A Generic Journey

Design decisions about datatype-generic programming in Haskell

Andres Löh, Well-Typed LLP

2020-06-13

In this text, I will try to explain the implementation of datatype-generic programming in Haskell, by starting from scratch and going towards `GHC.Generics`, discussing the important features and design decisions along the way.

In the second part, we will continue and modify `GHC.Generics` in several steps, trying to remove some potential disadvantages. We will see how every change affects our generic programs – including a number of setbacks – but ultimately we will reach a scenario that is close to `generics-sop`. It is my hope that by taking this journey, the relation between different approaches to generic programming and their trade-offs becomes more apparent, and that through increased understanding, it also becomes easier to use all of these libraries.

1 Towards `GHC.Generics`

1.1 Introduction

The goal of *(datatype-)generic programming* is to define functions that work on many different types by exploiting the common structure of datatypes.

What does this mean? Consider an example datatype such as the following type of *binary trees*:

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

Every Haskell datatype is introduced by the `data` construct, comprises a number of different constructors, each of which have a number of fields. Yet, Haskell treats

each datatype as completely new and different from everything else, because it employs a *nominal* type system. Another datatype such as

```
data Baum a = Blatt a | Knoten (Baum a) (Baum a)
```

– even though it has exactly the same structure as `Tree` – is nevertheless going to be treated as incompatible. But if we're planning to exploit the structure of datatypes, we have to somehow weaken this nominal view temporarily.

The core idea of generic programming is to introduce a *representation* of datatypes that is isomorphic to the original type, but focuses on the structure rather than the names, ideally by reducing the structure to a limited number of building blocks.

To illustrate this idea, we introduce the type class

```
class Generic a where
  type Rep a :: Type
  from :: a -> Rep a
  to :: Rep a -> a
```

which associates with a type `a` its representation type `Rep a` and contains functions `from` and `to` that witness the isomorphism between `Rep a` and `a`, so in particular `to . from` should be the identity.

There is a lot of freedom in how exactly we build the representations, and we will discuss a number of variants in the following.

As a start, let us pursue the following plan:

- We replace the *choice* between constructors with the use of `Either`. The Haskell type `Either a b` captures nicely the choice between two types `a` and `b`, so it can be used to model a type with two constructors. If we have a type with more than two constructors, we can use `Either` multiple times, in a nested way.
- We replace the *combination* of fields (i.e., constructor arguments) with the use of `(,)`. The Haskell type `(a, b)` captures the combination of two types `a` and `b`, so it can be used to model a constructor with two fields. If a constructor has more than two fields, we can use `(,)` repeatedly, in a nested way.

If we apply this plan to `Tree`, we might obtain

```
instance Generic (Tree a) where
  type Rep (Tree a) = Either a (Tree a, Tree a)
```

I'm saying "we might obtain", because there are actually several option for what we could do with the recursive occurrences of `Tree a`. For example:

- We can recursively transform them – but this would lead to `Rep` being an infinite type synonym, or we would have to turn it into a **`newtype`**.
- We can introduce an explicit type constructor such as `Fix` to capture the recursion.
- We can just leave them alone (as I did above).

Whether to capture recursion using fixed points is a important design decision, with dramatic implications. Capturing recursion makes a lot of things harder, but in turn enables us to write many interesting functions such as, for example, recursion patterns.

In the following, we take the simple route and "ignore" recursion. In effect, we structurally represent just one layer of the datatype, keeping all the arguments of all the constructors in their original shape, whether they are recursive occurrences of the type under consideration or not.

This happens to be the approach taken by both `GHC.Generics` and `generics-sop`. If you are interested in approaches explicitly capturing recursion, you might want to look at libraries such as `compdata`, `regular` or `multirec`.

It remains to define the `from` and `to` conversion functions. This is entirely straightforward:

```
from :: Tree a -> Rep (Tree a)
from (Leaf x)   = Left x
from (Node l r) = Right (l, r)

to :: Rep (Tree a) -> Tree a
to (Left x)       = Leaf x
to (Right (l, r)) = Node l r
```

In general, instances of the different variants of the `Generic` class we are going to consider are so simple to write that this can easily be automated. In some cases (such as for `GHC.Generics`), this automation might be built into GHC directly. In other cases, Template Haskell can be used to derive these instances.

They key point, though, is that `Generic` is now the *only* class that requires special treatment in this way. Other classes we might want to derive can instead be user-defined as *generic programs*.

1.2 A first generic function: generic equality

As a standard example, let us consider generic equality, such as given by the `Eq` class. The functionality to derive this is already in GHC for many years, of course,

but if our generic programming library is going to be good for anything, we should be able to easily reproduce this.

The goal is to define equality by moving to the representation type and operating on that, like in the following:

```
eq :: (Generic a, …) => a -> a -> Bool
eq a1 a2 = geq (from a1) (from a2)
```

Here, `geq` should somehow be able to deal with all the shapes of representation types that can plausibly occur (in particular, with types built using `Either` and `(,)`). In Haskell, if we want to define a function that can operate on a number of different types, we usually use a type class, so let us try this:

```
eq :: (Generic a, GEq (Rep a)) => a -> a -> Bool
eq a1 a2 = geq (from a1) (from a2)

class GEq a where
  geq :: a -> a -> Bool
```

So far, so good. We now need to define instances of `GEq`, at the very least for `Either` and `(,)`. This is not particularly difficult; we just have to recall how to define equality for these two types:

```
instance (GEq a, GEq b) => GEq (Either a b) where
  geq (Left a1)  (Left a2)  = geq a1 a2
  geq (Right b1) (Right b2) = geq b1 b2
  geq _          _          = False
instance (GEq a, GEq b) => GEq (a, b) where
  geq (a1, b1) (a2, b2) = geq a1 a2 && geq b1 b2
```

Are these instances enough? Let us see what happens if, at this point, we try to define equality on trees via `eq`:

```
instance Eq a => Eq (Tree a) where    -- fails
  (==) = eq
```

The error message reports that there is no instance for `GEq a`. If we look closely, it is understandable that this happens, because `a` occurs in `Rep (Tree a)`. But where is this instance on `GEq a` going to come from? Well, we would like to just reuse an existing `Eq` instance on `a`, as we have indicated in the instance declaration for `Tree` above.

