Sums of Products for Mutually Recursive Datatypes

The Appropriationist's View on Generic Programming

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

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

Abstract

Generic programming for mutually recursive families of datatypes is hard. On the other hand, most interesting abstract syntax trees are described by a mutually recursive family of datatypes. We could give up on using that mutually recursive structure, but then we lose the ability to use those generic operations which take advantage of that same structure. We present a new approach to generic programming that uses modern Haskell features to handle mutually recursive families with explicit *sum-of-products* structure. This additional structure allows us to remove much of the complexity previously associated with generic programming over these types.

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

Keywords Generic Programming, Datatype, Haskell

ACM Reference Format:
Victor Cacciari Miraldo and Alejandro Serrano. 2018. Sums of Products for Mutually Recursive Datatypes: The Appropriationist's View on Generic Programming. In *Proceedings of ACM SIGPLAN Conference on Programming Languages (TyDe'18).* ACM, New York, NY, USA, **??** pages.

1 Introduction

(Datatype-)generic programming provides a mechanism to write functions by induction on the structure of algebraic datatypes [**?**]. A well-known example is the **deriving** mechanism in Haskell, which frees the programmer from writing repetitive functions such as equality [**?**]. A vast range of approaches are available as preprocessors, language extensions, or libraries for Haskell [**? ?**]. In **??** we outline the main design differences between a few of those libraries.

The core idea underlying generic programming is the fact that a great number of datatypes can be described in a uniform fashion. Consider the following datatype representing binary trees with data stored in their leaves:

```
data Bin a = Leaf a | Bin (Bin a) (Bin a)
```

A value of type $Bin\ a$ consists of a choice between two constructors. For the first choice, it also contains a value of type a whereas for the second it contains two subtrees as children. This means that the $Bin\ a$ type is isomorphic to $Either\ a\ (Bin\ a, Bin\ a)$.

Different libraries differ on how they define their underlying generic descriptions. For example, `GHC.Generics` [**?**] defines the representation of Bin as the following datatype:

```
Rep (Bin a) = K1 R a :+: (K1 R (Bin a) :*: K1 R (Bin a))
```

which is a direct translation of $Either\ a\ (Bin\ a, Bin\ a)$, but using the combinators provided by `GHC.Generics`, namely `:+:` and `:*:`. In addition, we need two conversion functions $from :: a \to Rep\ a$ and $to :: Rep\ a \to a$ which form an isomorphism between $Bin\ a$ and $Rep\ (Bin\ a)$. All this information is tied to the original datatype using a type class:

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

Most generic programming libraries follow a similar pattern of defining the *description* of a datatype in the provided uniform language by some type level information, and two functions witnessing an isomorphism. A important feature of such library is how this description is encoded and which are the primitive operations for constructing such encodings, as we shall explore in **??**. Some libraries, mainly deriving from the `SYB` approach [**? ?**], use the $Data$ and $Typeable$ type classes instead of static type level information to provide generic functionality. These are a completely different strand of work from ours.

?? shows the main libraries relying on type level representations. In the *pattern functor* approach we have `GHC.Generics` [**?**

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

	Pattern Functors	Codes
No Explicit Recursion	`GHC.Generics`	`generics-sop`
Simple Recursion	`regular`	**`generics-mrsop`**
Mutual Recursion	`multirec`	

Figure 1. Spectrum of static generic programming libraries

], being the most commonly used one, that effectively replaced `regular` [**?**]. The former does not account for recursion explicitly, allowing only for a *shallow* representation, whereas the later allows for both *deep* and *shallow* representations by maintaining information about the recursive occurrences of a type. Maintaining this information is central to some generic functions, such as the generic *map* and *Zipper*, for instance. Oftentimes though, one actually needs more than just one recursive type, justifying the need to `multirec` [**?**].

These libraries are too permissive though, for instance, *K1 R Int* `:*:` *Maybe* is a perfectly valid `GHC.Generics` *pattern functor* but will break generic functions, i.e., *Maybe* is not a combinator. The way to fix this is to ensure that the *pattern functors* abide by a certain format, by defining them by induction on some *code* that can be inspected and matched on. This is the approach of `generics-sop` [**?**]. The more restrictive code approach allows one to write concise, combinator-based, generic programs. The novelty in our work is in the intersection of both the expressivity of `multirec`, allowing the encoding of mutually recursive families, with the convenience of the more modern `generics-sop` style. In fact, it is worth noting that neither of the aforementioned libraries *compete* with out work. We extend both in orthogonal directions, resulting in a new design altogether, that takes advantage of some modern Haskell extensions that the authors of the previous work could not enjoy.

1.1 Contributions

In this paper we make the following contributions:

- We extend the sum-of-products approach of **?**] to care for recursion (**??**), allowing for *shallow* and *deep* representations. We proceed generalizing even further to mutually recursive families of datatypes (**??**).
- We illustrate the use of our library on familiar examples such as equality, α-equivalence (**??**) and the zipper (**??**), illustrating how it subsumes the features of the previous approaches.
- We provide Template Haskell functionality to derive all the boilerplate code needed to use our library (in **??**, due to space restrictions). The novelty lies in our handling of instantiated type constructors.

We have packaged our results as a Haskell library. This library, `generics-mrsop`, fills the hole in **??** for a code-based approach with support for mutual recursion.

1.2 Design Space

The availability of several libraries for generic programming witnesses the fact that there are trade-offs between expressivity, ease of use, and underlying techniques in the design of such a library. In this section we describe some of these trade-offs, especially those to consider when using the static approach.

Explicit Recursion. There are two ways to define the representation of values. Those that have information about which fields of the constructors of the datatype in question are recursive versus those that do not.

If we do not mark recursion explicitly, *shallow* encodings are our sole option, where only one layer of the value is turned into a generic form by a call to *from*. This is the kind of representation we get from `GHC.Generics`, among others. The other side of the spectrum would be the *deep* representation, in which the entire value is turned into the representation that the generic library provides in one go.

Marking the recursion explicitly, like in `regular` [**?**], allows one to choose between *shallow* and *deep* encodings at will. These representations are usually more involved as they need an extra mechanism to represent recursion. In the *Bin* example, the description of the *Bin* constructor changes from "this constructor has two fields of the *Bin a* type" to "this constructor has two fields in which you recurse". Therefore, a *deep* encoding requires some explicit *least fixpoint* combinator – usually called *Fix* in Haskell.

Depending on the use case, a shallow representation might be more efficient if only part of the value needs to be inspected. On the other hand, deep representations are sometimes easier to use, since the conversion is performed in one go, and afterwards one only has to work with the constructs from the generic library.

The fact that we mark explicitly when recursion takes place in a datatype gives some additional insight into the description. Some functions really need the information about which fields of a constructor are recursive and which are not, like the generic *map* and the generic *Zipper* – we describe the latter in **??**. This additional power has also been used to define regular expressions over Haskell datatypes [**?**].

Sum of Products Most generic programming libraries build their type level descriptions out of three basic combinators: (1) *constants*, which indicate a type is atomic and should not be expanded further; (2) *products* (usually written as `:*:`) which are used to build tuples; and (3) *sums* (usually written as `:+:`) which encode the choice between constructors. *Rep* (*Bin a*) above is expressed in this form. Note, however, that there is no restriction on *how* these can be combined.

In practice, one can always use a sum of products to represent a datatype – a sum to express the choice of constructor, and within each constructor a product to declare which fields you have. The `generic-sop` library [**?**] explicitly uses a list

of lists of types, the outer one representing the sum and each inner one thought of as products. The $'$ sign in the code below marks the list as operating at the type level, as opposed to term-level lists which exist at run-time. This is an example of Haskell's *datatype* promotion [**?**].

$$Code_{\text{sop}}\ (Bin\ a) = '['[a], '[Bin\ a, Bin\ a]]$$

The shape of this description follows more closely the shape of Haskell datatypes, and make it easier to implement generic functionality.

Note how the *codes* are different than the *representation*. The latter being defined by induction on the former. This is quite a subtle point and it is common to see both terms being used interchangeably. Here, the *representation* is mapping the *codes*, of kind $'['[*]]$, into $*$. The *code* can be seen as the format that the *representation* must adhere to. Previously, in the pattern functor approach, the *representation* was not guaranteed to have a certain structure. The expressivity of the language of *codes* is proportional to the expressivity of the combinators the library can provide.

