```

In preparation for the upcoming constructions, we introduce the type variables for the *T* constructor explicitly. In particular, this is required to use visible type application [8]. The code uses record syntax to generate an eliminator *unT* for the only field in *T*.[3]

```
data NA (ζ :: Kind) :: Γ ζ → Atom ζ (∗) → ∗ where
  T :: forall ζ t α . { unT :: Ty ζ α t } → NA ζ α t
```

These new parameters to *NA* appear also in the new $SOP_\star$ type, which interpets codes of the new shape:

```
type SOP⋆ (ζ :: Kind) (c :: DataType ζ) (α :: Γ ζ)
  = NS (NP (NA ζ α)) c
```

A unified $Generic_\star^{sop}$**.** The definition of the generic representation of a certain datatype is not comprised only of its code, the two isomorphisms *from* and *to* are also required. In our case, however, it seems like there is a family of such conversion functions:

```
from  :: f     → SOP⋆ (∗)             (Code f) ϵ
from_1 :: f a   → SOP⋆ (k_1 → ∗)       (Code f) (a :&: ϵ)
from_2 :: f a b → SOP⋆ (k_1 → k_2 → ∗) (Code f) (a :&: b :&: ϵ)
 -- and so on
```

The difficulty arises on the first argument, since *f* is applied to a different number of variables in each case. Hence, we have to describe this situation as *f* being applied to a *context* whose length varies.

```
from  :: Apply f ϵ              → SOP⋆ ...
from_1 :: Apply f (a :&: ϵ)      → SOP⋆ ...
from_2 :: Apply f (a :&: b :&: ϵ) → SOP⋆ ...
 -- and so on
```

Where *Apply* is defined as a type family. In order to help the compiler in forecoming developments, we also include the kind of the context as an explicit argument to this family.

```
type family Apply ζ (f :: ζ) (α :: Γ ζ) :: (∗) where
  Apply (∗)       f ϵ       = f
  Apply (k → ks) f (t :&: τ) = Apply ks (f t) τ
```

This is not yet sufficient to type either the *from* or *to* functions. The complete declarations can be found in Figure 3, together with an example instance. For pedagogical purposes, let us start from a naive type signature for *to* and build it up to the real signature, one piece at a time. We start with:

```
to :: SOP⋆ ζ (Code f) α → Apply ζ f α
```

Here the type variable *f* appears only as an argument of type families: *Apply* and *Code*. None of the type families are injective, which means that the instantiation of *f* cannot be inferred in its usage sites. In fact, if we have *Either a b* as a result, there are three different calls to *Apply* which would give the same result:

```
Apply (∗ → ∗ → ∗) Either    (a :&: b :&: ϵ)
Apply (∗ → ∗)     (Either a)   (b :&: ϵ)
Apply (∗)         (Either a b)        ϵ
```

Datatypes, on the other hand, are injective. Hence we lift *Apply* to a GADT, *ApplyT*. This provides *evidence* of which part of the type is the constructor *f* and which are the type parameters.

Next, we need to inform the typechecker about the shape of the context $\Gamma\ \zeta$. Unfortunately Haskell cannot infer that only from the kind ζ, even though this should be enough in theory. We introduce singletons for contexts, $S\Gamma$, and an accompanying type class $SFor\Gamma$, which witnesses the one-to-one correspondence between the singleton term and its indexed context. In short, a singleton for τ is a datatype indexed by that τ which accurately reflects the structure of τ [7]. Thus, by pattern matching on the singleton, we gain

[3]Not to be confused with Agda and Idris' syntax for implicit parameters.

```
class Generic⋆^sop ζ (f :: ζ) where
  type Code f :: DataType ζ
  from :: ApplyT ζ f α → SOP⋆ ζ (Code f) α
  to   :: SFor Γ ζ α
       ⇒ SOP⋆ ζ (Code f) α → ApplyT ζ f α
data ApplyT ζ (f :: k) (α :: Γ ζ) :: ∗ where
  A_0 :: { unA_0 :: f } → ApplyT (∗)     f ϵ
  A^+ :: { unA^+ :: ApplyT ks (f t) τ }
                      → ApplyT (k → ks) f (t :&: τ)
data SΓ (ζ :: Kind) (α :: Γ ζ) where
  Sϵ ::                SΓ (∗)       ϵ
  S& :: SΓ ks α → SΓ (k → ks) (t :&: τ)
class SForΓ k (α :: Γ k) where
  sΓ :: SΓ k τ
instance SForΓ (∗) ϵ where
  sΓ = Sϵ
instance SForΓ ks τ ⇒ SForΓ (k → ks) (t :&: τ) where
  sΓ = S& sΓ

instance Generic⋆^sop (∗ → ∗) [ ] where
  type Code [ ] = '['[ ], '[ V_0, Kon [ ] :@: V_0 ] ]
  from (A^+ (A_0 [ ]))    =         Here $ Nil
  from (A^+ (A_0 (x:xs))) = There $ Here $ T x :∗ T xs :∗ Nil
  to :: forall α . SForΓ (∗ → ∗) α
     ⇒ SOP⋆ (∗ → ∗) (Code [ ]) α → ApplyT (∗ → ∗) [ ] α
  to sop = case sΓ@(∗ → ∗)@α of
    S& Sϵ → case sop of
                Here Nil                   → A^+ $ A_0 [ ]
          There (Here (T x :∗ T xs :∗ Nil)) → A^+ $ A_0 $ x:xs
```

Figure 3. The $Generic_\star^{sop}$ class and its instance for lists

```
gfmap   :: forall f a b . (Generic_★^sop (∗ → ∗) f,
             All_2 FunctorAtom (Code f))
          ⇒ (a → b) → f a → f b
gfmap f  =        unA_0 ∘ unA^+ ∘ to
          ∘ goS ∘ from ∘ A^+ ∘ A_0
  where
    goS :: All_2 FunctorAtom xs
        ⇒ NS (NP (NA (∗ → ∗) (a :&: ε))) xs
        → NS (NP (NA (∗ → ∗) (b :&: ε))) xs
    goS (Here  x) = Here  (goP x)
    goS (There x) = There (goS x)
    goP :: All FunctorAtom xs
        ⇒ NP (NA (∗ → ∗) (a :&: ε)) xs
        → NP (NA (∗ → ∗) (b :&: ε)) xs
    goP Nil          = Nil
    goP (T x :∗ xs)  = gfmapF f (T x) :∗ goP xs
class FunctorAtom (t :: Atom (∗ → ∗) (∗)) where
  gfmapF :: (a → b) → NA (∗ → ∗) (a :&: ε) t
                    → NA (∗ → ∗) (b :&: ε) t
instance FunctorAtom V_0 where
  gfmapF f (T x) = T (f x)
instance (Functor f, FunctorAtom x)
          ⇒ FunctorAtom (Kon f :@: x) where
  gfmapF f (T x)
    = T (fmap (unT ∘ gfmapF f ∘ T @_@x) x)
instance FunctorAtom (Kon t) where
  gfmapF f (T x) = T x
```

Figure 4. Generic *fmap* using $\mathit{Generic}_{\star}^{\mathsf{sop}}$

information about τ itself. In this case, the information about the shape of the context is reified.

4.1 Generic Functor

Just like Figure 1, we can also write a generic *fmap* for functors in the *GenericsNSOP* universe, given in Figure 4. In fact, the code in Figure 4, does not significantly differ from that of Figure 1, where we only had $\mathit{Generic}_{1}^{\mathsf{sop}}$ at our hand. The main change is our treatment of atoms. In the case of $\mathit{Generic}_{1}^{\mathsf{sop}}$, we knew how to implement *fmap* for every possible shape of field. However, the language of atoms in $\mathit{Generic}_{\star}^{\mathsf{sop}}$ is much broader, and we cannot always write the desired implementation. In order to delineate which *Atom*s we can handle, we introduce a *FunctorAtom* type class. The instances correspond to three different scenarios:

- If the field mentions the type variable, then we apply the function of type $a \to b$ to it.
- If the field has the form $f\ a$, where f is a functor and a the type variable, we can apply the operation under the functor f. This idea generalizes to fields of the form $f_1\ (\ldots\ (f_n\ a))$, giving rise to a recursive instance.
- Finally, if the field does not mention the variable, *Kon t*, we just keep it unchanged.

In the second instance we make use of a partial type signature @_ [33]. That way we can ask the compiler to infer the kind which ought to be passed to *T* from the surrounding context. In this case it can be readily obtained from the following @*x* type application.

Note that this use of type classes in the definition of generic *fmap* is quite different from the usage in Section 2.1. There the instances describe how to handle sums and products, which we do simply by recursion on the structure of *NS* and *NP*. In our case *FunctorAtom* describes which shapes of *atoms* can appear as fields of a datatype which supports the *Functor* operations. We need such a restriction because the universe of types we can describe is very wide, and in many scenarios only a subset of those can be handled.

Another important difference from the $\mathit{Generic}_{1}^{\mathsf{sop}}$ version is that we need to manually wrap and unwrap *ApplyT* constructors A_0 and A^+. Note that the *user* of the generic operation is oblivious to these fact, they can use the operation directly, as the following example from the interpreter shows:

```
> gfmap (+1) [1,2,3]
[2,3,4]
```

Arity-generic ***fmap*****.** The construction in this section can be generalized to work on type constructors of every kind of the form $\ast \to \ldots \to \ast \to \ast$, that is, taking only ground types as type arguments. Using our framework, we automatize the instantiation of the following type class *KFunctor*, which generalizes *Functor* and *Bifunctor* to any kind of the aforementioned shape

```
class KFunctor ζ (f :: ζ) where
  kmap :: SForΓ ζ β ⇒ Mappings α β
        → ApplyT ζ f α → ApplyT ζ f β
```

The *fmap* method in *Functor* takes one single function $a \to b$ as argument, since there is only one type variable to update. The *bimap* in *Bifunctor* takes two functions, one per argument. The following *Mappings* data type generalized this idea for *KFunctor*, requiring one function per type variable.

```
data Mappings (α :: Γ ζ) (β :: Γ ζ) where
  MNil  :: Mappings ε ε
  MCons :: (a → b) → Mappings α β
        → Mappings (a :&: α) (b :&: β)
```

Assuming the $\mathit{Generic}_{\star}^{\mathsf{sop}}$ instance for lists, one can implement the usual *map* function as follows:

```
map :: (a → b) → [a] → [b]
map f = unA_0 ∘ unA^+ ∘ kmap (MCons f MNil) ∘ A^+ ∘ A_0
```

Since we need to pass an *ApplyT* value to *kmap*, we need to manually wrap and unwrap using A_0 and A^+. This process can be automated, though, and we explain how to do so in the next section. The other detail to consider is that we need to create a *Mappings* with the single function f, as [ ] only takes one type argument.

The full implementation of *kmap* is given in Appendix C. The version described also supports constraints in datatype constructors as described in Section 5.

4.2 Unraveling Singletons

Although the type family *Apply* exposes a nicer interface to the programmer, the introduction of the *ApplyT* datatype was essential for the definition of $Generic^{sop}_{\star}$. Turns out we can maintain that nice interface by wrapping or unwrapping values using A_0 and A^+ automatically.

Going from *ApplyT* to *Apply* is trivial. This is expected, as *ApplyT* is a refinement of the type family in which the evidence is explicit.

```
unravel :: ApplyT k f α → Apply k f α
unravel (A₀ x) = x
unravel (A⁺ x) = unravel x
```

For the converse direction, we find ourselves in the same scenario as for the definition of *to*. In order to define the function, we need to match on the shape of the context: it the context is empty we wrap the value using A_0, and otherwise we add one layer of A^+. The solution is asking for a singleton and inspecting that value instead.

```
ravel  :: forall k f α . SForΓ k α
       ⇒ Apply k f α → ApplyT k f α
ravel = go (sΓ @_@α)
  where
    go  :: forall k f α . SΓ k α
        → Apply k f α → ApplyT k f α
    go Sε       x = A₀ x
    go (S& τ) x = A⁺ (go τ x)
```

The definition of *gfmap* no longer needs to care about the amount of wrapping needed by the generic operation, this is inferred from the types involved.

```
gfmap f = unravel ∘ to ∘ goS ∘ from ∘ ravel
```

In fact, we can expose only the combination of (un)raveling with *to* and *from*, making the writer of generic operations completely unaware of the intermediate *ApplyT* datatype.

4.3 Are We Inconsistent?

In order to perform the entire construction, we were forced to enable the *TypeInType* extension in GHC. This extension is quite powerful: it allows working with kinds as they were types, and to promote GADTs, among others. But it also adds an axiom $* : *$, which is known to introduce inconsistency when we view the language as a logic [9]. We would like to know whether this latter axiom is necessary for our construction, or it could be achieved using a hierarchy of universe levels.

To answer the question we have build a model of $Generic^{sop}_{\star}$ in Agda, which we describe in Appendix A. If we assume that our universe of basic types lives in Set_0, our codes live in Set_1, and our interpretation of those in Set_2. The code compiles fine, showing that the $* : *$ axiom is not essential to our construction.

5 Constraints

The move from ADTs to GADTs makes it possible to require a constraint to be satisfied when using a certain constructor of a datatype. The *Expr* type described in the Introduction is one example: it mandates the index to be exactly *Bool* in the *IsZ* case. Here is another example, in which the constructor *Refl* mandates the two type arguments to coincide by imposing an equality constraint $a \sim b$.

```
data Eql a b where
  Refl :: a ∼ b ⇒ Eql a b
```

Since version 7.4.1, GHC treats constraints — in short, anything which appears before the ⇒ arrow — as regular ground types, with the caveat that its kind is *Constraint* instead of $*$. We sometimes refer to constraints as *implicit* parameters, since they are filled in by the compiler, as opposed to explicit parameters which need to be given in the code. In fact, other languages such as Agda and Scala have a native notion of implicit parameters, which are often used to simulate Haskell's type class mechanism.

Up to now a datatype was defined as [[*Atom* ζ (∗)]], where each element of the inner list represents a field in a constructor. Now we introduce an additional layer, which specified for each field whether it is implicit — and thus should have kind *Constraint* — or explicit.

```
data Field (ζ :: Kind) where
  Explicit :: Atom ζ (∗)          → Field ζ
  Implicit :: Atom ζ Constraint → Field ζ
type DataType ζ = [[Field ζ]]
```

The interpretation functor *NA* has to be adapted:

```
data NA (ζ :: Kind) :: Γ ζ → Field ζ → ∗ where
  E :: forall ζ t α . Ty ζ α t → NA ζ α (Explicit t)
  I :: forall ζ t α . Ty ζ α t ⇒ NA ζ α (Implicit t)
```

The two constructors look almost the same. But the fact that *Ty* ζ α *t* appears before a regular → arrow in *E*, and before a ⇒ arrow in *I* is enough to require the right kind to come out of the application of *Ty*. This updated framework is enough to describe the shape of the *Eql* datatype; we give its $Generic^{sop}_{\star}$ instance in Figure 5.

This is a rather slim layer that has to be added on top of the previous constructions. Our Agda model in Appendix A has constraints in it. Hence, this layer introduces no inconsistency.