But how can we instruct GHC to use the `Eq a` instance at the place where `a` appears in `Rep (Tree a)`? We might be briefly tempted to say something like

```
instance Eq a => GEq a where    -- bad
  geq = (==)
```

but this is a bad instance, as it would overlap with everything else.

1.3 Wrapping constructor fields

What we can (and should) do instead is to *change the representation* and *wrap* the occurrence of the variable `a` in a dedicated type, and then tie the reuse of an existing `Eq` instance to the occurrence of this wrapper.

We therefore introduce

```
newtype Wrap a = Wrap a
```

and then adapt the `Generic` instance for `Tree2`:

```
instance Generic (Tree a) where
  type Rep (Tree a) = Either (Wrap a) (Tree a, Tree a)
  from :: Tree a -> Rep (Tree a)
  from (Leaf x)   = Left (Wrap x)
  from (Node l r) = Right (l, r)
  to :: Rep (Tree a) -> Tree a
  to (Left (Wrap x)) = Leaf x
  to (Right (l, r))  = Node l r
```

We have to adapt `from` and `to` as well, to insert the constructor applications of `Wrap` in the right places.

Now we can extend our `GEq` class with an instance for `Wrap`, saying that in places where `Wrap` occurs, we want to fall back on `Eq`:

```
instance Eq a => GEq (Wrap a) where
  geq (Wrap a1) (Wrap a2) = a1 == a2
```

Now, if we try instantiating `eq` again, we get a different error:

```
instance Eq a => Eq (Tree a) where    -- fails
  (==) = eq
```

The error message now complains that an instance for `GEq (Tree a)` is missing. But this is the same situation as just before! We are defining an instance `Eq (Tree a)` right now, and we want to reuse this equality in the recursive positions.

So we will have to apply `Wrap` uniformly to all constructor fields appearing in the representation type. We therefore revise our `Generic` instance once more:

```
instance Generic (Tree a) where
  type Rep (Tree a) = Either (Wrap a) (Wrap (Tree a), Wrap (Tree a))

  from :: Tree a -> Rep (Tree a)
  from (Leaf x)   = Left (Wrap x)
  from (Node l r) = Right (Wrap l, Wrap r)

  to :: Rep (Tree a) -> Tree a
  to (Left (Wrap x))          = Leaf x
  to (Right (Wrap l, Wrap r)) = Node l r
```

This time, defining equality for trees works:

```
instance Eq a => Eq (Tree a) where
  (==) = eq
```

And it works as expected.

We thus now have basic generic programming machinery in place: we represent the choice of constructors by (nested) `Either`s, and the combination of constructor fields by (nested) `(,)`s, and we wrap every constructor field in `Wrap`. We do not need anything in particular to handle recursion, because we can reuse existing type class instances in the generic `Wrap` instances.

Exercise 1. Define a `Generic` instance for the following type of lambda terms:

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

Remember to use nested `Either` to represent the choice between three constructors. Define an instance `Eq Term` using the generic equality function and observe that it works.

Exercise 2. Define generic comparison, i.e.,

```
cmp :: (Generic a, GCmp (Rep a)) => a -> a -> Ordering
```

and

```
class GCmp a where …
```

such that `cmp` can be used as a generic definition for `compare` when instantiating the `Ord` class.

1.4 Generic enumeration and constructors without fields

Is our development so far sufficient to also define other generic functions? Let us consider another – quite different – example, namely a generic operation that enumerates all values of a datatype as a list – similary, yet not quite the same as what the `Enum` class is offering.

Following the pattern established for equality, we would like to be able to define

```
enum :: (Generic a, GEnum (Rep a)) => [a]
enum = to <$> genum
```

with

```
class GEnum a where
  genum :: [a]
```

Note how this time, we are *producing* values of a particular type, whereas for equality, we have been *consuming* values. This is why `eq` only needed `from`, and for `enum`, we need `to`.

While it is possible and interesting to define generic enumeration functions for complicated, recursive, types such as `Tree`, we will here limit ourselves to the goal of defining `enum` in such a way that it works on *enumeration types*, i.e., types that consist of a number of constructors without any constructor arguments. A similar restriction is actually in place in GHC when trying to derive the `Enum` class for a datatype.

To this end, let us introduce a new example datatype that falls into this class:

```
data Colour = Red | Green | Blue
  deriving Show
```

Once we try to define a `Generic` instance for `Colour`, we run into a small problem: we do not have a way to represent a constructor without any arguments. But this is easy to fix: we use `()`.

```
instance Generic Colour where
  type Rep Colour = Either () (Either () ())
  from :: Colour -> Rep Colour
  from Red   = Left ()
  from Green = Right (Left ())
  from Blue  = Right (Right ())
  to :: Rep Colour -> Colour
```

```
to (Left ())          = Red
to (Right (Left ()))  = Green
to (Right (Right ())) = Blue
```

There are a number of questions we should briefly discuss at this point:

- Could we have used `Wrap ()` instead of `()`? No, we would use `Wrap ()` when we have a constructor that has an argument of type `()`. Both situations should have a different representation.
- In what way do we nest the `Either` (and `(,)`) applications if we have multiple constructors (and fields)? For now, we just pick one consistently, and argue that it "should not matter", i.e., functions should not rely on a particular nesting. We will get back to this point later.

Now that we have effectively added `()` to the types that can occur in representations, we should adapt `GEq` to have a case for `()` – otherwise, we will not be able to use generic equality on types containing constructors without arguments, such as `Colour`.

```
instance GEq () where
  geq () () = True
```

One can argue here whether `geq` on `()` should be strict in its arguments or just always return `True`. And indeed it makes some difference. But e.g. GHC's equality for `()` is strict in both arguments, so we copy that behaviour.

With this change, we can define

```
instance Eq Colour where
  (==) = eq
```

and it works as expected.

However, we actually wanted to define generic enumeration via the `GEnum` class. We start with the instance for `()`:

```
instance GEnum () where
  genum = [()]
```

The `()` type can be seen as an enumeration type with a single value, so we return that value as a one-element list.