Mutually recursive datatypes. We have described several axes taken by different approaches to generic programming in Haskell. Unfortunately, most of the approaches restrict themselves to *regular* types, in which recursion always goes into the *same* datatype, which is the one being defined. Sometimes one would like to have the mutually recursive structure handy, though. The syntax of many programming languages, for instance, is expressed naturally using a mutually recursive family. Consider Haskell itself, one of the possibilities of an expression is to be a **do** block, while a **do** block itself is composed by a list of statements which may include expressions.

```
data Expr = ... | Do [Stmt] | ...
data Stmt = Assign Var Expr | Let Var Expr
```

Another example is found in HTML and XML documents. They are better described by a Rose tree, which can be described by this family of datatypes:

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

The mutual recursion becomes apparent once one instantiaties a to some ground type, for instance:

```
data RoseI = Fork Int ListI
data ListI = Nil | RoseI:ListI
```

The `multirec` library [**?**] is a generalization of `regular` which handles mutually recursive families using this very technique. The mutual recursion is central to some applications such as generic diffing [**?**] of abstract syntax trees.

The motivation of our work stems from the desire of having the concise structure that *codes* give to the *representations*, together with the information for recursive positions in a mutually recursive setting.

Deriving the representation. Generic programming alleviates the problem of repetitively writing operations such as equality or pretty-printing, which depend on the structure of the datatype. But in order to do so, they still require the programmer to figure out the right description and write conversion functions *from* and *to* that type. This is tedious, and also follows the shape of the type!

For that reason, most generic programming libraries also include some automatic way of generating this boilerplate code. `GHC.Generics` is embedded in the compiler; most others use Template Haskell [**?**], the meta-programming facility found in GHC. In the former case, programmers write:

```
data Bin a = ...deriving Generic
```

to open the doors to generic functionality.

There is an interesting problem that arises when we have mutually recursive datatypes and want to automatically generate descriptions. The definition of *Rose a* above uses the list type, but not simply [*a*] for any element type *a*, but the specific instance [*Rose a*]. This means that the procedure to derive the code must take this fact into account. Shallow descriptions do not suffer too much from this problem. For deep approaches, though, how to solve this problem is key to derive a useful description of the datatype.

2 Background

Before diving head first into our generic programming framework, let us take a tour of the existing generic programming libraries. For that, will be looking at a generic *size* function from a few different angles, illustrating how different techniques relate and the nuances between them. This will let us gradually build up to our framework, that borrows pieces of each of the different approaches, and combines them without compromise.

2.1 GHC Generics

Since version 7.2, GHC supports some off the shelf generic programming using `GHC.Generics` [**?**], which exposes the *pattern functor* of a datatype. This allows one to define a function for a datatype by induction on the structure of its (shallow) representation using *pattern functors*.

These *pattern functors* are parametrized versions of tuples, sum types (*Either* in Haskell lingo), and unit, empty and constant functors. These provide a unified view over data: the generic *representation* of values. The values of a suitable type a are translated to this representation by means of the function $from_{\text{gen}} :: a \to Rep_{\text{gen}}\ a$. Note that the subscripts are there solely to disambiguate names that appear in many libraries. Hence, $from_{\text{gen}}$ is, in fact, the *from* in module *GHC.Generics*.

Defining a generic function is done in two steps. First, we define a class that exposes the function for arbitrary types, in our case, *size*, which we implement for any type via *gsize*:

```
  size (Bin (Leaf 1) (Leaf 2))
= gsize (from_gen (Bin (Leaf 1) (Leaf 2)))
= gsize (R1 (K1 (Leaf 1) :*: K1 (Leaf 2)))
= gsize (K1 (Leaf 1)) + gsize (K1 (Leaf 2))
†= size (Leaf 1) + size (Leaf 2)
= gsize (from_gen (Leaf 1)) + gsize (from_gen (Leaf 2))
= gsize (L1 (K1 1)) + gsize (L1 (K1 2))
= size (1 :: Int) + size (2 :: Int)
```

Figure 2. Reduction of *size* (*Bin* (*Leaf* 1) (*Leaf* 2))

```
class Size (a :: *) where
  size :: a → Int
instance (Size a) ⇒ Size (Bin a) where
  size = gsize ∘ from_gen
```

Next we define the *gsize* function that operates on the level of the representation of datatypes. We have to use another class and the instance mechanism to encode a definition by induction on representations:

```
class GSize (rep :: * → *) where
  gsize :: rep x → Int
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
```

We still have to handle the cases where we might have an arbitrary type in a position, modeled by the constant functor *K1*. We require an instance of *Size* so we can successfully tie the recursive knot.

```
instance (Size x) ⇒ GSize (K1 R x) where
  gsize (K1 x) = size x
```

To finish the description of the generic *size*, we also need instances for the *unit*, *void* and *metadata* pattern functors, called *U1*, *V1*, and *M1* respectively. Their *GSize* is rather uninteresting, so we omit them for the sake of conciseness.

This technique of *mutually recursive classes* is quite specific to `GHC.Generics` flavor of generic programming. **??** illustrates how the compiler goes about choosing instances for computing *size* (*Bin* (*Leaf* 1) (*Leaf* 2)). In the end, we just need an instance for *Size Int* to compute the final result. Literals of type *Int* illustrate what we call *opaque types*: those types that constitute the base of the universe and are *opaque* to the representation language.

One interesting aspect we should note here is the clearly *shallow* encoding that *from* provides. That is, we only represent *one layer* at a time. For example, take the step marked as (†) in **??**: after unwrapping the calculation of the first *layer*, we are back to having to calculate *size* for *Bin Int*, not their generic representation.

Upon reflecting on the generic *size* function above, we see a number of issues. Most notably is the amount of boilerplate to achieve a conceptually simple task: sum up all the sizes of the fields of whichever constructors make up the value. This is a direct consequence of not having access to the *sum-of-products* structure that Haskell's **data** declarations follow. A second issue is that the generic representation does not carry any information about the recursive structure of the type. The `regular` [**?**] library addresses this issue by having a specific *pattern functor* for recursive positions.

Perhaps even more subtle, but also more worrying, is that we have no guarantees that the $Rep_{gen}\ a$ of a type *a* will be defined using only the supported *pattern functors*. Fixing this would require one to pin down a single language for representations, that is, the *code* of the datatype. Besides correctness issues, having *codes* greatly improves the definition of *ad-hoc* generic combinators. Every generic function has to follow the *mutually recursive classes* technique we shown.

2.2 Explicit Sums of Products

We will now examine the approach used by **?**]. The main difference is in the introduction of *Codes*, that limit the structure of representations.

Had we had access to a representation of the *sum-of-products* structure of *Bin*, we could have defined our *gsize* function following an informal description: sum up the sizes of the fields inside a value, ignoring the constructor.

Unlike `GHC.Generics`, the representation of values is defined by induction on the *code* of a datatype, this *code* is a type level list of lists of kind ∗, whose semantics is consonant to a formula in disjunctive normal form. The outer list is interpreted as a sum and each of the inner lists as a product. This section provides an overview of `generic-sop` as required to understand our techniques, we refer the reader to the original paper [**?**] for a more comprehensive explanation.

Using a *sum-of-products* approach one could write the *gsize* function as easily as:

```
gsize :: (Generic_sop a) ⇒ a → Int
gsize = sum ∘ elim (map size) ∘ from_sop
```

Ignoring the details of *gsize* for a moment, let us focus just on its high level structure. Remembering that *from* now returns a *sum-of-products* view over the data, we are using an eliminator, *elim*, to apply a function to the fields of the constructor used to create a value of type *a*. This eliminator then applies *map size* to the fields of the constructor, returning something akin to a [*Int*]. We then *sum* them up to obtain the final size.

Codes consist of a type level list of lists. The outer list represents the constructors of a type, and will be interpreted as a sum, whereas the inner lists are interpreted as the fields of the respective constructors, interpreted as products.

```
type family   Code_sop (a :: *) :: '[ '[*]]
type instance Code_sop (Bin a) = '[ '[a], '[Bin a, Bin a]]
```

The *representation* is then defined by induction on $Code_{\text{sop}}$ by the means of generalized n-ary sums, NS, and n-ary products, NP. With a slight abuse of notation, one can view NS and NP through the lens of the following type isomorphisms:

$$NS\ f\ [k_1, k_2, \ldots] \equiv f\ k_1 :+: (f\ k_2 :+: \ldots)$$
$$NP\ f\ [k_1, k_2, \ldots] \equiv f\ k_1 :*: (f\ k_2 :*: \ldots)$$

We could then define Rep_{sop} to be $NS\ (NP\ (K1\ R))$, echoing the isomorphisms above, where **data** $K1\ R\ a = K1\ a$ is borrowed from `GHC.Generics`. Note that we already need the parameter f to pass NP to NS here. This is exactly the representation we get from `GHC.Generics`.

$$\begin{aligned} Rep_{\text{sop}}\ (Bin\ a) &\equiv NS\ (NP\ (K1\ R))\ (Code_{\text{sop}}\ (Bin\ a)) \\ &\equiv K1\ R\ a :+: (K1\ R\ (Bin\ a) :*: K1\ R\ (Bin\ a)) \\ &\equiv Rep_{\text{gen}}\ (Bin\ a) \end{aligned}$$