```
instance Generic⋆^sop (k → k → ∗) Eql where
  type Code Eql
    = '['[Implicit (Kon (∼) :@: V₀ :@: V₁)]]
  from (A⁺ (A⁺ (A₀ Refl))) = Here $ I :∗ Nil
  to  :: forall α . SForΓ (k → k → ∗) α
    ⇒ SOP⋆ (k → k → ∗) (Code Refl) α
    → ApplyT (k → k → ∗) Refl α
  to sop = case sΓ @_@α of
    S& (S& Sϵ) → case sop of
      Here (I :∗ Nil) → A⁺ $ A⁺ $ A₀ Refl
```

Figure 5. The $Generic^{sop}_{\star}$ instance for *Eql*

6 Existentials

Apart from constraints, every constructor in a GADT may introduce one or more *existentially quantified type variables*, which are available once your pattern match. Although seemingly rare, existentials become ubiquituous once you consider how GHC represents GADTs. Consider another simple type of well-typed expressions, which features only integer literals and pairs:

```
data Expr′ t where
  AnInt :: Int → Expr′ Int
  APair :: Expr′ a → Expr′ b → Expr′ (a, b)
```

Under the hood, the refinement in the index of *Expr′* is turned into an equality constraint, and every new new variable is quantified. Thus, we obtain the following form:

```
data Expr′ t where
  AnInt ::                 t ∼ Int    ⇒ Int → Expr′ t
  APair :: forall a b . t ∼ (a, b) ⇒ Expr′ a → Expr′ b
                                     → Expr′ t
```

Our language of codes is enough to describe *AnInt*, but cannot handle the introduction of new variables in *APair*. Let us look at what would it take to extend our technique to handle existential types.

Existentials are introduced at the level of constructors. Before, each constructor was merely a [*Field* ζ], but now we are going to refine it with the possibility of introducing new type variables; for each new variable we need to record its kind. In a dependently-typed language we can encode this construction using a recursive *Branch* datatype: we introduce new variables by repeated applications of *Exists*, and then move to describe the fields with *Constr*.

```
data Branch (ζ :: Kind) where
  Exists  :: (ℓ :: Kind) → Branch (ℓ → ζ) → Branch ζ
  Constr :: [Field ζ]                          → Branch ζ
```

Haskell does not support full dependent types, so *Branch* cannot be declared as above. As we did in Section 4 for the indices of variables, we use a singleton instead. In contrast with *SNat*, *SKind* does not reflect any information about the kind itself, but we do not need to inspect that information either for the upcoming constructions.[4]

```
data SKind (ℓ :: Kind) = K
data Branch (ζ :: Kind) where
  Exists  :: SKind ℓ → Branch (ℓ → ζ) → Branch ζ
  Constr :: [Field ζ]                     → Branch ζ
```

As a consequence of this intermediate layer, datatypes are no longer represented as mere lists of lists of fields, but as [*Branch* ζ], where each element contains information both about existentials and about the fields.

The *Expr′* datatype above can be described using our extended language of codes as given in Figure 6. For that, we do not use the user-facing version, but the second representation with explicit quantification and equalities. Note that each *Exists* "shift" the position of type variables: the first variable in the context is now the second, and so on. As a result, V_0 refers to the last-introduced variable, b in this case, V_1 corresponds to a, and V_2 is the original type argument to *Expr′*, which we called t in the datatype declaration.

The next step is to update the interpretation of the codes. Unfortunately, our datatypes are not described by a list of lists anymore. This means we cannot define its interpretation as a simple composition of *NS*, *NP*, and *NA*. We then introduce *NB*, which interprets *Branch*es and has the form:

```
data NB (ζ :: Kind) :: Γ ζ → Branch ζ → ∗ where
  Ex :: forall ℓ (t :: ℓ) (p :: SKind ℓ) ζ α c .
        NB (ℓ → ζ) (t :&: α) c → NB ζ α (Exists p c)
  Cr :: NP (NA ζ α) fs          → NB ζ α (Constr fs)
```

The recursion in the syntax of existential quantification is reflected in the recursive use of *NB* in the constructor *Ex*. More importantly, thanks to the singleton *SKind* ℓ we can obtain the kind ℓ which was introduced in the code. Then, we use existential quantification at the meta-level to generate a fresh type t of that kind, which we add to the context in the first position, matching the change in the structure that *Exists* performs in the kind. Once we do not need more existential variables, *Cr* just continues as usual, by requiring *NP* (*NA* ζ α) for the fields *fs*.

Since now the call to *NP* is inside *NB*, we need to update the top-level $SOP_{\star}$ type too.

```
type SOP⋆ ζ (c :: DataType ζ) (α :: Γ ζ) = NS (NB ζ α) c
```

The $Generic^{sop}_{\star}$ type class, on the other hand, is not affected by these changes. The instances, however, need to change their codes and isomorphisms to reflect the new intermediate layer between outer and inner lists.

[4]Peyton Jones et al. [25] describes *TypeRep*, which provides type-indexed type representations. However, it is not (yet) possible to promote *TypeRep* operations to the type level.

```
type TmCode
  = '[                      Constr '[Implicit (Kon (~) :@: Kon Int :@: V0), Explicit (Kon Int)],
       Exists K (Exists K (Constr '[Implicit (Kon (~) :@: V2 :@: (Kon (,) :@: V1 :@: V0)),
                                    Explicit (Kon Expr' :@: V1), Explicit (Kon Expr' :@: V0)]))]
```

Figure 6. Code for *Expr′*

7 Related Work

Sums of products. Our approach to generic programming is heavily inspired by the original list-of-list-of-types construction by de Vries and Löh [6]. Each of the extensions we present: support for multiple kinds, constraints, explicit recursion, and existentials, could be applied independently of the original framework. Conversely, `generics-sop` supports metadata about types and constructors, and the same techniques are readily applicable to our case.

There seems to be a trade-off in the amount of *traversal combinators* that can be implemented. `generics-sop` comes with a huge library of maps, sequences, and folds. Our definition of *gfmap*, on the other hand, traverses the $SOP_\star$ structure manually; and we cannot easily abstract that pattern because of the very strong types which are involved.

Generic universes. With respect to our Agda model, Appendix A, we have adapted the technique of generic programming with universes [1], where one separates the description of types from their interpretation into different hierarchies. That is, if the description of types live in Set_i, then their interpretation lives in Set_{i+1}. Differently from Altenkirch et al. [1], we enforce that the descriptions must be in the *sums-of-products* shape, we do not handle mutual recursion and we handle predicates over variables. This requires us to put the elements of the interpretation in Set_{i+2}. These differences stems from the fact that we aim at representing Haskell datatypes, including GADTs with potential constraints.

GADTs. The problem of generic programming for GADTs have not received much attention in the literature. The approach of Magalhães and Jeuring [16] is based on pattern functors: the basic set of blocks is extended with *CEq*, which represents equalities at the level of constructors, and a "mobility family" *X* to fake existentials. Our approach reuses most of the machinery for regular types, by taking advantage of the availability of *Constraint* as a kind in GHC. Using quantified class constraints [5], Scott [27] describes how to derive *Generic* for some GADTs. The approach does not scale, though, to handle existentials or kinds different from $*$.

Kind-genericity. Hinze [10, 11] describe an approach to generic operations for different kinds. The leverage the pattern functors (as those used in *GHC.Generics*) to work on different kinds, and define operations indexing by the kind. Our approach is quite different, as we extend the language of atoms in `generic-sop` to cover different kinds.

Weirich and Casinghino [30] define arity-generic operations, such as the family of functions *zipWith*, *zipWith3*, and so on. In their case the operations are indexed by a natural number which specifies the amount of arguments; in contrast our development is indexed by contexts Γ which specify the kind of each type variable.

8 Conclusion and Future Work

Although we greatly exapended the set of types that the (generic) programmer has access to, this is still not exhaustive. With the introduction of the *TypeInType* extension, quantification in types works as a telescope. That is, the kind of a variable might depend on the variables introduced before it. For example, here t depends on the kind ζ:

```
data KTProxy ζ t where
  KTProxy :: forall ζ (t :: ζ) . KTProxy ζ t
```

The *Exists* combinator in our library only allows *constant* kinds to quantify over.

Another shortcoming of our library is that only *single* recursion can be represented. The `multirec` [35] library adds support families of mutually recursive datatypes, and `generic-mrsop` [21] translated its approach to the sum-of-products style. However, in both cases all members of a datatype are restricted to be of kind $*$. It might be possible to use a similar technique — adding an index to the *Rec* atom – to try tackling this. The difficulty is in the possibility of different members of the family having different kinds.

Through several refinements, starting with the original sum-of-products construction, we have built a generic programming library supporting a wider range of datatypes. Our main novelties are the uniform treatment of different kinds, and the support for the most important features in GADTs. To do so, we have leveraged many of the Haskell extensions proposed in the literature and available in GHC.

References

[1] Thorsten Altenkirch, Conor McBride, and Peter Morris. 2007. Generic Programming with Dependent Types. In *Proceedings of the 2006 International Conference on Datatype-generic Programming (SSDGP'06)*. Springer-Verlag, Berlin, Heidelberg. http://dl.acm.org/citation.cfm?id=1782894.1782898

[2] Arthur Baars, S. Doaitse Swierstra, and Marcos Viera. 2010. Typed Transformations of Typed Grammars: The Left Corner Transform. *Electronic Notes in Theoretical Computer Science* 253, 7 (2010). https://doi.org/10.1016/j.entcs.2010.08.031 Proceedings of the Ninth Workshop on Language Descriptions Tools and Applications.

[3] Baldur Blöndal, Andres Lö"h, and Ryan Scott. 2018. Deriving Via. In *Proceedings of the 11th ACM Haskell Symposium (Haskell '18)*.

[4] Max Bolingbroke. 2011. Constraint Kinds for GHC. http://blog.omega-prime.co.uk/?p=127 Blog post.

[5] Gert-Jan Bottu, Georgios Karachalias, Tom Schrijvers, Bruno C. d. S. Oliveira, and Philip Wadler. 2017. Quantified Class Constraints. In *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell (Haskell 2017)*. ACM, New York, NY, USA. https://doi.org/10.1145/3122955.3122967

[6] Edsko de Vries and Andres Löh. 2014. True Sums of Products. In *Proceedings of the 10th ACM SIGPLAN Workshop on Generic Programming (WGP '14)*. ACM, New York, NY, USA. https://doi.org/10.1145/2633628.2633634

[7] Richard A. Eisenberg and Stephanie Weirich. 2012. Dependently Typed Programming with Singletons. In *Proceedings of the 2012 Haskell Symposium (Haskell '12)*. ACM, New York, NY, USA. https://doi.org/10.1145/2364506.2364522

[8] Richard A. Eisenberg, Stephanie Weirich, and Hamidhasan G. Ahmed. 2016. Visible Type Application. In *Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings (Lecture Notes in Computer Science)*, Peter Thiemann (Ed.), Vol. 9632. Springer.

[9] Jean-Yves Girard. 1972. *Interpretation fonctionnelle et elimination des coupures de l'arithmetique d'ordre superieur*. Ph.D. Dissertation.

[10] Ralf Hinze. 2000. A New Approach to Generic Functional Programming. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '00)*. ACM, New York, NY, USA. https://doi.org/10.1145/325694.325709

[11] Ralf Hinze. 2000. Polytypic Values Possess Polykinded Types. In *Mathematics of Program Construction*. Springer Berlin Heidelberg.

[12] Ralf Hinze and Johan Jeuring. 2003. *Generic Haskell: Applications*. Springer Berlin Heidelberg, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-45191-4_2

[13] Ralf Lämmel and Simon Peyton Jones. 2003. 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 (TLDI '03)*. ACM, New York, NY, USA. https://doi.org/10.1145/604174.604179

[14] Eelco Lempsink, Sean Leather, and Andres Löh. 2009. Type-safe Diff for Families of Datatypes. In *Proceedings of the 2009 ACM SIGPLAN Workshop on Generic Programming (WGP '09)*. ACM, New York, NY, USA. https://doi.org/10.1145/1596614.1596624

[15] José Pedro Magalhães, Atze Dijkstra, Johan Jeuring, and Andres Löh. 2010. A Generic Deriving Mechanism for Haskell. In *Proceedings of the Third ACM Haskell Symposium (Haskell '10)*. ACM, New York, NY, USA. https://doi.org/10.1145/1863523.1863529

[16] José Pedro Magalhães and Johan Jeuring. 2011. Generic Programming for Indexed Datatypes. In *Proceedings of the Seventh ACM SIGPLAN Workshop on Generic Programming (WGP '11)*. ACM, New York, NY, USA. https://doi.org/10.1145/2036918.2036924

[17] José Pedro Magalhães and Andres Löh. 2012. A Formal Comparison of Approaches to Datatype-Generic Programming. In *Proceedings Fourth Workshop on Mathematically Structured Functional Programming, Tallinn, Estonia, 25 March 2012*, James Chapman and Paul Blain Levy (Eds.). https://doi.org/10.4204/EPTCS.76.6

[18] José Pedro Magalhães and Andres Löh. 2014. Generic Generic Programming. In *Practical Aspects of Declarative Languages*, Matthew Flatt and Hai-Feng Guo (Eds.).

[19] Simon Marlow et al. 2010. Haskell 2010 Language Report. https://www.haskell.org/onlinereport/haskell2010/.

[20] Victor Cacciari Miraldo, Pierre-Évariste Dagand, and Wouter Swierstra. 2017. Type-directed Diffing of Structured Data. In *Proceedings of the 2nd ACM SIGPLAN Workshop on Type-Driven Development (TyDe 2017)*. ACM, New York, NY, USA. https://doi.org/10.1145/3122975.3122976

[21] Victor Cacciari Miraldo and Alejandro Serrano. 2018. Sums of Products for Mutually Recursive Datatypes. In *Proceedings of the 3nd ACM SIGPLAN Workshop on Type-Driven Development (TyDe 2018)*.

[22] Neil Mitchell and Colin Runciman. 2007. Uniform Boilerplate and List Processing. In *Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop (Haskell '07)*. ACM, New York, NY, USA. https://doi.org/10.1145/1291201.1291208

[23] Thomas van Noort, Alexey Rodriguez, Stefan Holdermans, Johan Jeuring, and Bastiaan Heeren. 2008. A Lightweight Approach to Datatype-generic Rewriting. In *Proceedings of the ACM SIGPLAN Workshop on Generic Programming (WGP '08)*. ACM, New York, NY, USA. https://doi.org/10.1145/1411318.1411321

[24] Ulf Norell. 2009. Dependently typed programming in Agda. In *Proceedings of TLDI'09: 2009 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Savannah, GA, USA, January 24, 2009*. https://doi.org/10.1145/1481861.1481862

[25] Simon Peyton Jones, Stephanie Weirich, Richard A. Eisenberg, and Dimitrios Vytiniotis. 2016. A Reflection on Types. In *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*.

[26] Alexey Rodriguez, Johan Jeuring, Patrik Jansson, Alex Gerdes, Oleg Kiselyov, and Bruno C. d. S. Oliveira. 2008. Comparing Libraries for Generic Programming in Haskell. In *Proceedings of the First ACM SIGPLAN Symposium on Haskell (Haskell '08)*. ACM, New York, NY, USA. https://doi.org/10.1145/1411286.1411301

[27] Ryan Scott. 2018. How to derive Generic for (some) GADTs using QuantifiedConstraints. Blog post, available at https://ryanglscott.github.io/2018/02/11/how-to-derive-generic-for-some-gadts/.

[28] Alejandro Serrano and Jurriaan Hage. 2016. Generic Matching of Tree Regular Expressions over Haskell Data Types. In *Practical Aspects of Declarative Languages - 18th International Symposium, PADL 2016, St. Petersburg, FL, USA, January 18-19, 2016. Proceedings*. https://doi.org/10.1007/978-3-319-28228-2_6

[29] Tim Sheard and Simon Peyton Jones. 2002. Template meta-programming for Haskell. https://www.microsoft.com/en-us/research/publication/template-meta-programming-for-haskell/

[30] Stephanie Weirich and Chris Casinghino. 2010. Arity-generic Datatype-generic Programming. In *Proceedings of the 4th ACM SIGPLAN Workshop on Programming Languages Meets Program Verification (PLPV '10)*. ACM, New York, NY, USA. https://doi.org/10.1145/1707790.1707799

[31] Stephanie Weirich, Justin Hsu, and Richard A. Eisenberg. 2013. System FC with Explicit Kind Equality. *SIGPLAN Not.* 48, 9 (Sept. 2013). https://doi.org/10.1145/2544174.2500599

[32] Stephanie Weirich, Antoine Voizard, Pedro Henrique Azevedo de Amorim, and Richard A. Eisenberg. 2017. A Specification for Dependent Types in Haskell. *Proc. ACM Program. Lang.* 1, ICFP (Aug. 2017). https://doi.org/10.1145/3110275

[33] Thomas Winant, Dominique Devriese, Frank Piessens, and Tom Schrijvers. 2014. Partial Type Signatures for Haskell. In *Practical Aspects of Declarative Languages*, Matthew Flatt and Hai-Feng Guo (Eds.).

[34] Hongwei Xi, Chiyan Chen, and Gang Chen. 2003. Guarded Recursive Datatype Constructors. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '03)*. ACM, New York, NY, USA. https://doi.org/10.1145/604131.604150

[35] Alexey Rodriguez Yakushev, Stefan Holdermans, Andres Löh, and Johan Jeuring. 2009. Generic Programming with Fixed Points for Mutually Recursive Datatypes. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP '09)*. ACM, New York, NY, USA. https://doi.org/10.1145/1596550.1596585

[36] Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. 2012. Giving Haskell a Promotion. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI '12)*. ACM, New York, NY, USA. https://doi.org/10.1145/2103786.2103795

A Reference Implementation in Agda

In this section we shall describe the Agda model of *GenericsNSOP*. Agda [24] is a depentently typed language with a predicative hierarchy of universes. This allows us to model our construction with a finer level of detail. Ultimately showing one does not need the *Set*:*Set* axiom to make the construction work.

The starting point for our construction is modelling the notion of kind, as we aim to encode arbitrarily kinded types. Here we mimick Haskell's kind syntax.