Next, let us consider the `Either` instance:

```
instance (GEnum a, GEnum b) => GEnum (Either a b) where
  genum = (Left <$> genum) ++ (Right <$> genum)
```

If we have a choice between multiple constructors of the type we want to generate, we generate all values for both options, and prefix the resulting values with the correct constructors from the `Either` type.

Do we need additional instances for `GEnum`? Well, at least not for `Colour`, because only `Either` and `()` occur in its representation type.

```
GHCi> enum @Colour
[Red, Green, Blue]
```

More generally, recall that we said we only want our `enum` function to work for *enumeration types*. If enumeration types only have constructors without arguments, then `(,)` and `Wrap` will indeed never occur in their representations, so our definition is fine.

It is a bit sad though that this knowledge is ultimately *implicit*: we have to understand our encoding to see which types we will be able to apply `enum` to. For now, let us leave it at that.

Exercise 3. Define a generic function

```
baseCases :: (Generic a, GBaseCases (Rep a)) => [a]
```

that for enumeration types coincides with `enum`, however works for other types as well, always returning just those constructors that have no arguments.

Exercise 4. Define a generic function

```
memp :: (Generic a, GMemp (Rep a)) => a
```

that generically implements the `mempty` operation for monoids on *product types*. This function should statically fail on types that have more than one constructor, and it may require that all fields of that constructor are an instance of `Monoid`.

Exercise 5. Define a generic function

```
total :: (Generic a, GTotal (Rep a)) => a -> Int
```

that should require that all fields of all constructors are of type `Int`, and then compute the total of the `Int` values appearing in the constructor fields of the given argument.

For example, for

```
data Foo =
     Bar Int Int Int
   | Baz Int Int
```

with a suitable `Generic` instance, the call `total (Bar 3 4 5)` should return `12`.

1.5 Accessing constructor names

We originally argued that one key ingredient of generic programming is to get away from the nominal nature of Haskell's type system and focus on just the structure. There is no hint in our current representations what e.g. the names of the constructors were. On the other hand, there are functions that look like they should be definable generically, but definitely need access to information such as constructor names. A prime example are the methods from the `Show` and `Read` type classes in Haskell: clearly, the textual format produced and consumed by these operations is very systematically derived from the datatype declarations, and metadata such as names of constructors or names of record selectors plays a prominent role in this representation.

Perhaps we can embed such metadata in the representation without losing the property that the representation types should be built from just a few ingredients.

To limit the scope of our exploration, let us focus just on constructor names for now (and disregard the name of the datatype, names of record selectors, and other potential metadata such as module names or information about the priority of infix constructors). And as an example function we aim to define, let us not consider something quite as complex as `show` or `read`, but rather a function

```
conName :: (Generic a, …) => a -> String
```

that merely produces the name of the outermost constructor of the provided input value.

In order to achieve our goal to embed constructor information uniformly, we again add a new type for use in our representations:

```
newtype Con (n :: Symbol) a = Con a
```

Its use is best demonstrated by looking at how the `Generic` instance for trees has to be adapted:

```
instance Generic (Tree a) where
  type Rep (Tree a) =
    Either (Con "Leaf" (Wrap a))
           (Con "Node" (Wrap (Tree a), Wrap (Tree a)))
```

Both constructors are no embedded in an application of `Con`. The constructor name is added a a type-level string, but does not feature on the term level.

Somewhat unfortunately, the presence of `Con` still produces traces on the term level, and we have to adapt `from` and `to` accordingly:

```
  from :: Tree a -> Rep (Tree a)
  from (Leaf x)   = Left (Con (Wrap x))
  from (Node l r) = Right (Con (Wrap l, Wrap r))

  to :: Rep (Tree a) -> Tree a
  to (Left (Con (Wrap x)))          = Leaf x
  to (Right (Con (Wrap l, Wrap r))) = Node l r
```

The `Generic` instance for `Colour` does of course change similarly:

```
instance Generic Colour where
  type Rep Colour =
    Either     (Con "Red"   ())
      (Either (Con "Green" ())
              (Con "Blue"  ()))

  from :: Colour -> Rep Colour
  from Red   = Left (Con ())
  from Green = Right (Left (Con ()))
  from Blue  = Right (Right (Con ()))

  to :: Rep Colour -> Colour
  to (Left (Con ()))          = Red
  to (Right (Left (Con ())))  = Green
  to (Right (Right (Con ()))) = Blue
```

And of course, we have to extend our existing generic functions with new (boring) cases on how to handle `Con`:

```
instance GEq a => GEq (Con n a) where
  geq (Con a1) (Con a2) = geq a1 a2

instance GEnum a => GEnum (Con n a) where
  genum = Con <$> genum
```

For both functions, we are not actually interested in the constructor names, and just referring to the case of the underlying type `a`. It is perhaps noteworthy that the `GEnum` instance shows that it is good that we have not embedded the constructor name on the term level. If we had, we would have to *produce* a constructor name in that instance, but that does not make sense. There is only one valid choice at any given point, and that choice is statically known.

We can now return to `conName`:

```
conName :: (Generic a, GConName (Rep a)) => a -> String
conName a = gconName (from a)
```

```
class GConName a where
  gconName :: a -> String
```

Up to here, we follow the by now established pattern. The interesting case for this function is certainly the one for `Con`:

```
instance KnownSymbol n => GConName (Con n a) where
  gconName _ = symbolVal (Proxy @n)
```

At this point, we should already know the current constructor is the correct one, and all we need to do is to produce a term-level `String` corresponding to the type-level symbol `n`, which we can do using `symbolVal`.

We also need an instance for `Either`:

```
instance (GConName a, GConName b) => GConName (Either a b) where
  gconName (Left a)  = gconName a
  gconName (Right b) = gconName b
```

Here, we cannot ignore the value, but have to follow the path through the `Either`-structure as indicated by the constructors.

It turns out that these two cases are sufficient, even though, unlike `genum`, the function `conName` works for all types in the `Generic` class, so e.g. also for trees:

```
GHCi> conName (Node (Leaf 'x') (Leaf 'y'))
"Node"
GHCi> conName Green
"Green"
```

How can that be, given that we have defined fewer cases? The key difference is that for `gconName`, there are no preconditions on the instance for `Con`, so we never look deeper than that, whereas for `genum`, we do continue to look underneath the `Con`, and only stop at `()`, failing in situations where there is a `(,)` or `Wrap` occurrence.