It makes no sense to go through all the trouble of adding the explicit *sums-of-products* structure to forget this information in the representation. Instead of piggybacking on *pattern functors*, we define NS and NP from scratch using *GADTs* [**?**]. By pattern matching on the values of NS and NP we inform the type checker of the structure of $Code_{\text{sop}}$.

```
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
  NP0 ::                    NP f '[]
  (×)  :: f x → NP f xs → NP f (x ': xs)
```

Finally, since our atoms are of kind $*$, we can use the identity functor, I, to interpret those and define the final representation of values of a type a under the *SOP* view:

```
type Rep_sop a = NS (NP I) (Code_sop a)
newtype I (a :: ∗) = I {unI :: a}
```

To support the claim that one can define general combinators for working with these representations, let us look at *elim* and *map*, used to implement the *gsize* function in the beginning of the section. The *elim* function just drops the constructor index and applies f, whereas the *map* applies f to all elements of a product.

```
elim :: (∀ k . f k → a) → NS f ks → a
elim f (Here  x) = f x
elim f (There x) = elim f x
map :: (∀ k . f k → a) → NP f ks → [a]
map f NP0      = []
map f (x × xs) = f x:map f xs
```

Reflecting on the current definition of *size*, especially in comparison to the `GHC.Generics` implementation of *size*, we see two improvements: (A) we need one fewer type class, namely *GSize*, and, (B) the definition is combinator-based. Considering that the generated *pattern functor* representation of a Haskell datatype will already be in a *sums-of-products*, we do not lose anything by enforcing this structure.

There are still downsides to this approach. A notable one is the need to carry constraints around: the actual *gsize* written with the `generics-sop` library and no sugar reads as follows.

```
gsize :: (Generic_sop a, All2 Size (Code_sop a)) ⇒ a → Int
gsize = sum ∘ hcollapse
      ∘ hcmap (Proxy :: Proxy Size) (mapIK size) ∘ from_sop
```

Where *hcollapse* and *hcmap* are analogous to the *elim* and *map* combinators we defined above. The $All2\ Size\ (Code_{\text{sop}}\ a)$ constraint tells the compiler that all of the types serving as atoms for $Code_{\text{sop}}\ a$ are an instance of *Size*. In our case, $All2\ Size\ (Code_{\text{sop}}\ (Bin\ a))$ expands to $(Size\ a, Size\ (Bin\ a))$. The *Size* constraint also has to be passed around with a *Proxy* for the eliminator of the n-ary sum. This is a direct consequence of a *shallow* encoding: since we only unfold one layer of recursion at a time, we have to carry proofs that the recursive arguments can also be translated to a generic representation. We can relieve this burden by recording, explicitly, which fields of a constructor are recursive or not.

3 Explicit Fix: Diving Deep and Shallow

In this section we will start to look at our approach, essentially combining the techniques from the `regular` and `generics-sop` libraries. Later we extend the constructions to handle mutually recursive families instead of simple recursion. As we discussed in the introduction, a fixpoint view over generic functionality is required to implement some functionality like the *Zipper* generically. In other words, we need an explicit description of which fields of a constructor are recursive and which are not.

Introducing information about the recursive positions in a type requires more expressive codes than in **??**, where our *codes* were a list of lists of types, which could be anything. Instead, we will now have a list of lists of *Atom* to be our codes:

```
data Atom = I | KInt | ...
type family   Code_fix (a :: ∗) :: '['[Atom]]
type instance Code_fix (Bin Int) = '['[KInt], '[I, I]]
```

Where I is used to mark the recursive positions and $KInt, \ldots$ are codes for a predetermined selection of primitive types, which we refer to as *opaque types*. Favoring the simplicity of the presentation, we will stick with only hard coded *Int* as the only opaque type in the universe. Later on, in **??**, we parametrize the whole development by the choice of opaque types.

We can no longer represent polymorphic types in this universe – the *codes* themselves are not polymorphic. Back in **??** we have defined $Code_{\text{sop}}\ (Bin\ a)$, and this would work for any a. This might seem like a disadvantage at first, but it is in fact the opposite. This allows us to provide a deep conversion

for free and drops the need to carry constraints around. Beyond doubt one needs to have access to the $Code_{sop}\ a$ when converting a $Bin\ a$ to its deep representation. By specifying the types involved beforehand, we are able to get by without having to carry all of the constraints we needed, for instance, for *gsize* at the end of **??**. We can benefit the most from this in the simplicity of combinators we are able to write, as shown in **??**.

Wrapping our to_{fix} and $from_{fix}$ isomorphism into a type class and writing the instance that witnesses that $Bin\ Int$ has a $Code_{fix}$ is straightforward. We ommit the to_{fix} function as it is the opposite of $from_{fix}$:

```
class Generic_fix a where
  from_fix :: a → Rep_fix a a
  to_fix   :: Rep_fix a a → a
instance Generic_fix (Bin Int) where
  from_fix (Leaf x)
    = Rep (        Here (NA_K x × NP0))
  from_fix (Bin l r)
    = Rep (There (Here (NA_I l  × NA_I r × NP0)))
```

In order to define Rep_{fix} we just need a way to map an $Atom$ into $*$. Since an atom can be either an opaque type, known statically, or some other type that will be used as a recursive position later on, we simply receive it as another parameter. The NA datatype relates an $Atom$ to its semantics:

```
data NA :: * → Atom → * where
  NA_I :: x   → NA x I
  NA_K :: Int → NA x KInt
newtype Rep_fix a x
          = Rep { unRep :: NS (NP (NA x)) (Code_fix a) }
```

It is an interesting exercise to implement the $Functor$ instance for $(Rep_{fix}\ a)$. We were only able to lift it to a functor by recording the information about the recursive positions. Otherwise, there would be no way to know where to apply f when defining $fmap\ f$.

Nevertheless, working directly with Rep_{fix} is hard – we need to pattern match on $There$ and $Here$, whereas we actually want to have the notion of *constructor* for the generic setting too! The main advantage of the *sum-of-products* structure is to allow a user to pattern match on generic representations just like they would on values of the original type, contrasting with `GHC.Generics`. One can precisely state that a value of a representation is composed by a choice of constructor and its respective product of fields by the $View$ type.

```
data Nat = Z | S Nat
data View :: [[Atom]] → * → * where
  Tag :: Constr n t → NP (NA x) (Lkup t n) → View t x
```

A value of $Constr\ \ n\ \ sum$ is a proof that n is a valid constructor for sum, stating that $n < length\ sum$. $Lkup$ performs list lookup at the type level. In order to improve type error messages, we generate a $TypeError$ whenever we reach a given index n that is out of bounds. Interestingly, our design guarantees that this case is never reached by $Constr$.

```
data Constr :: Nat → [k] → * where
  CZ ::                Constr Z     (x:xs)
  CS :: Constr n xs → Constr (S n) (x:xs)
type family Lkup (ls :: [k]) (n :: Nat) :: k where
  Lkup '[]    _     = TypeError "Index out of bounds"
  Lkup (x:xs) 'Z    = x
  Lkup (x:xs) ('S n) = Lkup xs n
```

Now we are able to easily pattern match and inject into and from generic values. Unfortunately, matching on Tag requires describing in full detail the shape of the generic value using the elements of $Constr$. Using pattern synonyms [**?**] we can define those patterns once and for all, and give them more descriptive names. For example, here are the synonyms describing the constructors Bin and $Leaf$. [1]

```
pattern Leaf x   = Tag CZ       (NA_K x × NP0)
pattern Bin  l r = Tag (CS CZ) (NA_I l × NA_I r × NP0)
```

The functions that perform the pattern matching and injection are the *inj* and *sop* below.

```
inj :: View  sop x → Rep_fix sop x
sop :: Rep_fix sop x → View  sop x
```

The $View$ type and the hability to split a value into a choice of constructor and its fields is very handy for writing generic functions, as we can see in **??**.

Having the core of the *sums-of-products* universe defined, we can turn our attention to writing the combinators that the programmer will use. These will be defined by induction on the $Code_{fix}$ instead of having to rely on instances, like in **??**. For instance, lets look at *compos*, which applies a function f everywhere on the recursive structure.

```
compos :: (Generic_fix a) ⇒ (a → a) → a → a
compos f = to_fix ∘ fmap f ∘ from_fix
```

Although more interesting in the mutually recursive setting, **??**, we can illustrate its use for traversing a tree and adding one to its leaves. This example is a bit convoluted, since one could get the same result by simply writing $fmap\ (+1) :: Bin\ Int \to Bin\ Int$, but shows the intended usage of the *compos* combinator just defined.

```
example :: Bin Int → Bin Int
example (Leaf n) = Leaf (n + 1)
example x        = compos example x
```

It is worth noting the *catch-all* case, allowing one to focus only on the interesting patterns and using a default implementation everywhere else.