```
data 𝕂 : Set where
  ⋆     : 𝕂
  _⇒_ : 𝕂 → 𝕂 → 𝕂
```

Although their semantics are simple, we already need to start mapping $\mathbb{K}$ to Set_1, since we want to have ground types in *Set*:

```
⟦_⟧𝕂 : 𝕂 → Set₁
⟦⋆⟧𝕂         = Set
⟦k₁ ⇒ k₂⟧𝕂 = ⟦k₁⟧𝕂 → ⟦k₂⟧𝕂
```

In order to refer to the kind of a type variable within a larger kind declaration we declare the *TyVar* relation below. Essentially, an inhabitant of type *TyVar ks k* provides a way to extract some kind *k* from within *ks*.

```
data TyVar : 𝕂 → 𝕂 → Set where
  VZ : ∀ {k ks}                      → TyVar (k ⇒ ks) k
  VS : ∀ {k k′ ks} → TyVar ks k → TyVar (k′ ⇒ ks) k
```

Next, we can define the language of terms and contexts, just like we did in Section 4:

```
data Atom (ζ : 𝕂) : 𝕂 → Set₁ where
  Var : ∀ {k₁} → TyVar ζ k₁ → Atom ζ k₁
  Kon : ∀ {k₁} → ⟦k₁⟧𝕂 → Atom ζ k₁
  App : ∀ {k₁ k₂} → Atom ζ (k₁ ⇒ k₂) → Atom ζ k₁
      → Atom ζ k₂
data Γ : 𝕂 → Set₁ where
  GZ : Γ ⋆
  GS : ∀ {k₁ k₂} → ⟦k₁⟧𝕂 → Γ k₂ → Γ (k₁ ⇒ k₂)
```

Now, given a environment γ, we can interpret a term t : *Atom* ζ k into an inhabitant of $[\![k]\!]_{\mathbb{K}}$.

```
Ty : ∀ {res k} → Γ k → Atom k res → ⟦res⟧𝕂
Ty GZ        (Var ())
Ty (GS g γ) (Var VZ)      = γ
Ty (GS g γ) (Var (VS v)) = Ty γ (Var v)
Ty γ          (Kon x)       = x
Ty γ          (App f x)    = Ty γ f (Ty γ x)
```

Note how we can discharge the case of interpreting a variable in an empty environment. Finally, we are now able to define the fields of the constructors. These come in two flavours. Explicit fields are values of some ground type whereas implicit fields are constraints over the type parameters in scope.

```
data Field (k : 𝕂) : Set₂ where
  Explicit : Atom k ⋆        → Field k
  Implicit : (Γ k → Set₁) → Field k
```

The most interesting part of the model is, in fact, the handling of constraints. A constraint is a predicate over the types that will be in scope when type checking that constructor. These types are the interpretation of some kind k, $[\![k]\!]_{\mathbb{K}}$, and hence, inhabitants of Set_1. This forces us to map Γ k into Set_1, which brings the universe of *Field* into Set_2. Interpreting a *Field* can be done back in Set_1 again:

```
⟦_⟧A : ∀ {k} → Field k → Γ k → Set₁
⟦Explicit t⟧A    γ = Lift (Ty γ t)
⟦Implicit ctr⟧A γ = ctr γ
```

Where *Lift* : *Set* → Set_1 lifts an inhabitant of a smaller universe into a bigger one. The rest of the model is trivial. We define a product of kind k and a sum of kind k as lists, and interpret them using *All* and *Any*, respectively.

```
Prod SoP : 𝕂 → Set₂
Prod k = List (Field k)
SoP   k = List (Prod k)

⟦_⟧P : ∀ {k} → Prod k → Γ k → Set₂
⟦α⟧P γ = All (λα → ⟦α⟧A γ) α

⟦_⟧S : ∀ {k} → Prod k → Γ k → Set₂
⟦ps⟧S γ = Any (λπ → ⟦π⟧P γ) ps
```

Finally, going full circle and encoding the example shown at the introduction would look like:

```
data IsNat : Set → Set where
  Prf : Nat → IsNat Nat

isnatSOP : SoP (⋆ ⇒ ⋆)
isnatSOP = (Implicit ctr :: Explicit (Var VZ) :: [ ]) :: [ ]
  where
    ctr : Γ (⋆ ⇒ ⋆) → Set₁
    ctr (x :: [ ]) = x ≡ Nat
```

Adding an extra constructor to *Atom* to mark, explicitely, which are the recursive positions is quite simple. The constructor would have type:

```
Rec : Term k k
```

And we would need one extra parameter of type $[\![k]\!]_{\mathbb{K}}$ in the interpretation functions. Again, as long as we do not take the least fixpoint of this construction, there is no need for the *Set* : *Set* axiom.

B Explicit Recursion

Up until now, we have not distinguished recursive positions from regular fields in our datatypes. This is also the case in *Generic* and $\mathit{Generic}^{\mathrm{sop}}$, where recursion is *implicit*.

Marking recursion *explicitly* is advantageous but introduces a more intricate design. It enables one to write combinators exploiting recursion schemes, such as fold, but it introduces some extra complexity to the atoms of the universe and one extra parameter to the interpretation of codes. In fact, some generic operations require this explicit recursion information. Such as the definition of diff and patch [14, 20], zippers [12], and tree regular expressions [28].

The technique of marking recursive positions [23, 35] starts with expanding the atoms with a new building block:

```
data Rec p = Rec p
```

Although isomorphic to *I*, *Rec* serves quite a different purpose. Nevertheless, the second step is to bubble up one extra parameter to the interpretation of codes, lifting it to $* \to *$. The *from* function would then have a type similar to:

```
from :: a → Rep a a
```

Passing *a* as this extra parameter closes the recursive knot.

Let us now apply the same technique to our scenario. We start by extending the *Atom* type with a new constructor:

```
data Atom (ζ :: Kind) k where
  ...
  Rec :: Atom ζ ζ
```

The kind of a recursive occurence is exactly the kind of whatever datatype we are defining. As an example, we can provide a more informative code for [ ], where the recursion is explicit:

```
type ListCode = '['[ ], '[V₀, Rec :@: V₀]]
```

The next step is to extend the interpretation of atoms to include this new construction. As in the case of Noort et al. [23], we only tie the recursive knot at the level of the $\mathit{Generic}^{\mathrm{sop}}_{\star}$ type class. In the meantime, *Ty* is extended with a new argument, which declares which is the type to be used whenever *Rec* is found.

```
type family Ty (ζ :: Kind) (r :: ζ) (α :: Γ ζ)
               (t :: Atom ζ k) :: k where
  ...
  Ty ζ r α Rec = r
```

As a consequence, *NA* and $\mathit{SOP}_{\star}$ also gain a new type parameter for this recursive position.

```
data NA (ζ :: Kind) :: ζ → Γ ζ → Field ζ → ∗ where ...
type SOP⋆ ζ (c :: DataType ζ) (r :: ζ) (α :: Γ ζ)
  = NS (NP (NA ζ r α)) c
```

Finally, the updated $\mathit{Generic}^{\mathrm{sop}}_{\star}$ mandates this recursive position to be instantiated with the datatype we are describing, tying the knot. This approach is similar to the $\mathit{Generic}^{\mathrm{regular}}$ type class.

```
class Generic⋆^sop ζ (f :: ζ) where
  type Code f :: DataType ζ
  to   :: ApplyT ζ f α → SOP⋆ ζ (Code f) f α
  from :: SForΓ ζ α
       ⇒ SOP⋆ ζ (Code f) f α → ApplyT ζ f α
```

Since the constructors in *NA* do not change depending on whether we have used *Rec* or not to describe the datatype, the instances we provided for the previous version of $\mathit{Generic}^{\mathrm{sop}}_{\star}$ keep working in the version with explicit recursion.

It is important to note that marking recursive positions explicitly is still sound, as demonstrated by the Agda model in Appendix A. Unfolding this recursion and taking the least fixed point of a type is not, however. That is, we cannot write a *Fix* type, in the lines of:

```
data Fix f = Fix (f (Fix f))
```

That is because the interpretation of sums lives in the second predicative universe (Set_2), which forces *Fix* to live in Set_2 aswell. However, the argument we have to pass to the interpretation of must be an inhabitant of Set_1, hence we cannot feed *Fix f* back into *f*. This would require the *Set*:*Set* axiom, breaking consistency. Hence, just marking the positions is fine, unfolding the recursion is where we would find problems.

***Updating** gfmap.* In Figure 4 we used an ancillary *FunctorAtom* type class to describe which fields we could map over. It is impossible, though, to write an instance of this form:

```
instance (FunctorAtom x) ⇒ FunctorAtom (Rec :@: x)
```

In fact, *gfmap* defines the operation for the type we are recurring over, so this instance ought to exist! In order to convince the compiler, we play the same trick as before: work with *r* as an independent entity, and only tie the knot at the level of *gfmap*. We need to pass the function which works on the recursive position as an additional argument.

```
class FunctorAtom r (t :: Atom (∗ → ∗) (∗)) where
  gfmapF :: (forall x y . (x → y) → r x → r y)
         → (a → b)
         → NA (∗ → ∗) r (a :&: ϵ) t
         → NA (∗ → ∗) r (b :&: ϵ) t

instance (FunctorAtom r x)
         ⇒ FunctorAtom r (Rec :@: x) where
  gfmapF r f (T x)
    = T (r (unT ∘ gfmapF r f ∘ T @_@x@r) x)
```

The trick now is to make the *FunctorAtom* constraint used in All_2 refer to the same *f* as in the code. And to close the loop, when we call *gfmapF*, we pass *gfmap* itself as the function to execute in the when *Rec* is found.

```
gfmap :: (Generic★sop (∗ → ∗) f,
          AllD (FunctorAtom f) (Code f))
       ⇒ (a → b) → f a → f b
gfmap f = ...
  where
    goP Nil          = Nil
    goP (T x :∗ xs) = gfmapF gfmap f (T x) :∗ goP xs
```

This approach makes the definition of *gfmap* self-contained, in contrast to the definition without explicit recursion, in which a *Functor* instance needs to be written to make recursion possible.

B.1 Explicit Recursion and Existentials

The combination of existentials with explicit recursion is not as straightforward as the combination of constraints from Section 5 with explicit recursion. In this section we outline the changes required to bring both concepts under the same umbrella.

For the language of codes the introduction of *Rec* does not stop the original implementation from compiling. But we also need to update the interpretation datatype *NB*, by adding an additional argument for the recursive position:

```
data NB (ζ :: Kind) :: ζ → Γ ζ → Branch ζ → ∗ where
  Ex :: forall ℓ (t :: ℓ) (p :: SKind ℓ) ζ α r c .
         NB (ℓ → ζ)?(t :&: α) c → NB ζ r α (Exists p c)
  Cr :: NP (NA ζ r α) fs       → NB ζ r α (Constr fs)
```

If we do not introduce any new variables – the case of *Constr* – we can pass down to NA the same type for the recursive position. On the other hand, the case of *Exists* forces the argument to *Ex* to have kind $\ell \to \zeta$. But r has kind ζ instead.

The problem is that we are using the kind ζ for two different tasks. On the one hand, ζ fixes the shape of the context Γ. On the other hand, ζ specified which is the kind obtained when recursion is performed, that is, when *Rec* is used as part of an atom. When one uses *Exists* the additional type should only be introduced in the context, whereas the recursive positions should stay as they were.

The solution is to decouple this two modes of use of ζ. An atom should take *two* kind arguments: ρ specifies the kind of recursive positions, and ζ specifies the kinds in the context.