This is unfortunately all very subtle! We are exploiting information on how we choose to represent the types, such as that there is an `Either`-structure, under which there is guaranteed to be a `Con` occurrence before there can be anything else. This knowledge has to be carefully documented, and makes being certain about the correctness of a generic function somewhat difficult.

Before we look at this topic again and set out to improve some of these flaws, let us consider one more example of a generic function.

1.6 Constructor index

Now we want to define a generic function that is a variant of `conName`. Rather than producing the *name* of the outermost constructor, it should produce the *index* of that constructor, where we define the index to be a `0`-based numbering of the constructors in order they appear in the datatype declaration. So, e.g., in the case of `Colour`, we want `Red` to have index `0`, `Green` to have index `1`, and `Blue` to have index `2`.

The good news is that we will not have to change our setup once again in order to accommodate the constructor index computation. It is nevertheless not entirely trivial. Once again following our general pattern, we start with

```
conIx :: (Generic a, GConIx (Rep a)) => a -> Int
conIx a = gconIx (from a)

class GConIx a where
  gconIx :: a -> Int
```

As this function is very similar to `gconName`, we might again expect to need cases for `Con` and `Either`. The one for `Con` could look as follows:

```
instance GConIx (Con n a) where
  gconIx _ = 0
```

From the perspective of a single `Con n a` constructor, this is the only constructor. And we have no further information available here, so `0` seems the only plausible result we can produce.

Next, let us look at `Either`:

```
instance (GConIx a, GConIx b) => GConIx (Either a b) where
  gconIx (Left a)  = gconIx a
  gconIx (Right b) = gconIx b
```

Here, we are in real trouble. The given definition typechecks, but is rather useless. We will now *always* get `0` as the result. We need to increment the index at some point.

There are multiple options to achieve this. One solution is to add a second method `gconCount` to the `GConIx` class, with the idea that it returns the total number of `Con` ocurrences in a representation:

```
class GConIx a where
  gconIx :: a -> Int
  gconCount :: Int
```

Note that the type of `gconCount` does not mention `a`, yet still depends on it: we need no value of type `a` in order to decide how many constructors the type `a` has. GHC will not be able to infer at what type we are calling `gconCount`, forcing us to enable the `AllowAmbiguousTypes` extension and to use `TypeApplications` to explicitly instantiate the type argument.

Generally, I am a huge fan of using `Proxy` arguments for such functions even in the presence of `TypeApplications`, because the `Proxy` arguments *document* that an explicit instantiation of the type argument is required. However, as I consider classes such as `GConIx` to be internal to the definition of the `conIx` function, and `gconCount` is thus not a public function, and we can be expected to recall our own calling conventions, I opt for conciseness here.

Given the now added `gconCount`, we have to revisit the instance for `Con`:

```
instance GConIx (Con n a) where
  gconIx _ = 0
  gconCount = 1
```

So far, so good. Let us see if we can improve the `Either` instance now:

```
instance (GConIx a, GConIx b) => GConIx (Either a b) where
  gconIx (Left a)  = gconIx a
  gconIx (Right b) = gconCount @a + gconIx b
  gconCount = gconCount @a + gconCount @b
```

If we descend to the left, then we can just recursively call `gconIx`. However, if we descend to the right, we can now offset the index returned by the recursive call by the number of constructors we are skipping over in the component of type `a`.

Counting the constructors in an `Either` means just adding the constructors occurring to the left and to the right.

Note that with this definition, we make no assumptions as to how the `Either` occurrences are nested in the case of many constructors.

Let us test the function:

```
GHCi> conIx <$> enum @Colour
[0, 1, 2]
```

Exercise 6. Define a simple parser for enumeration types, i.e., a generic function that takes a `String` representation of the constructor name and produces the appropriate value.

Using a parser library is not required here – you can compare the full given string for equality with the candidate values.

Exercise 7. Create a way for the representations to provide access to the labels of record field selectors. Define an example record type and an appropriate `Generic` instance.

Exercise 8. Based on the work of the previous exercise, define a simplified `show` function for product types that may require all components to be in the `Show` class and prefixes each component with its field selector name, and ideally separates the fields with commas (but without a trailing or initial comma).

1.7 Checkpoint: `GHC.Generics`

Our current setup is quite close to that used by `GHC.Generics`. The main difference is that `GHC.Generics` aims to not just represent datatypes of kind `Type`, but also e.g. datatypes of kind `Type -> Type`.

If we allow that, we can for example try to generically define `fmap` from the `Functor` class. This is made difficult with our representation, because we do not get first class access to the argument of a type constructor in our current representation.

For `Type -> Type` datatypes, we have to use a different `Generic` class, because ours is limited to kind `Type`. But a comparatively small change will do:

```
class Generic1 (f :: k -> Type) where
  type Rep1 f :: k -> Type

  from1 :: f x -> Rep1 f x
  to1 :: Rep1 f x -> f x
```

At no extra cost we can even generalise to kind `k -> Type`. If our representations now have to be of kind `k -> Type` as well, so we can e.g. no longer use `Either` because it has the wrong kind. Instead, a *lifted* version of `Either` is used, which is called `(:+:)`:

```
data (f :+: g) x = L1 (f x) | R1 (g x)
```

All the other type constructors we have discussed for representations, `(,)`, `()`, `Wrap`, `Con` are replaced by lifted variants as well.

Also, `Con` is replaced by a more general construct `M1` that is used to express different kinds of metadata (datatype names, constructor names, selector names, and other info about them) throughout the representation.

Next to `Generic1`, `GHC.Generics` still has a `Generic` class for types of kind `Type`. Given the need for these lifted types for `Generic1`, `GHC.Generics` could have made the decision to use different representation types for its `Rep` and `Rep1` representations.

However, the design decision has been made to use the lifted type constructors also for the `Rep`, leading to an unused type argument in the conversion functions `from` and `to`. To illustrate, this is how the `Generic` class in `GHC.Generics` looks like:

```
class Generic a where
  type Rep a :: Type -> Type
  from :: a -> Rep a x
  to :: Rep a x -> a
```

But this is where the differences end. Generic functions are defined via type classes with instances for the type constructors appearing in the representations, and our generic functions that we have written so far can straight-forwardly be converted to work with `GHC.Generics`.