Converting to a deep representation. The $from_{fix}$ function returns a shallow representation. But by constructing the

[1] Throughout this paper we use the syntax $\overline{C}$ to refer to the pattern describing a view for constructor C.

```
crush :: (Generic_fix a)
      ⇒ (∀ x . Int → b) → ([b] → b)
      → a → b
crush k cat = crushFix ∘ deepFrom
  where
    crushFix :: Fix (Rep_fix a) → b
    crushFix = cat ∘ elimNS (elimNP go) ∘ unFix
    go (NA_I x) = crushFix x
    go (NA_K i) = k i
```

Figure 3. Generic *crush* combinator

least fixpoint of $Rep_{fix}\ a$ we can easily obtain the deep encoding for free, by simply recursively translating each layer of the shallow encoding.

```
newtype Fix f = Fix {unFix :: f (Fix f)}
deepFrom :: (Generic_fix a) ⇒ a → Fix (Rep_fix a)
deepFrom = Fix ∘ fmap deepFrom ∘ from_fix
```

So far, we handle the same class of types as the `regular` [**??**] library, but we are imposing the representation to follow a *sum-of-products* structure by the means of $Code_{fix}$. Those types are guaranteed to have an initial algebra, and indeed, the generic fold is defined as expected:

```
fold :: (Rep_fix a b → b) → Fix (Rep_fix a) → b
fold f = f ∘ fmap (fold f) ∘ unFix
```

Sometimes we actually want to consume a value and produce a single value, but do not need the full expressivity of *fold*. Instead, if we know how to consume the opaque types and combine those results, we can consume any $Generic_{fix}$ type using *crush*, which is defined in **??**. The behavior of *crush* is defined by (1) how to turn atoms into the output type b – in this case we only have integer atoms, and thus we require an $Int \to b$ function – and (2) how to combine the values bubbling up from each member of a product. Finally, we come full circle to our running *gsize* example as it was promised in the introduction. This is noticeably the smallest implementation so far, and very straight to the point.

```
gsize :: (Generic_fix a) ⇒ a → Int
gsize = crush (const 1) sum
```

Let us take a step back and reflect upon what we have achieved so far. We have combined the insight from the `regular` library of keeping track of recursive positions with the convenience of the `generics-sop` for enforcing a specific *normal form* on representations. By doing so, we were able to provide a *deep* encoding for free. This essentially frees us from the burden of maintaining complicated constraints needed for handling the types within the topmost constructor. The information about the recursive position allows us to write neat combinators like *crush* and *compos* together with a convenient *View* type for easy generic pattern matching. The only thing keeping us from handling real life applications is the limited form of recursion. When a user requires a generic programming library, chances are they need to traverse and consume mutually recursive structures.

4 Mutual Recursion

Conceptually, going from regular types (**??**) to mutually recursive families is simple. We just need to be able to reference not only one type variable, but one for each element in the family. This is usually [**? ?**] done by adding an index to the recursive positions that represents which member of the family we are recursing over. As a running example, we use the *rose tree* family from the introduction.

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

The previously introduced $Code_{fix}$ is not expressive enough to describe this datatype. In particular, when we try to write $Code_{fix}\ (Rose\ Int)$, there is no immediately recursive appearance of *Rose* itself, so we cannot use the atom I in that position. Furthermore $[Rose\ a]$ is not an opaque type either, so we cannot use any of the other combinators provided by *Atom*. We would like to record information about $[Rose\ Int]$ referring to itself via another datatype.

Our solution is to move from codes of datatypes to *codes for families of datatypes*. We no longer talk about $Code_{fix}\ (Rose\ Int)$ or $Code_{fix}\ [Rose\ Int]$ in isolation. Codes only make sense within a family, that is, a list of types. Hence, we talk about $Code_{mrec}\ '[Rose\ Int, [Rose\ Int]]$. That is, the codes of the two types in the family. Then we extend the language of *Atom*s by appending to I a natural number which specifies the member of the family to recurse into:

```
data Atom = I Nat | KInt | ...
```

The code of this recursive family of datatypes can finally be described as:

```
type FamRose          = '[Rose Int, [Rose Int]]
type Code_mrec FamRose = '['['[KInt, I (S Z)]]
                          ,'['[], '[I Z, I (S Z)]]]
```

Let us have a closer look at the code for *Rose Int*, which appears in the first place in the list. There is only one constructor which has an *Int* field, represented by *KInt*, and another in which we recurse via the second member of our family (since lists are 0-indexed, we represent this by $S\ Z$). Similarly, the second constructor of $[Rose\ Int]$ points back to both *Rose Int* using $I\ Z$ and to $[Rose\ Int]$ itself via $I\ (S\ Z)$.

Having settled on the definition of *Atom*, we now need to adapt *NA* to the new *Atom*s. In order to interpret any *Atom* into $*$, we now need a way to interpret the different recursive positions. This information is given by an additional type parameter φ that maps natural numbers into types.

```
data NA :: (Nat → *) → Atom → * where
  NA_I :: φ n → NA φ (I n)
  NA_K :: Int → NA φ KInt
```

This additional φ naturally bubbles up to Rep_{mrec}.

```
type Rep_mrec (φ :: Nat → ∗) (c :: [[Atom]])
     = NS (NP (NA φ)) c
```

The only piece missing here is tying the recursive knot. If we want our representation to describe a family of datatypes, the obvious choice for $\varphi\ n$ is to look up the type at index n in *FamRose*. In fact, we are simply performing a type level lookup in the family, so we can reuse the *Lkup* from **??**.

In principle, this is enough to provide a ground representation for the family of types. Let *fam* be a family of types, like $'[Rose\ Int, [Rose\ Int]]$, and *codes* the corresponding list of codes. Then the representation of the type at index *ix* in the list *fam* is given by:

```
Rep_mrec (Lkup fam) (Lkup codes ix)
```

This definition states that to obtain the representation of the type at index *ix*, we first lookup its code. Then, in the recursive positions we interpret each $I\ n$ by looking up the type at that index in the original family. This gives us a *shallow* representation. As an example, below is the expansion for index 0 of the rose tree family. Note how it is isomorphic to the representation that `GHC.Generics` would have chosen for *Rose Int*:

```
Rep_mrec (Lkup FamRose) (Lkup (Code_mrec FamRose) Z)
  = Rep_mrec (Lkup FamRose)        '['[KInt, I (S Z)]]
  = NS (NP (NA (Lkup FamRose))) '['[KInt, I (S Z)]]
  ≡ K1 R Int :∗: K1 R (Lkup FamRose (S Z))
  = K1 R Int :∗: K1 R [Rose Int]
  = Rep_gen (Rose Int)
```

Unfortunately, Haskell only allows saturated, that is, fully-applied type families. Hence, we cannot partially apply *Lkup* like we did it in the example above. As a result, we need to introduce an intermediate datatype *El*,

```
data El :: [∗] → Nat → ∗ where
  El :: Lkup fam ix → El fam ix
```

The representation of the family *fam* at index *ix* is thus given by $Rep_{\text{mrec}}\ (El\ fam)\ (Lkup\ codes\ ix)$. We only need to use *El* in the first argument, because that is the position in which we require partial application. The second position has *Lkup* already fully-applied, and can stay as is.

We still have to relate a family of types to their respective codes. As in other generic programming approaches, we want to make their relation explicit. The *Family* type class below realizes this relation, and introduces functions to perform the conversion between our representation and the actual types. Using *El* here spares us from using a proxy for *fam* in $from_{\text{mrec}}$ and to_{mrec}:

```
class Family (fam :: [∗]) (codes :: [[[Atom]]]) where
  from_mrec :: SNat ix
            → El fam ix → Rep_mrec (El fam) (Lkup codes ix)
  to_mrec   :: SNat ix
            → Rep_mrec (El fam) (Lkup codes ix) → El fam ix
```

One of the differences between other approaches and ours is that we do not use an associated type to define the *codes* for the family *fam*. One of the reasons to choose this path is that it alleviates the burden of writing the longer $Code_{\text{mrec}}\ fam$ every time we want to refer to *codes*. Furthermore, there are types like lists which appear in many different families, and in that case it makes sense to speak about a relation instead of a function. In any case, we can choose the other point of the design space by moving *codes* into an associated type or introduce a functional dependency $fam \to codes$.