```
data Atom (ρ :: Kind) (ζ :: Kind) k where
  Var   :: TyVar ζ k → Atom ρ ζ k
  Kon   :: k          → Atom ρ ζ k
  (:@:) :: Atom ρ ζ (k1 → k2) → Atom ρ ζ k1
                      → Atom ρ ζ k2
  Rec   ::              Atom ρ ζ ρ
```

This change generates a chain reaction of updates (we refer the reader to Appendix D for the complete code), the most important being in *NB*:

```
data NB (ρ :: Kind) (ζ :: Kind)
    :: ρ → Γ ζ → Branch ρ ζ → ∗ where
  Ex :: forall ℓ (t :: ℓ) (p :: SKind ℓ) ρ ζ r α c .
          NB ρ (ℓ → ζ) r (t :&: α) c
                              → NB ρ ζ r α (Exists p c)
  Cr :: NP (NA ρ ζ r α) fs → NB ρ ζ r α (Constr fs)
```

Note how *only* the context argument is extended from ζ to $\ell \to \zeta$. The kind of the recursive position is kept as ρ.

The fact that ζ and ρ coincide before any existential is introduced is made explicit in the updated definitions of *DataType* and *SOP*★:

```
type DataType ζ = [Branch ζ ζ]
type SOP★ ζ (c :: DataType ζ) (r :: ζ) (α :: Γ ζ)
   = NS (NB ζ ζ r α) c
```

In both cases the single argument which represents the kind of the datatype to describe is used as kind for recursion and context.

C Arity-generic *fmap*

The generic *fmap* described in Section 4.1 can be generalized to a version which works on any type constructor whose type variables are all of kind ∗. The full code is given in Figure 7, with some auxiliary definitions in Figures 8 and 9.

As we have already discussed, the mapping operation *kmap* requires a different amount of functions to apply over each type parameter depending on the kind of the type constructor. The *Mappings* datatype defines such a structure by recursion over two contexts α and β which define the source and target types.

The body of the generic implementation *gkmap* recurses over the structure of a generic value. The first difference with the implementation of *gfmap* – the generic *fmap* for one-argument type constructors we developed in Section 4.1 – is that we do not only handle explicit fields in *goP*, but also implicit values. Take the following datatype:

```
data Showy a where
  Showable    :: Show a ⇒ a → Showy a
  NotShowable :: String   → a → Showy a
```

If we want to perform a shape-preserving map, we need to ensure that the target type argument satisfies the constraints imposed by the different constructors. In this case, the type of the map should be:

```
showyMap :: Show b ⇒ (a → b) → Showy a → Showy b
```

The *ISatisfiedD* type family in Figure 9 inspects the code of a datatype building up the set of such constraints. The body of the family is mostly recurring over the code; the interesting bit is in the case of an *Implicit* field, for which we reify the description of the type of the constraint by using *Ty*.

Most of the magic happens in *KFunctorField*, which takes care of applying the mappings to each field in the datatype.

```
data Mappings (α :: Γ ζ) (β :: Γ ζ) where
  MNil  ::                         Mappings ϵ        ϵ
  MCons :: (a → b) → Mappings α β → Mappings (a :&: α) (b :&: β)

class KFunctor ζ (f :: ζ) where
  kmap :: SForΓ ζ β ⇒ Mappings α β → ApplyT ζ f α → ApplyT ζ f β
  default kmap  ::  (Generic^sop_⋆ ζ f, SForΓ ζ β, AllD KFunctorField (Code f), ISatisfiedD ζ β (Code f))
                ⇒ Mappings α β → ApplyT ζ f α → ApplyT ζ f β
  kmap fs = to ∘ gkmap (Proxy :: Proxy f) fs ∘ from

gkmap  ::  forall ζ (f :: ζ) (α :: Γ ζ) (β :: Γ ζ) .
           (Generic^sop_⋆ ζ f, AllD KFunctorField (Code f), ISatisfiedD ζ β (Code f))
        ⇒ Proxy f → Mappings α β → SOP_⋆ ζ (Code f) α → SOP_⋆ ζ (Code f) β
gkmap _ f = goS
  where
    goS :: (AllD KFunctorField xs, ISatisfiedD k β xs) ⇒ NS (NB k α) xs → NS (NB k β) xs
    goS (Here  x)   = Here  (goB x)
    goS (There x)   = There (goS x)
    goB :: (AllB KFunctorField xs, ISatisfiedB k β xs) ⇒ NB k α xs → NB k β xs
    goB (Cr x)      = Cr (goP x)
    goP :: (AllE KFunctorField xs, ISatisfiedE k β xs) ⇒ NP (NA k α) xs → NP (NA k β) xs
    goP Nil          = Nil
    goP (E x :∗ xs) = kmapf f (E x) :∗ goP xs
    goP (I   :∗ xs) = I :∗ goP xs

class KFunctorField (t :: Atom ζ ∗) where
  kmapf :: Mappings α β → NA ζ α (Explicit t) → NA ζ β (Explicit t)
instance forall ζ (v :: TyVar ζ Type) . SForTyVar k v ⇒ KFunctorField (Var v) where
  kmapf f (E x) = E (go (styvar@ζ@v) f x)
    where go :: forall k (α :: Γ k) (β :: Γ k) (v :: TyVar k ∗) . STyVar k v → Mappings α β → Ty k α (Var v) → Ty k β (Var v)
          go SVZ      (MCons g _)  x = g x
          go (SVS v′) (MCons _ f′) x = go v′ f′ x
instance KFunctorField (Kon t) where
  kmapf f (E x) = E x
instance forall f x . (KFunctorHead f, KFunctorField x) ⇒ KFunctorField (f :@: x) where
  kmapf f (E x) = E $ unA_0 $ unA^+
                  $ kmaph (Proxy :: Proxy f) f (MCons (unE ∘ kmapf f ∘ E @_@x) MNil)
                  $ A^+ $ A_0 x

class KFunctorHead (t :: Atom ζ k) where
  kmaph  ::  SForΓ k τ ⇒ Proxy t → Mappings α β → Mappings ρ τ
         → ApplyT k (Ty ζ α t) ρ → ApplyT k (Ty ζ β t) τ
instance forall f x . (KFunctorHead f, KFunctorField x) ⇒ KFunctorHead (f :@: x) where
  kmaph _ f r x = unA^+ $ kmaph (Proxy :: Proxy f) f (MCons (unE ∘ kmapf f ∘ E @_@x) r) $ A^+ x
instance forall k (f :: k) . (KFunctor k f) ⇒ KFunctorHead (Kon f) where
  kmaph _ _ r x = kmap r x
```

Figure 7. Arity-generic map *kmap* and associated type class *KFunctor*

```
data STyVar k (t :: TyVar k ∗) where
  SVZ ::                STyVar (∗ → k) VZ
  SVS  :: STyVar k v → STyVar (∗ → k) (VS v)
class SForTyVar k (t :: TyVar k ∗) where
  styvar :: STyVar k t
instance SForTyVar (∗ → k) VZ where
  styvar = SVZ
instance SForTyVar k v ⇒ SForTyVar (∗ → k) (VS v) where
  styvar = SVS styvar
```

Figure 8. Auxiliary singleton for *TyVar*

```
type family AllD c xs :: Constraint where
  AllD c '[ ]                = ()
  AllD c (x ': xs)           = (AllB c x, AllD c xs)
type family AllB c xs :: Constraint where
  AllB  c (Constr x)         = AllE c x
type family AllE c xs :: Constraint where
  AllE  c '[ ]               = ()
  AllE  c (Explicit x ': xs) = (c x, AllE c xs)
  AllE  c (Implicit x ': xs) =       AllE c xs
type family ISatisfiedD ζ (α :: Γ ζ) xs :: Constraint where
  ISatisfiedD ζ α '[ ] = ()
  ISatisfiedD ζ α (x ': xs)
    = (ISatisfiedB ζ α x, ISatisfied ζ α xs)
type family ISatisfiedB ζ (α :: Γ ζ) xs :: Constraint where
  ISatisfiedB ζ α (Constr x) = ISatisfiedE ζ α x
type family ISatisfiedE ζ (α :: Γ ζ) xs :: Constraint where
  ISatisfiedE ζ α '[ ] = ()
  ISatisfiedE ζ α (Implicit x ': xs)
    = (Ty ζ α x, ISatisfiedE ζ α xs)
  ISatisfiedE ζ α (Explicit x ': xs)
    =              ISatisfiedE ζ α xs
```

Figure 9. Auxiliary type families for arity-generic *fmap*

The *Kon* case is the simplest one, as no change has to take place. The case of a type variable is more complicated, as we need to lookup the function to apply in the list of mappings. Since the de Bruijn index of the variable only exists at type level, but we need different run-time behavior depending on it, we are require to introduce a singleton, which we do in Figure 8. Once we obtain the singleton using *styvar*, we traverse it in sync with the list of mappings. The indices in both cases ensure that we can only apply the correct mapping from the ones available.

Consider now the case of an application of a type constructor to one or more types. For example, in the second field of *Fork*,