Now that we have reached this point, let us evaluate a bit. It is certainly great that we can represent datatypes using just a few constructs, and that we can elegantly deal with recursive functions on recursive types without having to do any real work, by just tying the knot via type classes, as we have seen in the generic equality case. It is also convenient that we can just define the instances we need, and add extra constraints to these instances just as needed (see, for example, the constraints placed on the `Wrap` instance in `GEq`, or `KnownSymbol` for `Con`, or the fact that we decided *not* to require `GConName a` in the instance for `GConName (Con n a)`).

Considering performance, we certainly are likely to incur *some* overhead, because we a converting between our original types and their structural representation, and then operate on that. However, some aspects also work in our favour: since `from` and `to` are shallow and convert only the top layer, they are non-recursive and can be inlined. The generic functions themselves are defined via classes, which get resolved at compile time. Together with aggressive inlining, there is hope that most of the overhead is optimised away, and indeed GHC is rather good at optimising generic functions by means of compile-time inlining and partial evaluation.

On the other hand, there are certainly shortcomings, some of which we have already identified: It is only implicitly defined which types occur at all or are allowed to occur in representations. A lot of invariants are supposed to hold on representations, but they are all implicit. For example, all constructor arguments are supposed to be wrapped in `Wrap`, and directly underneath the `Either`-structure representing the choice between constructors, an occurrence of `Con` is supposed to occur. No `(,)`s are supposed to appear anywhere except underneath the `Con` occurrences. And so on. Generic functions may exploit these invariants, even though they are not explicitly stated, and that makes it difficult to read or understand how generic functions work, and it also makes them fragile, because someone could change the representation in subtle ways and accidentally break a lot of functions,

or function authors might start relying on assumptions that are *not* actually intended invariants, and thereby cause their functions to later break in subtle ways.

Another disadvantage is that the general pattern for each new generic function is to define a new class and cases for all (or most) of the types occurring in representations. That feels somewhat low-level, such as if we would define every Haskell function by explicit pattern matching. In practice, we often resort to higher-order functions and compositions instead, and while this style may not be impossible to use for generic function, it seems to at least not be encouraged.

In the following, we will gradually change our design in order to address some of these shortcomings.

Exercise 9. Complete the sketched `Generic1` class from above so that you can represent `Tree`, and try to define a generic implementation of `fmap` based on that class.

Exercise 10. Define the `Generic` instance from `GHC.Generics` manually for at least one of the datatypes we discussed, and rewrite at least one of the generic functions we discussed to `GHC.Generics`.

2 From `GHC.Generics` to `generics-sop`

2.1 A kind for descriptions

Up until now, our representations have just been of kind `Type`, and thereby conceptually *open*. For a class such as `GEq`, there's no clearly defined set of acceptable instances. We could easily make `Int` and instance of `GEq`, for example, even though `Int` is never intended to appear un-wrapped in any representations.

We can do better than that by introducing a closed data kind for representations:

```
data Representation =
       Sum Representation Representation       -- corresponds to Either
     | Constructor Symbol Representation       -- corresponds to Con
     | Product Representation Representation   -- corresponds to (,)
     | Unit                                    -- corresponds to ()
     | Atom Type                               -- corresponds to Wrap
```

This type is only supposed to be used as a kind, in promoted form.

We still need to map these codes to types (of kind `Type`) for our conversions `from` and `to`. For this, there are several options. We can use a type family, or a data family, or a GADT, and possibly even other techniques. Probably the simplest option to start with is a type family:

```
type family Interpret (r :: Representation) :: Type where
  Interpret (Sum r s)         = Either (Interpret r) (Interpret s)
  Interpret (Constructor n r) = Interpret r
  Interpret (Product r s)     = (Interpret r, Interpret s)
  Interpret Unit              = ()
  Interpret (Atom a)          = a
```

Perhaps surprisingly, the types `Con` and `Wrap` do not appear in the image of the `Interpret` type family. We do not really need them, as they do not change the run-time representation of the value being represented, and the structural information is now contained in the code.

We now define

```
class Generic (a :: Type) where
  type Code (a :: Type) :: Representation

  from :: a -> Rep a
  to :: Rep a -> a

type Rep a = Interpret (Code a)
```

where `Rep` is now a type synonym defined in terms of `Interpret` and `Code`, and `Code` is now the associated type in the `Generic` class, but maps to something of kind `Representation`.

Let us look at how the instances of our example type `Tree` is affected:

```
instance Generic (Tree a) where
  type Code (Tree a) =
    Sum (Constructor "Leaf" (Atom a))
        (Constructor "Node" (Product (Atom (Tree a)) (Atom (Tree a))))

  from :: Tree a -> Rep (Tree a)
  from (Leaf x)   = Left x
  from (Node l r) = Right (l, r)

  to :: Rep (Tree a) -> Tree a
  to (Left x)       =       Leaf x
  to (Right (l, r)) =       Node l r
```

The conversion functions have actually become simpler, because we no longer have to deal with the `Con` and `Wrap` newtypes.

The instance for `Colour` changes in a similar way:

```
instance Generic Colour where
  type Code Colour =
```

```
    Sum     (Constructor "Red"   Unit)
      (Sum (Constructor "Green" Unit)
           (Constructor "Blue"  Unit))

from :: Colour -> Rep Colour
from Red   = Left ()
from Green = Right (Left ())
from Blue  = Right (Right ())

to :: Rep Colour -> Colour
to (Left ())          = Red
to (Right (Left ()))  = Green
to (Right (Right ())) = Blue
```

Next, let us see how generic functions are affected. Let us start with generic equality again:

```
eq :: forall a. (Generic a, GEq (Code a)) => a -> a -> Bool
eq a1 a2 = geq @(Code a) (from a1) (from a2)

class GEq (r :: Representation) where
  geq :: Interpret r -> Interpret r -> Bool
```

The main change is now that the `GEq` class is parameterised over a type of kind `Representation`, making it clear for which types we may be expected to define instances. However, `geq` is now also ambiguously typed: the `Interpret` type family is not injective, i.e., several representations map to the same result type. This is what simplified our conversion functions. The price is that GHC cannot infer the choice of `r` in calls to `geq`, and we have to use explicit type applications.