Since now $from_{\text{mrec}}$ and to_{mrec} operate on families, we have to specify how to translate *each* of the members of the family back and forth the generic representation. This translation needs to know which is the index of the datatype we are converting between in each case, hence the additional $SNat\ ix$ parameter. Pattern matching on this singleton [**?**] type informs the compiler about the shape of the *Nat* index. Its definition is:

```
data SNat (n :: Nat) where
  SZ ::          SNat 'Z
  SS :: SNat n → SNat ('S n)
```

For example, in the case of our family of rose trees, $from_{\text{mrec}}$ has the following shape:

```
from_mrec SZ      (El (Fork x ch))
  = Rep (         Here (NA_K x × NA_I ch × NP0))
from_mrec (SS SZ) (El [])
  = Rep (         Here NP0))
from_mrec (SS SZ) (El (x:xs))
  = Rep (There (Here (NA_I x × NA_I xs × NP0)))
```

By pattern matching on the index, the compiler knows which family member to expect as a second argument. This then allows the pattern matching on the *El* to typecheck.

The limitations of the Haskell type system lead us to introduce *El* as an intermediate datatype. Our $from_{\text{mrec}}$ function does not take a member of the family directly, but an *El*-wrapped one. However, to construct that value, *El* needs to know its parameters, which amounts to the family we are embedding our type into and the index in that family. Those values are not immediately obvious, but we can use Haskell's visible type application [**?**] to work around it. The *into* function injects a value into the corresponding *El*:

```
into :: ∀ fam ty ix . (ix ~ Idx ty fam, Lkup fam ix ~ ty)
     ⇒ ty → El fam ix
into = El
```

where *Idx* is a closed type family implementing the inverse of *Lkup*, that is, obtaining the index of the type *ty* in the list *fam*. Using this function we can turn a $[Rose\ Int]$ into its generic representation by writing $from_{\text{mrec}} \circ into\ @FamRose$. The type application *@FamRose* is responsible for fixing the mutually recursive family we are working with, which allows the type checker to reduce all the constraints and happily inject the element into *El*.

Deep representation. In **??** we have described a technique to derive deep representations from shallow representations. We can play a very similar trick here. The main difference is the definition of the least fixpoint combinator, which receives an extra parameter of kind *Nat* indicating which *code* to use first:

```
newtype Fix (codes :: [[[Atom]]]) (ix :: Nat)
  = Fix { unFix :: Rep_mrec (Fix codes) (Lkup codes ix) }
```

Intuitively, since now we can recurse on different positions, we need to keep track of the representations for all those positions in the type. This is the job of the *codes* argument. Furthermore, our *Fix* does not represent a single datatype, but rather the *whole* family. Thus, we need each value to have an additional index to declare on which element of the family it is working on.

As in the previous section, we can obtain the deep representation by iteratively applying the shallow representation. Earlier we used *fmap* since the Rep_{fix} type was a functor. Rep_{mrec} on the other hand cannot be given a *Functor* instance, but we can still define a similar function *mapRec*,

```
mapRep :: (∀ ix . φ1 ix → φ2 ix)
       → Rep_mrec φ1 c → Rep_mrec φ2 c
```

This signature tells us that if we want to change the φ_1 argument in the representation, we need to provide a natural transformation from φ_1 to φ_2, that is, a function which works over each possible index this φ_1 can take and does not change this index. This follows from φ_1 having kind $Nat \to *$.

```
deepFrom :: Family fam codes
         ⇒ El fam ix → Fix (Rep_mrec codes ix)
deepFrom = Fix ∘ mapRec deepFrom ∘ from_mrec
```

Only well-formed representations are accepted. At first glance, it may seem like the *Atom* datatype gives too much freedom: its *I* constructor receives a natural number, but there is no apparent static check that this number refers to an actual member of the recursive family we are describing. For example, the list of codes `'['['[KInt, I (S (S Z))]]]` is accepted by the compiler although it does not represent any family of datatypes.

A direct solution to this problem is to introduce yet another index, this time in the *Atom* datatype, which specifies which indices are allowed. The *I* constructor is then refined to take not any natural number, but only those which lie in the range – this is usually known as *Fin n*.

```
data Atom (n :: Nat) = I (Fin n) | KInt | ...
```

The lack of dependent types makes this approach very hard, in Haskell. We would need to carry around the inhabitants *Fin n* and define functionality to manipulate them, which is more complex than what meets the eye. This could greatly hinder the usability of the library.

By looking a bit more closely, we find that we are not losing any type-safety by allowing codes which reference an arbitrary number of recursive positions. Users of our library are allowed to write the previous ill-defined code, but when trying to write *values* of the representation of that code, the *Lkup* function detects the out-of-bounds index, raising a type error and preventing the program from compiling.

4.1 Parametrized Opaque Types

Up to this point we have considered *Atom* to include a predetermined selection of *opaque types*, such as *Int*, each of them represented by one of the constructors other than *I*. This is far from ideal, for two conflicting reasons:

1. The choice of opaque types might be too narrow. For example, the user of our library may decide to use *ByteString* in their datatypes. Since that type is not covered by *Atom*, nor by our generic approach, this implies that `generics-mrsop` becomes useless to them.
2. The choice of opaque types might be too wide. If we try to encompass any possible situation, we end up with a huge *Atom* type. But for a specific use case, we might be interested only in *Int*s and *Float*s, so why bother ourselves with possibly ill-formed representations and pattern matches which should never be reached?

Our solution is to *parametrize Atom*, giving programmers the choice of opaque types:

```
data Atom kon = I Nat | K kon
```

For example, if we only want to deal with numeric opaque types, we can write:

```
data NumericK = KInt | KInteger | KFloat
type NumericAtom = Atom NumericK
```

The representation of codes must be updated to reflect the possibility of choosing different sets of opaque types. The *NA* datatype in this final implementation provides two constructors, one per constructor in *Atom*. The *NS* and *NP* datatypes do not require any change.

```
data NA :: (kon → *) → (Nat → *) → Atom kon → * where
  NA_I :: φ n → NA κ φ (I n)
  NA_K :: κ k → NA κ φ (K k)

type Rep_mrec (κ :: kon → *) (φ :: Nat → *) (c :: [[Atom kon]])
     = NS (NP (NA κ φ)) c
```

The NA_K constructor in *NA* makes use of an additional argument κ. The problem is that we are defining the code for the set of opaque types by a specific kind, such as *Numeric* above. On the other hand, values which appear in a field must have a type whose kind is $*$. Thus, we require a mapping from each of the codes to the actual opaque type they represent, this is exactly the *opaque type interpretation* κ. Here is the datatype interpreting *NumericK* into ground types:

```
data NumericI :: NumericK → * where
  IInt     :: Int     → NumericI KInt
  IInteger :: Integer → NumericI KInteger
  IFloat   :: Float   → NumericI KFloat
```

The last piece of our framework which has to be updated to support different sets of opaque types is the *Family* type class, as given in **??**. This type class provides an interesting use case for the new dependent features in Haskell; both κ and *codes* are parametrized by an implicit argument *kon* which represents the set of opaque types.

We stress that the parametrization over opaque types does *not* mean that we can use only closed universes of opaque types. It is possible to provide an *open* representation by choosing $(*)$ – the whole kind of Haskell's ground types – as argument to *Atom*. As a consequence, the interpretation ought to be of kind $* \to *$, as follows:

```
data Value :: * → * where
  Value :: t → Value t
```

In order to use $(*)$ as an argument to a type, we are required to enable the `TypeInType` language extension [**? ?**].

4.2 Combinators

In the remainder of this section we wish to showcase a selection of particularly powerful combinators that are simple to define by exploiting the *sums-of-products* structure coupled with the mutual recursion information. Defining the same combinators in `multirec` would produce much more complicated code. In `GHC.Generics` these are even impossible to write due to the absence of recursion information.