```
data Rose a = Fork a [Rose a]
```

To implement map over *Rose*, we need to call the map operation on [ ]. However, we cannot maintain the same list of mappings, since we need to map *Rose a* to *Rose b*. As a consequence, when we find a type application in a field, we need to build the *new* list of mappings, which is then fed to the type constructor on the head of the application.

Such an algorithm is implemented by the *KFunctorHead* type class in Figure 7. Note that the *kmaph* operation take not one, but *two* lists of mappings. The first one refers to the original context α and β, the second one accumulates the new mappings for the type arguments ρ and τ. If we find an application we attach a new mapping – note the use of *MCons* –, if we are already at the end of the application we recursively apply *kmap*. There is no case for a head being a variable, as we assume that the datatype for which we are defining *kmap* has only $*$-kinded arguments.

D Full Implementation

Throughout the paper we have refined the different types and classes involved in our generic programming library. We give the end result as a reference, with support for constraints, existentials, and explicit recursion. Figure 10 describes the language of codes, Figure 11 the datatypes involved in the interpretation of the codes, Figure 12 the $Generic^{sop}_{\star}$ type class and ancillary constructions, and Figure 13 the conversion between applied types and the evidence-carrying *ApplyT*.

D.1 List of Extensions

For reference, we list here the language extensions that must be enabled in GHC version 8.4.1.

- For the core construction we require *TypeInType*, *GADTs*, *DataKinds*, *TypeFamilies*, *ScopedTypeVariables*; and the following extensions to type classes: *MultiParamTypeClasses*, *InstanceSigs*, *FlexibleContexts*, and *FlexibleInstances*.
- Support for constraints: *ConstraintKinds*.
- Support for explicit recursion: *RankNTypes*.
- To refer to the kind (∗) explicitly: *ExplicitNamespaces*.

There are also some language extensions whose use is not essential to the construction, but are required to compile the code as given:

- *TypeOperators* is required to use (:@:), (:&:), and (:∗), as constructor names.
- *TypeApplications* to fix the types of some uses of *T*. We could have used *Proxy* values instead, but this approach is clearer.

```
data TyVar (ζ :: Kind) k where
  VZ ::                 TyVar (x → xs) x
  VS :: TyVar xs k → TyVar (x → xs) k
data Atom (ρ :: Kind) (ζ :: Kind) k where
  Var   :: TyVar ζ k → Atom ρ ζ k
  Kon   :: k          → Atom ρ ζ k
  Rec   ::              Atom ρ ζ ρ
  (:@:) :: Atom ζ (k1 → k2) → Atom ζ k1 → Atom ζ k2
data Field (ρ :: Kind) (ζ :: Kind) where
  Explicit :: Atom ρ ζ (∗)        → Field ρ ζ
  Implicit :: Atom ρ ζ Constraint → Field ρ ζ
data SKind (ℓ :: Kind) = K
data Branch (ρ :: Kind) (ζ :: Kind) where
  Exists  :: SKind ℓ → Branch ρ (ℓ → ζ) → Branch ρ ζ
  Constr :: [Field ρ ζ ]                  → Branch ρ ζ
type DataType ζ = [Branch ζ ζ ]
```

Figure 10. Full implementation, part 1: language of codes

```
data Γ (ζ :: Kind) where
  ϵ     ::                Γ (∗)
  (:&:) :: k → Γ ks → Γ (k → ks)
type family Ty (ρ :: Kind) (ζ :: Kind) (r :: ρ) (α :: Γ ζ) (t :: Atom ρ ζ k) :: k where
  Ty ρ (k → ks) r (t :&: α) (Var VZ)      = t
  Ty ρ (k → ks) r (t :&: α) (Var (VS v)) = Ty ρ ks r α (Var v)
  Ty ρ ζ        r α         (Kon t)       = t
  Ty ρ ζ        r α         (f :@: x)     = (Ty ρ ζ r α f) (Ty ρ ζ r α x)
  Ty ρ ζ        r α         Rec           = r
data NA (ρ :: Kind) (ζ :: Kind) :: ρ → Γ ζ → Field ζ → ∗ where
  E :: forall ρ ζ t r α .   {unE :: Ty ρ ζ r α t} → NA ρ ζ r α (Explicit t)
  I :: forall ρ ζ t r α .          Ty ρ ζ r α t  ⇒ NA ρ ζ r α (Implicit t)
data NP :: (k → ∗) → [k] → ∗ where
  Nil ::                  NP f '[ ]
  (:∗) :: f x → NP f xs → NP f (x ': xs)
data NB (ρ :: Kind) (ζ :: Kind) :: ρ → Γ ζ → Branch ζ → ∗ where
  Ex :: forall ℓ (t :: ℓ) (p :: SKind ℓ) ρ ζ r α c . NB ρ (ℓ → ζ) r (t :&: α) c → NB ρ ζ r α (Exists p c)
  Cr ::                                              NP (NA ρ ζ r α) fs          → NB ρ ζ r α (Constr fs)
data NS :: (k → ∗) → [k] → ∗ where
  Here  :: f k      → NS f (k ': ks)
  There :: NS f ks → NS f (k ': ks)
type SOP★ (ζ :: Kind) (c :: DataType ζ) (r :: ζ) (α :: Γ ζ) = NS (NB ζ ζ r α) c
```

Figure 11. Full implementation, part 2: interpretation of codes

```
data SΓ ζ (α :: Γ ζ) where
  Sϵ  ::                SΓ (∗)          ϵ
  S& :: SΓ ks τ → SΓ (k → ks) (t :&: τ)
class SForΓ k (α :: Γ k) where
  sΓ :: SΓ k α
instance SForΓ (∗) ϵ where
  sΓ = Sϵ
instance SForΓ ks τ ⇒ SForΓ (k → ks) (t :&: τ) where
  sΓ = S& sΓ
data ApplyT k (f :: k) (α :: Γ k) :: ∗ where
  A_0 :: { unA_0 :: f } → ApplyT (∗)          f ϵ
  A^+ :: { unA^+ :: ApplyT ks (f t) τ }
                         → ApplyT (k → ks) f (t :&: τ)
class Generic_⋆^sop ζ (f :: ζ) where
  type Code f :: DataType ζ
  from  ::  ApplyT ζ f α → SOP_⋆ ζ (Code f) f α
  to     ::  SForΓ ζ α
        ⇒ SOP_⋆ ζ (Code f) f α → ApplyT ζ f α
```

Figure 12. Full implementation, part 3: $Generic^{sop}_{\star}$ type class

```
type family Apply ζ (f :: ζ) (α :: Γ ζ) :: ∗ where
  Apply (∗)          f ϵ          = f
  Apply (k → ks) f (t :&: τ) = Apply ks (f t) τ
unravel :: ApplyT k f α → Apply k f α
unravel (A_0 x) = x
unravel (A^+ x) = unravel x
ravel  ::  forall k f α . SForΓ k α
       ⇒ Apply k f α → ApplyT k f α
ravel = go (sΓ @_@α)
  where
    go  ::  forall k f α . SΓ k α
         → Apply k f α → ApplyT k f α
    go Sϵ          x = A_0 x
    go (S& τ)      x = A^+ (go τ x)
```

Figure 13. Full implementation, part 4: utility functions