Had we used a data family or a GADT, GHC could have performed this inference, but at the price of having to keep the `Con` and `Wrap` types around. This trade-off is not clear cut, but I think that generally, the type arguments on the `geq` function are good for clarity, and that function does only occur internally, so it does not affect the use of `eq` which constitutes the interface to the end user.

Apart from the type arguments on recursive calls and the slightly changed types we operate on, the instances of `GEq` do not change much:

```
instance (GEq r, GEq s) => GEq (Sum r s) where
  geq (Left a1)  (Left a2)  = geq @r a1 a2
  geq (Right b1) (Right b2) = geq @s b1 b2
  geq _          _          = False
instance (GEq r) => GEq (Constructor n r) where
  geq = geq @r
```

```
instance (GEq r, GEq s) => GEq (Product r s) where
  geq (a1, b1) (a2, b2) = geq @r a1 a2 && geq @s b1 b2
instance GEq Unit where
  geq () () = True
instance Eq a => GEq (Atom a) where
  geq = (==)
```

To get a bit more practice, let us rewrite `enum` as well:

```
enum :: forall a. (Generic a, GEnum (Code a)) => [a]
enum = to <$> genum @(Code a)
class GEnum (r :: Representation) where
  genum :: [Interpret r]
instance (GEnum r, GEnum s) => GEnum (Sum r s) where
  genum = (Left <$> genum @r) ++ (Right <$> genum @s)
instance (GEnum r) => GEnum (Constructor n r) where
  genum = genum @r
instance GEnum Unit where
  genum = [()]
```

We'll refrain from adapting `conName` and `conIx` at this point, because the changes required are rather systematic and do not depend much on the function in question.

Reflecting briefly, we have clarified the situation slightly by making the role of the classes that are used to define generic functions a bit more precise. But a lot of the subtleties about the invariants about the way in which the representations are being composed remain.

Before we address that, though, there is another interesting avenue worth exploring enabled by the fact that we are now using a closed kind.

2.2 From `class` to `case`

Now that we have a *closed* set of types of kind `Representation`, could we perhaps define functions using a normal `case` expression, rather than via a type class?

It turns out this is possible, by using the standard technique of a singleton type. So let us create a singleton type for the `Representation` kind:

```
data SRepresentation (r :: Representation) where
  SSum         :: SRepresentation r -> SRepresentation s
                  -> SRepresentation (Sum r s)
```

```
  SConstructor :: KnownSymbol n => Proxy n -> SRepresentation r
                    -> SRepresentation (Constructor n r)
  SProduct     :: SRepresentation r -> SRepresentation s
                    -> SRepresentation (Product r s)
  SUnit        :: SRepresentation Unit
  SAtom        :: SRepresentation (Atom a)
```

The relevant property here is that for every type `r` of kind `Representation`, there is exactly one value of type `SRepresentation r`, so by pattern matching on a value of type `SRepresentation r`, we can reveal knowledge about `r`.

An interesting question is what to do with the occurrences of other kinds, such as the name of the constructor `n` in `Constructor` and the type `a` in `Atom`. It turns out that for `n`, it is always reasonable to require `KnownSymbol n`, so that – if needed – be can invoke `symbolVal` for that symbol. We include a `Proxy` for convenience, so that this becomes easier to do. For `Atom`, the situation is less clear. Therefore, let us see first where the current approach of just ignoring the `a` fails while trying to rewrite a function such as generic equality by pattern matching on `SRepresentation`:

```
geq :: SRepresentation r -> Interpret r -> Interpret r -> Bool
geq (SSum r _)         (Left a1)  (Left a2)  = geq r a1 a2
geq (SSum _ s)         (Right b1) (Right b2) = geq s b1 b2
geq (SSum _ _)         _          _          = False
geq (SConstructor _ r) a1         a2         = geq r a1 a2
geq (SProduct r s)     (a1, b1)   (a2, b2)   = geq r a1 a2 && geq s b1 b2
geq SUnit              ()         ()         = True
geq SAtom              a1         a2         = a1 == a2   -- fails
```

We cannot complete the case for `SAtom` because there is no instance `Eq a` in scope. And of course, adding an `Eq` constraint into the definition of `SAtom` is not a good idea: the `SRepresentation` type is supposed to be universal, and not specific to the definition of a single generic function.

However, what we *can* do is to parameterise the `SRepresentation` type over a constraint to be required in the `SAtom` case:

```
data SRepresentation (c :: Type -> Constraint) (r :: Representation) where
  SSum         :: SRepresentation c r -> SRepresentation c s
                    -> SRepresentation c (Sum r s)
  SConstructor :: KnownSymbol n => Proxy n -> SRepresentation c r
                    -> SRepresentation c (Constructor n r)
  SProduct     :: SRepresentation c r -> SRepresentation c s
                    -> SRepresentation c (Product r s)
```

```
  SUnit        :: SRepresentation c Unit
  SAtom        :: c a => SRepresentation c (Atom a)
```

We can now complete the definition:

```
geq :: SRepresentation Eq r -> Interpret r -> Interpret r -> Bool
geq (SSum r _)         (Left a1)  (Left a2)  = geq r a1 a2
geq (SSum _ s)         (Right b1) (Right b2) = geq s b1 b2
geq (SSum _ _)         _          _          = False
geq (SConstructor _ r) a1         a2         = geq r a1 a2
geq (SProduct r s)     (a1, b1)   (a2, b2)   = geq r a1 a2 && geq s b1 b2
geq SUnit              ()         ()         = True
geq SAtom              a1         a2         = a1 == a2
```

In order to invoke `geq`, we now need a value of type `SRepresentation`. For this, we define a type class again, of types of kind `Representation` that have a corresponding singleton (i.e., all such types):

```
class IsRepresentation (c :: Type -> Constraint) (r :: Representation) where
  representation :: SRepresentation c r
```

The instances are all straight-forward and just invoke the corresponding constructor of `SRepresentation`:

```
instance (IsRepresentation c r, IsRepresentation c s)
            => IsRepresentation c (Sum r s) where
  representation = SSum representation representation
instance (KnownSymbol n, IsRepresentation c r)
            => IsRepresentation c (Constructor n r) where
  representation = SConstructor Proxy representation
instance (IsRepresentation c r, IsRepresentation c s)
            => IsRepresentation c (Product r s) where
  representation = SProduct representation representation
instance IsRepresentation c Unit where
  representation = SUnit
instance c a => IsRepresentation c (Atom a) where
  representation = SAtom
```

Given this, we can now define

```
eq :: forall a. (Generic a, IsRepresentation Eq (Code a)) => a -> a -> Bool
eq a1 a2 = geq (representation @Eq @(Code a)) (from a1) (from a2)
```

Is this version in any way better than the previous one? Well, perhaps it feels slightly more lightweight. We do not have to define any new type class (note that `IsRepresentation` has to be defined only once), making the look of `geq` much more compact.