For the sake of fostering intuition instead of worrying about notational overhead, we write values of $Rep_{\text{mrec}}\ \kappa\ \varphi\ c$ just like we would write normal Haskell values. They have the same *sums-of-products* structure anyway. Whenever a function is defined using the $\doteq$ symbol, $C\ x_1\ \ldots\ x_n$ will stand for a value of the corresponding $Rep_{\text{mrec}}\ \kappa\ \varphi\ c$, that is, $There\ (\ldots\ (Here\ (x_1 \times \ldots \times x_n \times NP0)))$. Since each of these $x_1\ \ldots\ x_n$ might be a recursive type or an opaque type, whenever we have two functions f_I and f_K in scope, $f\ x_j$ will denote the application of the correct function for recursive positions, f_I, or opaque types f_K. For example, here is the actual code of the function which maps over a *NA* structure:

$$bimapNA\ f_K\ f_I\ (NA_I\ \ i) = NA_I\ \ (f_I\ \ i)$$
$$bimapNA\ f_K\ f_I\ (NA_K\ k) = NA_K\ (f_K\ k)$$

which following this convention becomes:

$$bimapNA\ f_K\ f_I\ x \doteq f\ x$$

The first obvious combinator which we can write using the sum-of-products structure is *map*. Our $Rep_{\text{mrec}}\ \kappa\ \varphi\ c$ is no longer a regular functor, but a higher bifunctor. In other words, it requires two functions, one for mapping over opaque types and another for mapping over *I* positions.

$$bimapRep\ ::\ (\forall\ k\ .\ \kappa_1\ k \to \kappa_2\ k) \to (\forall\ ix\ .\ \varphi_1\ ix \to \varphi_2\ ix) \to Rep_{\text{mrec}}\ \kappa_1\ \varphi_1\ c \to Rep_{\text{mrec}}\ \kappa_2\ \varphi_2\ c$$
$$bimapRep\ f_K\ f_I\ (C\ x_1\ \ldots\ x_n) \doteq C\ (f\ x_1)\ \ldots\ (f\ x_n)$$

More interesting than a map perhaps is a general eliminator. In order to destruct a $Rep_{\text{mrec}}\ \kappa\ \varphi\ c$ we need a way for eliminating every recursive position or opaque type inside the representation and a way of combining these results.

$$elimRep\ ::\ (\forall\ k\ .\ \kappa\ k \to a) \to (\forall\ ix\ .\ \varphi\ ix \to a) \to ([a] \to b) \to Rep_{\text{mrec}}\ \kappa\ \varphi\ c \to b$$
$$elimRep\ f_K\ f_I\ cat\ (C\ x_1\ \ldots\ x_n) \doteq cat\ [f\ x_1, \ldots, f\ x_n]$$

Being able to eliminate a representation is useful, but it becomes even more useful when we are able to combine the data in different values of the same representation with a *zip* like combinator. Our *zipRep* will attempt to put two values of a representation "side-by-side", as long as they are constructed with the same injection into the *n*-ary sum, *NS*.

```
zipRep :: Rep_mrec κ1 φ1 c → Rep_mrec κ2 φ2 c
       → Maybe (Rep_mrec (κ1 :*: κ2) (φ1 :*: φ2) c)
zipRep (C x1 ... xn) (D y1 ... ym)
  | C ≡ D   ≐ Just (C (x1 :*: y1) ... (xn :*: yn))
              -- if C == D, then also n == m!
  | otherwise ≐ Nothing
```

This definition *zipRep* can be translated to work with an arbitrary $(Alternative\ f)$ instead of *Maybe*. The *compos* combinator, already introduced in **??**, shows up in a yet more expressive form. We are now able to change every subtree of whatever type we choose inside an arbitrary value of the mutually recursive family in question.

$$compos\ ::\ (\forall\ iy\ .\ El\ fam\ iy \to El\ fam\ iy) \to El\ fam\ ix \to El\ fam\ ix$$
$$compos\ f = to_{\text{mrec}} \circ bimapRep\ id\ f \circ from_{\text{mrec}}$$

Defining these combinators in `multirec` is not impossible, but involves a much bigger effort. Everything has to be implemented by the means of type classes and each supported combinator must have one instance.

It is worth noting that although we presented pure versions of these combinators, `generics-mrsop` defines monadic variants of these and suffixes them with a `M`, following the standard Haskell naming convention. We will need these monadic combinators in **??**.

5 Examples

In this section we present two applications of our generic programming approach, namely equality and α-equivalence. Our goal is to show that our approach is at least as powerful as any other comparable library, but brings in the union of their advantages. Even though some examples use a single recursive datatype for the sake of conciseness, those can be readily generalized to mutually recursive families. Another common benchmark for the power of a generic library, zippers, is described in **??** due to lack of space.

There are many other applications for generic programming which greatly benefit from supporting mutual recursion, if not requiring it. One great source of examples consists of operations on abstract syntax trees of realistic languages, such as generic diffing [**?**] or pretty-printing [**?**].

```
class Family (κ :: kon → ∗) (fam :: [∗]) (codes :: [[[Atom kon]]]) where
  from_mrec :: SNat ix → El fam ix → Rep_mrec κ (El fam) (Lkup codes ix)
  to_mrec   :: SNat ix → Rep_mrec κ (El fam) (Lkup codes ix) → El fam ix
```

Figure 4. *Family* type class with support for different opaque types

```
geq :: (Family κ fam codes)
    ⇒ (∀ k . κ k → κ k → Bool)
    → El fam ix → El fam ix → Bool
geq eq_K x y = go (deepFrom x) (deepFrom y)
  where go (Fix x) (Fix y)
          = maybe False (elimRep (uncurry eq_K) (uncurry go) and)
          $ zipRep x y
```

Figure 5. Generic equality

5.1 Equality

As usually done in generic programming papers, we should define generic equality in our own framework. In fact, with `generics-mrsop` we can define a particularly elegant version of generic equality, given in **??**.

Reading through the code we see that we convert both arguments of *geq* to their deep representation, then compare their top level constructor with *zipRep*. If they agree we go through each of their fields calling either the equality on opaque types eq_K or recursing.

5.2 α-Equivalence

A more involved exercise is the definition of *α-equivalence* for a language. In this section we start by showing a straightforward version for the λ-calculus and then move on to a more elaborate language. Although such problem has already been treated using generic programming [**?**], it provides a good example to illustrate our library.

Regardless of the language, determining whether two programs are α-equivalent requires one to focus on the constructors that introduce scoping, declare variables or reference variables. All the other constructors of the language should just combine the recursive results. Let us warm up with untyped λ-calculus:

```
data Term_λ = Var String | Abs String Term_λ | App Term_λ Term_λ
```

Let us explain the process step by step. First, for $t_1, t_2 :: Term_\lambda$ to be α-equivalent, they have to have the constructors on the same positions. Otherwise, they cannot be α-equivalent. Then we check the bound variables: we traverse both terms at the same time and every time we go through a binder, in this case *Abs*, we register a new *rule* saying that the bound variable names are equivalent for the terms under that scope. Whenever we find a reference to a variable, *Var*, we check if the referenced variable is equivalent under the registered *rules* so far.

```
alphaEq :: Term_λ → Term_λ → Bool
alphaEq x y = flip runState [[]]
                (galphaEq (deepFrom x) (deepFrom y))
  where
    galphaEq x y = maybe False (go Term) (zipRep x y)
    step = elimRepM (return ∘ uncurry (≡))
                    -- opaque types have to be equal!
                    (uncurry galphaEq)  -- recursive step
                    (return ∘ and)      -- combine
    go Term_λ x = case sop x of
      Var (v1 :∗: v2) → v1 ≈ v2
      Abs (v1 :∗: v2) (t1 :∗: t2)
        → scoped (addRule v1 v2 ≫ galphaEq t1 t2)
      _ → step x
```

Figure 6. α-equivalence for a λ-calculus

Let us abstract away this book-keeping functionality by the means of a monad with a couple of associated functions. The idea is that monad m will keep track of a stack of scopes, and each scope will register a list of *name-equivalences*. Indeed, this is very close to how one should go about defining equality for *nominal terms* [**?**].

```
class Monad m ⇒ MonadAlphaEq m where
  scoped  :: m a → m a
  addRule :: String → String → m ()
  (≈)     :: String → String → m Bool
```

Running a *scoped* f computation will push a new scope for running f and pop it after f is done. The *addRule* v_1 v_2 function registers an equivalence of v_1 and v_2 in the top of the scope stack. Finally, $v_1 \approx v_2$ is defined by pattern matching on the scope stack. If the stack is empty, then $(\approx)\ v_1\ v_2 = (v_1 \equiv v_2)$. Otherwise, let the stack be $s{:}ss$. We first traverse s gathering the rules referencing either v_1 or v_2. If there are none, we check if $v_1 \approx v_2$ under ss. If there are rules referencing either variable name in the topmost stack, we must ensure there is only one such rule, and it states a name equivalence between v_1 and v_2. The implementation of these functions for *MonadAlphaEq* (*State* [[(*String*, *String*)]]) is available as part of our library.

Returning to our main focus and leaving book-keeping functionality aside, we define in **??** our alpha equivalence decision procedure by encoding what to do for *Var* and *Abs* constructors. The *App* can be eliminated generically.