We buy this with a possible performance hit, because the type class resolution was happening at compilation time, making inlining and partial evaluation more likely, whereas `geq` is just a recursive function that will not be inlined.

A lingering question might be whether abstraction over a single constraint such as we have done for `SRepresentation` and `IsRepresentation` is a good idea. After all, what if we need two class constraints, or none? Fortunately, this does not pose a problem, because GHC's type class language easily allows us to express these situations via a single constraint:

```
class Top a
instance Top a

class (c a, d a) => And c d a
instance (c a, d a) => And c d a
```

We can use `Top` in a situation where no actual class constraint is required, and we can use `And` to combine several constraints into one.

Exercise 11. Port the generic function `total` from Exercise 5 to this setting.

2.3 Partial generic functions

There is another new problem, though, which is revealed if we try to write `enum` in pattern-matching style:

```
enum :: forall a. (Generic a, IsRepresentation Top (Code a)) => [a]
enum = to <$> genum (representation @Top @(Code a))

genum :: SRepresentation Top r -> [Interpret r]
genum (SSum r s)         = (Left <$> genum r) ++ (Right <$> genum s)
genum (SConstructor _ r) = genum r
genum SUnit              = [()]
```

The above definition corresponds to the three class instances we had defined for `genum` before. But what about the other cases? GHC will warn us now that we have "non-exhaustive patterns". And indeed, if we try to invoke `enum` on a type like `Tree`, we will now no longer get a static error, but a run-time exception.

This seems quite bad. We set out to achieve more structure and clarity, and instead, it seems, we have reached a point that is just worse.

Exercise 12. Can you define `conName` and `conIx` in this setting? Do they also suffer from the partiality problem?

2.4 Specifying enumeration types

However, all is not lost. The fact that we have the explicit `Representation` kind gives us new options. The function `enum` is supposed to work only for enumeration types. Perhaps we can explicitly state what it means to be an enumeration type.

We do this by employing a perhaps still somewhat underused technique: a type family that computes a constraint.

```
type family IsEnumRepresentation (r :: Representation) :: Constraint where
  IsEnumRepresentation (Sum r s)         =
    (IsEnumRepresentation r, IsEnumRepresentation s)
  IsEnumRepresentation (Constructor n r) = r ~ Unit
```

The above rather precisely captures what we consider to be an enumeration type. The absence of cases in the type family actually states that such constructors must not occur in the codes of enumeration types. All we admit is a `Sum`-structure with `Constructor`s underneath, and underneath a constructor there *must* be a `Unit`. We do not need a case for `Unit` itself here, because we do not recurse in `Constructor`, and even a single-constructor datatype would still start with a `Constructor` occurrence.

```
type IsEnumType a = (Generic a, IsEnumRepresentation (Code a))

enum :: forall a. (IsEnumType a, IsRepresentation Top (Code a)) => [a]
enum = to <$> genum (representation @Top @(Code a))

genum :: IsEnumRepresentation r => SRepresentation Top r -> [Interpret r]
genum (SSum r s)         = (Left <$> genum r) ++ (Right <$> genum s)
genum (SConstructor _ _) = [()]
```

We can simplify `genum` slightly now, removing the case for `SUnit` and returning a list containing `()` directly from the constructor case. The added constraint `r ~ Unit` from the `IsEnumRepresentation` family makes this type-correct.

The new version now fails again statically if we try to invoke it on a non-enumeration type such as `Tree`, because that type's code does not statisfy the constraint `IsEnumRepresentation`. However, GHC still complains about non-exhaustive patterns, with the pattern match checker currently not being able to see that the other cases are impossible.

And indeed, we can yet do better.

Exercise 13. Use GHC's `TypeError` type family to provide a better error message if `IsEnumRepresentation` fails.

Exercise 14. Define `IsProductRepresentation`, similar to `IsEnumRepresentation`, so that it succeeds only for product types.

2.5 Stratifying the codes

We know that certain invariants are supposed to hold for *all* representations, not just for representations corresponding to specific subclasses of types such as enumeration types. In particular, we know that there is going to be no arbitrary mixing of the sum and the product structures of a datatype. There is always going to be a choice between constructors first, and *then* a combination of constructor fields.

Therefore, let us split the `Representation` kind into two kinds:

```
data SumRepresentation =
      Sum SumRepresentation SumRepresentation
    | Constructor Symbol ProductRepresentation
data ProductRepresentation =
      Product ProductRepresentation ProductRepresentation
    | Unit
    | Atom Type
```

There is quite a bit of work following up from this. First of all, `Interpret` has to be split as well then:

```
type family InterpretSum (r :: SumRepresentation) :: Type where
  InterpretSum (Sum r s)         = Either (InterpretSum r) (InterpretSum s)
  InterpretSum (Constructor n r) = InterpretProduct r
type family InterpretProduct (r :: ProductRepresentation) :: Type where
  InterpretProduct (Product r s) = (InterpretProduct r, InterpretProduct s)
  InterpretProduct Unit          = ()
  InterpretProduct (Atom a)      = a
```