There is a number of remarks to be made for this example. First, note the application of *zipRep*. If two $Term_\lambda$s are made

```
data Stmt = SAssign String Exp
          | SIf    Exp  Stmt Stmt
          | SSeq   Stmt Stmt
          | SReturn Exp
          | SDecl  Decl
          | SSkip
data Decl = DVar String
          | DFun String String Stmt
data Exp  = EVar  String
          | ECall String Exp
          | EAdd Exp  Exp
          | ESub Exp  Exp
          | ELit  Int
```

```
go Stmt x = case sop x of
  SAssign (v1 :*: v2) (e1 :*: e2)        → addRule v1 v2 ≫ galphaEq e1 e2
  _                                      → step x
go Decl x = case sop x of
  DVar (v1 :*: v2)                       → addRule v1 v2 ≫ return True
  DFun (f1 :*: f2) (x1 :*: x2) (s1 :*: s2) → addRule f1 f2
                                         ≫ scoped (addRule x1 x2 ≫ galphaEq s1 s2)
  _                                      → step x
go Exp x = case sop x of
  EVar (v1 :*: v2)                       → v1 ≈ v2
  ECall (f1 :*: f2) (e1 :*: e2)          → (∧) <$> f1 ≈ f2 <*> galphaEq e1 e2
  _                                      → step x
go _ x = step x
```

Figure 7. α-equivalence for a toy imperative language

with different constructors, *galphaEq* will already return *False* because *zipRep* will fail. When *zipRep* succeeds though, we get access to one constructor with paired fields inside. The *go* is then responsible for performing the necessary semantic actions for the *Var* and *Abs* constructors and applying a general eliminator for anything else. In the actual library, the *pattern synonyms* $\overline{Term_\lambda}$, $\overline{Var}$, and $\overline{Abs}$ are automatically generated as we will see in **??**.

One might be inclined to believe that the generic programming here is more cumbersome than a straightforward pattern matching definition over $Term_\lambda$. If we consider a more intricate language, however, manual pattern matching becomes almost intractable very fast.

Take the toy imperative language defined in **??**. α-equivalence for this language can be defined with just a couple of changes to the definition for $Term_\lambda$. For one thing, *alphaEq*, *step* and *galphaEq* remain the same. We just need to adapt the *go* function. Here writing α-equivalence by pattern matching is not straightforward anymore. Moreover, if we decide to change this language and add more statements or more expressions, the changes to the *go* function are minimal, none if we do not introduce any additional construct which declares or uses variables. As long as we do not touch the constructors that *go* patterns matches on, we can even use the very same function.

In this section we have shown several recurring examples from the generic programming community. `generics-mrsop` gives both expressive power and convenience. The last point we have to address is that we still have to write the *Family* instance for the types we want to use. For instance, the *Family* instance for example in **??** is not going to be fun. Deriving these automatically is possible, but non-trivial; we give a full account in **??**

6 Conclusion and Future Work

Generic programming is an ever changing field. The more the Haskell language evolves, the more interesting generic programming libraries we can create. Indeed, some of the language extensions we require in our work were not available at the time that some of the libraries in the related work were developed.

Future work involves expanding the universe of datatypes that our library can handle. Currently, every type involved in a recursive family must be a ground type (of kind $*$ in Haskell terms); our Template Haskell derivations acknowledges this fact by implementing some amount of reduction for types. This limits the functions we can implement generically, for example we cannot write a generic *fmap* function, since it operates on types of kind $* \to *$. `GHC.Generics` supports type constructors with exactly one argument via the `Generic1` type class. We intend to combine the approach in this paper with that of **?**], in which atoms have a wider choice of shapes.

The original sum-of-products approach does not handle all the ground types either, only regular ones [**?**]. We inherit this restriction, and cannot represent recursive families which involve existentials or GADTs. The problem in this case is representing the constraints that each constructor imposes on the type arguments.

Our `generics-mrsop` is a powerful library for generic programming that combines the advantages of previous approaches to generic programming. We have carefully blended the information about (mutually) recursive positions from `multirec`, with the sums-of-products codes introduced by `generics-sop`, while maintaining the advantages of both. The programmer is now able to use simple, combinator-based generic programming for a more expressive class of types than the sums-of-products approach allows. This is interesting, especially since mutually recursive types were hard to handle in a generic fashion previous to `generics-mrsop`.

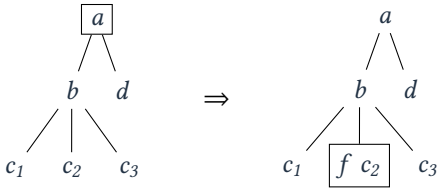
A The Generic Zipper

To add to our examples section we conduct a validation exercise involving a more complex application of generic programming. Zippers [**?**] are a well established technique for traversing a recursive data structure keeping track of the current *focus point*. Defining generic zippers is nothing new, this has been done by many authors [**? ? ?**] for many different classes of types in the past. To the best of the authors knowledge, this is the first definition in a direct *sums-of-products* style. We will not be explaining *are* zippers are in detail, instead, we will give a quick reminder and show how zippers fit within our framework.

Generally speaking, the zipper keeps track of a focus point in a data structure and allows for the user to conveniently move this focus point and to apply functions to whatever is under focus. This focus point is expressed by the means of a location type, *Loc*, with a couple of associated functions:

```
up, dowm, right :: Loc a → Maybe (Loc a)
update          :: (a → a) → Loc a → Loc a
```

Where a and *Loc* a are isomorphic, and can be converted by the means of *enter* and *leave* functions. For instance, the composition of *down*, *down*, *right* , *update* f will essentially move the focus two layers down from the root, then one element to the right and apply function f to the focused element, as shown below.



In our case, this location type consists of a distinguished element of the family *El fam ix* and a stack of contexts with a hole of type *ix*, where we can plug in the distinguished element. This stack of contexts may build a value whose type is a different member of the family; we recall its index as *iy*.

For the sake of conciseness we present the datatypes for a fixed interpretation of opaque types *ki* :: *kon* → ∗, a family *fam* :: [∗] and its associated codes *codes* :: [[[*Atom kon*]]]. In the actual implementation all those elements appear as additional parameters to *Loc* and *Ctxs*.

```
data Loc :: Nat → ∗ where
  Loc :: El fam iy → Ctxs ix iy → Loc ix
```

The second field of *Loc*, the stack of contexts, represents how deep into the recursive tree we have descended so far. Each time we unwrap another layer of recursion, we push some context onto the stack to be able to go back up. Note how the *Cons* constructor resembles some sort of composition operation.

```
data Ctxs :: Nat → Nat → ∗ where
  Nil  :: Ctxs ix ix
  Cons :: Ctx (Lkup codes iz) iy → Ctxs ix iz → Ctxs ix iy
```

Each element in this stack is an individual context, *Ctx c iy*. A context is defined by a choice of a constructor for the code c, paired a product of the correct type where one of the elements is a hole. This hole represents where the distinguished element in *Loc* was supposed to be.

```
data Ctx :: [[Atom kon]] → Nat → ∗ where
  Ctx :: Constr n c → NP□ (Lkup n c) iy → Ctx c iy

data NP□ :: [Atom kon] → Nat → ∗ where
  Here  :: NP (NA ki (El fam)) xs       → NP□ (I ix:xs) ix
  There :: NA ki (El fam) x → NP□ xs ix → NP□ (x   :xs) ix
```

The navigation functions are a direct translation of those defined for the `multirec` [**?**] library, that use the *first*, *fill*, and *next* primitives for working over *Ctx*s. The *fill* function can be taken over almost unchanged, whereas *first* and *next* require a simple trick: we have to wrap the *Nat* parameter of *NP*□ in an existential in order to manipulate it conveniently. The *ix* is packed up in an existential type since we do not really know beforehand which member of the mutually recursive family is seen first in an arbitrary product.

```
data ∃NP□ :: [Atom kon] → ∗ where
  Witness :: El fam ix → NP□ c ix → ∃NP□ c
```

Now we can define the $first_\exists$ and $next_\exists$, the counterparts of *first* and *next* from `multirec`. Intuitively, $first_\exists$ returns the *NP*□ with the first recursive position (if any) selected, $next_\exists$ tries to find the next recursive position in an *NP*□. These functions have the following types:

```
first∃ :: NP (NA ki (El fam)) xs → Maybe (∃NP□ xs)
next∃  :: ∃NP□ xs                → Maybe (∃NP□ xs)
```

To conclude we can now use flipped compositions for pure functions $(\ggg) :: (a \to b) \to (b \to c) \to a \to c$ and monadic functions $(>=>) :: (Monad\ m) \Rightarrow (a \to m\ b) \to (b \to m\ c) \to a \to m\ c$ to elegantly write some *location based* instruction to transform some value of the type $Term_\lambda$ defined in **??**. Here *enter* and *leave* witness the isomorphism between *El fam ix* and *Loc ix*.

```
tr :: Termλ → Maybe Termλ
tr = enter  >>> down
            >=> right
            >=> update (const $ Var "c")
            >>> leave
            >>> return

tr (App (Var "a") (Var "b"))
  ≡ Just (App (Var "a") (Var "c"))
```

We invite the reader to check the source code for a more detailed account of the generic zipper. In fact, we were able to provide the same zipper interface as the `multirec` library. Our implementation is shorter, however. This is because we do not need type classes to implement $first_\exists$ and $next_\exists$.

B Template Haskell

Having a convenient and robust way to get the *Family* instance for a given selection of datatypes is paramount for the usability of our library. In a real scenario, a mutually recursive family may consist of many datatypes with dozens of constructors. Sometimes these datatypes are written with parameters, or come from external libraries.

Our goal is to automate the generation of *Family* instances under all those circumstances using *Template Haskell* [**?**]. From the programmers' point of view, they only need to call *deriveFamily* with the topmost (that is, the first) type of the family. For example:

```
data Exp  var = ...
data Stmt var = ...
data Decl var = ...
data Prog var = ...

deriveFamily [t|Prog String|]
```

The *deriveFamily* takes care of unfolding the (type level) recursion until it reaches a fixpoint. In this case, the type synonym $FamProgString = '[Prog\ String, \ldots]$ will be generated, together with its *Family* instance. Optionally, one can also pass along a custom function to decide whether a type should be considered opaque. By default, it uses a selection of Haskell built-in types as opaque types.

B.1 Unfolding the Family

The process of deriving a whole mutually recursive family from a single member is conceptually divided into two disjoint processes. First we unfold all definitions and follow all the recursive paths until we reach a fixpoint. At that moment we know that we have discovered all the types in the family. Second, we translate the definition of those types to the format our library expects. During the unfolding process we keep a key-value map in a *State* monad, keeping track of the types we have seen, the types we have seen *and* processed and the indices of those within the family.

Let us illustrate this process in a bit more detail using our running example of a mutually recursive family and consider what happens within *Template Haskell* when it starts unfolding the *deriveFamily* clause.

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

deriveFamily [t|Rose Int|]
```

The first thing that happens is registering that we seen the type *Rose Int*. Since it is the first type to be discovered, it is assigned index zero within the family. Next we need to reify the definition of *Rose*. At this point, we query *Template Haskell* for the definition, and we obtain **data** $Rose\ x = Fork\ x\ [Rose\ x]$. Since *Rose* has kind $* \to *$, it cannot be directly translated – our library only supports ground types, which are those with kind $*$. But we do not need a generic definition for *Rose*, we just need the specific case where $x = Int$. Essentially, we just apply the reified definition of *Rose* to *Int* and β-reduce it, giving us $Fork\ Int\ [Rose\ Int]$.

The next processing step is looking into the types of the fields of the (single) constructor *Fork*. First we see *Int* and decide it is an opaque type, say *KInt*. Second, we see $[Rose\ Int]$ and notice it is the first time we see this type. Hence, we register it with a fresh index, $S\ Z$ in this case. The final result for *Rose Int* is $'[\,'[K\ KInt, I\ (S\ Z)]]$.

We now go into $[Rose\ Int]$ for processing. Once again we need to perform some amount of β-reduction at the type level before inspecting its fields. The rest of the process is the same that for *Rose Int*. However, when we encounter the field of type *Rose Int* this is already registered, so we just need to use the index Z in that position.

The final step is generating the actual Haskell code from the data obtained in the previous process. This is a very verbose and mechanical process, whose details we omit. In short, we generate the necessary type synonyms, pattern synonyms, the *Family* instance, and metadata information. The generated type synonyms are named after the topmost type of the family, passed to *deriveFamily*:

```
type FamRoseInt
       = '[Rose Int              , [Rose Int]]
type CodesRoseInt
       = '['['[K KInt, I (S Z)]], '['[], '[I Z, I (S Z)]]]
```

Pattern synonyms are useful for convenient pattern matching and injecting into the *View* datatype. We produce two different kinds of pattern synonyms. First, synonyms for generic representations, one per constructor. Second, synonyms which associate each type in the recursive family with their position in the list of codes.

```
pattern Fork x xs = Tag SZ       (NA_K x × NA_I xs × NP0)
pattern []        = Tag SZ       NP0
pattern x : xs    = Tag (SS SZ) (NA_I x × NA_I xs × NP0)

pattern RoseInt     = SZ
pattern ListRoseInt = SS SZ
```

The actual *Family* instance is exactly as the one shown in **??**

```
instance Family Singl FamRoseInt CodesRoseInt where ...
```

C Metadata

The representations described in this paper is enough to write generic equalities and zippers. But there is one missing ingredient to derive generic pretty-printing or conversion to JSON, for instance. We need to maintain the *metadata* information of our datatypes. This metadata includes the datatype name, the module where it was defined, and the name of the constructors. Without this information you cannot write a function which outputs the string

```
Fork 1 [Fork 2 [], Fork 3 []]
```

for a call to *genericShow* (*Fork* 1 [*Fork* 2 [ ], *Fork* 3 [ ]]). The reason is that the code of *Rose Int* does not contain the information that the constructor of *Rose* is called "*Fork*".

Like in `generics-sop` [**?**], having the code for a family of datatypes available allows for a completely separate treatment of metadata. This is yet another advantage of the sum-of-products approach when compared to the more traditional pattern functors. In fact, our handling of metadata is heavily inspired from `generics-sop`, so much so that we will start by explaining a simplified version of their handling of metadata, and then outline the differences to our approach.

The general idea is to store the meta information following the structure of the datatype itself. So, instead of data, we keep track of the names of the different parts and other meta information that can be useful. It is advantageous to keep metadata separate from the generic representation as it would only clutter the definition of generic functionality. This information is tied to a datatype by means of an additional type class *HasDatatypeInfo*. Generic functions may now query the metadata by means of functions like *datatypeName*, which reflect the type information into the term level. The definitions are given in **??**.

Our library uses the same approach to handle metadata. In fact, the code remains almost unchanged, except for adapting it to the larger universe of datatypes we can now handle. Unlike `generic-sop`, our list of lists representing the sum-of-products structure does not contain types of kind $*$, but *Atom*s. All the types representing metadata at the type level must be updated to reflect this new scenario:

```
data DatatypeInfo    :: [[Atom kon]] → ∗ where...
data ConstructorInfo :: [Atom kon]   → ∗ where...
data FieldInfo       ::  Atom kon    → ∗ where...
```

As we have discussed above, our library is able to generate codes not only for single types of kind $*$, like *Int* or *Bool*, but also for types which are the result of type level applications, such as *Rose Int* and [*Rose Int*]. The shape of the metadata information in *DatatypeInfo*, a module name plus a datatype name, is not enough to handle these cases. We replace the uses of *ModuleName* and *DatatypeName* in *DatatypeInfo* by a richer promoted type *TypeName*, which can describe applications, as required.

```
data TypeName = ConT ModuleName DatatypeName
              | TypeName :@: TypeName

data DatatypeInfo :: [[Atom kon]] → ∗ where
  ADT  :: TypeName → NP ConstructorInfo cs
       → DatatypeInfo cs
  New  :: TypeName →      ConstructorInfo '[c]
       → DatatypeInfo '[ '[c]]
```

The most important difference to `generics-sop`, perhaps, is that the metadata is not defined for a single type, but for a type *within* a family. This is reflected in the new signature of *datatypeInfo*, which receives proxies for both the family and the type. The type equalities in that signature reflect the fact that the given type *ty* is included with index *ix* within the family *fam*. This step is needed to look up the code for the type in the right position of *codes*.

```
class (Family κ fam codes)
      ⇒ HasDatatypeInfo κ fam codes ix
      | fam → κ codes where
  datatypeInfo :: (ix ∼ Idx ty fam, Lkup ix fam ∼ ty)
               ⇒ Proxy fam → Proxy ty
               → DatatypeInfo (Lkup ix codes)
```

The Template Haskell will then generate something similar to the instance below for the first type in the family, *Rose Int*:

```
instance HasDatatypeInfo Singl FamRose CodesRose Z where
  datatypeInfo _ _
    = ADT (ConT "E" "Rose" :@: ConT "Prelude" "Int")
    $ (Constructor "Fork") × NP0
```

Once all the metadata is in place, we can use it in the same fashion as `generics-sop`. We refer the interested reader to **?**] for examples.

```
data DatatypeInfo :: [[∗]] → ∗ where
  ADT :: ModuleName → DatatypeName → NP ConstructorInfo cs   → DatatypeInfo cs
  New :: ModuleName → DatatypeName →      ConstructorInfo '[c] → DatatypeInfo '['[c]]
data ConstructorInfo :: [∗] → ∗ where
  Constructor :: ConstructorName                          → ConstructorInfo xs
  Infix       :: ConstructorName → Associativity → Fixity → ConstructorInfo '[x, y]
  Record      :: ConstructorName → NP FieldInfo xs        → ConstructorInfo xs
data FieldInfo :: ∗ → ∗ where
  FieldInfo :: FieldName → FieldInfo a
class HasDatatypeInfo a where
  datatypeInfo :: proxy a → DatatypeInfo (Code a)
```

Figure 8. Definitions related to metadata from `generics-sop`