If we define

```
type Representation = SumRepresentation
type Interpret r = InterpretSum r
```

we can leave the `Generic` class unchanged. But we have to split `SRepresentation` and `IsRepresentation`. This is all entirely straight-forward, but we include the code here for completeness:

```
data SSumRepresentation
       (c :: Type -> Constraint) (r :: SumRepresentation) where
```

```
  SSum         :: SSumRepresentation c r -> SSumRepresentation c s
                  -> SSumRepresentation c (Sum r s)
  SConstructor :: KnownSymbol n => Proxy n -> SProductRepresentation c r
                  -> SSumRepresentation c (Constructor n r)

data SProductRepresentation
       (c :: Type -> Constraint) (r :: ProductRepresentation) where
  SProduct     :: SProductRepresentation c r -> SProductRepresentation c s
                  -> SProductRepresentation c (Product r s)
  SUnit        :: SProductRepresentation c Unit
  SAtom        :: c a => SProductRepresentation c (Atom a)

class IsSumRepresentation
       (c :: Type -> Constraint) (r :: SumRepresentation) where
  sumRepresentation :: SSumRepresentation c r

instance (IsSumRepresentation c r, IsSumRepresentation c s)
            => IsSumRepresentation c (Sum r s) where
  sumRepresentation = SSum sumRepresentation sumRepresentation

instance (KnownSymbol n, IsProductRepresentation c r)
            => IsSumRepresentation c (Constructor n r) where
  sumRepresentation = SConstructor Proxy productRepresentation

class IsProductRepresentation
        (c :: Type -> Constraint) (r :: ProductRepresentation) where
  productRepresentation :: SProductRepresentation c r

instance (IsProductRepresentation c r, IsProductRepresentation c s)
            => IsProductRepresentation c (Product r s) where
  productRepresentation = SProduct productRepresentation productRepresentation

instance IsProductRepresentation c Unit where
  productRepresentation = SUnit

instance c a => IsProductRepresentation c (Atom a) where
  productRepresentation = SAtom
```

We then turn to the generic functions. We have to to split `geq` into two functions as well:

```
eq :: forall a. (Generic a, IsSumRepresentation Eq (Code a))
         => a -> a -> Bool
eq a1 a2 = geqSum (sumRepresentation @Eq @(Code a)) (from a1) (from a2)

geqSum :: SSumRepresentation Eq r
             -> InterpretSum r -> InterpretSum r -> Bool
```

```
geqSum (SSum r _)         (Left a1)  (Left a2)  = geqSum r a1 a2
geqSum (SSum _ s)         (Right b1) (Right b2) = geqSum s b1 b2
geqSum (SSum _ _)         _          _          = False
geqSum (SConstructor _ r) a1         a2         = geqProduct r a1 a2

geqProduct :: SProductRepresentation Eq r
                -> InterpretProduct r -> InterpretProduct r -> Bool
geqProduct (SProduct r s) (a1, b1) (a2, b2) = geqProduct r a1 a2
                                              && geqProduct s b1 b2
geqProduct SUnit          ()       ()       = True
geqProduct SAtom          a1       a2       = a1 == a2
```

The payoff comes with `genum`, where a single function now suffices, and that function is now obviously total again:

```
enum :: forall a. (IsEnumType a, IsSumRepresentation Top (Code a)) => [a]
enum = to <$> genum (sumRepresentation @Top @(Code a))

genum :: IsEnumRepresentation r =>
           SSumRepresentation Top r -> [InterpretSum r]
genum (SSum r s)         = (Left <$> genum r) ++ (Right <$> genum s)
genum (SConstructor _ _) = [()]
```

Potentially, there is an even bigger payoff though. Consider `geqProduct` above: conceptually, what we are doing there is to "zip" two product structures of the same shape together, combining the atoms that are the elements with (`==`) in every position, and combining the `Boolean` results with (`&&`). Perhaps looking at operations on the sum and product structure separately enables us to reveal the computations that we are performing to be instances of higher-level patterns that we can extract and give names to.

We will explore this avenue further in the rest of this text.

2.6 Sums and Products

Let us take another look at our stratified codes:

```
data SumRepresentation =
     Sum SumRepresentation SumRepresentation
   | Constructor Symbol ProductRepresentation

data ProductRepresentation =
     Product ProductRepresentation ProductRepresentation
   | Unit
   | Atom Type
```

Both structures are actually quite similar: in both, we have a binary operator that combines two structures, and an operation to embed something: `Constructor` embeds the products structure, and `Atom` embeds a normal Haskell type. The main difference seems to be that we have no counterpart for `Unit` in the sum structure – but that is actually mostly an oversight on our part. We can, and should, add `Empty` as a neutral element of the sum structure, so that we have a way to represent datatypes with no constructors. As such datatypes are comparatively rare, it is perhaps not that much of a surprise that we have not done this until now.

Another observation is that that even our stratified representation still provides us with more degrees of freedom than we might want: we said before that generic functions should not rely on the way that the balancing of the sum or the product structure is done. But if we do not want the balancing to make a difference, then why not fix one particular arrangement that is convenient to operate with, for example a list-like structure?

For products, this would mean that we would look at them as a list of atoms rather than a binary tree of atoms. For sums, we would take the view that they provide a list of choices rather than a tree of choices.

Here is how this could look like:

```
data SumRepresentation =
      ConsSum ProductRepresentation SumRepresentation
    | NilSum
data ProductRepresentation =
      ConsProduct Type ProductRepresentation
    | NilProduct
```

I have removed the constructor names for the time being, in order to emphasise the symmetry between the two structures even more. We will discuss later how to bring them back.

Let us see how this affects interpretation and the `Generic` instance for trees for illustration:

```
type family InterpretSum (r :: SumRepresentation) :: Type where
  InterpretSum (ConsSum r s) = Either (InterpretProduct r) (InterpretSum s)
  InterpretSum (NilSum)      = Void
type family InterpretProduct (r :: ProductRepresentation) :: Type where
  InterpretProduct (ConsProduct a s) = (a, InterpretProduct s)
  InterpretProduct NilProduct        = ()
instance Generic (Tree a) where
  type Code (Tree a) =
```

```
    ConsSum (ConsProduct a NilProduct)
      (ConsSum (ConsProduct (Tree a) (ConsProduct (Tree a) NilProduct))
        NilSum)
  from :: Tree a -> Rep (Tree a)
  from (Leaf x)   = Left (x, ())
  from (Node l r) = Right (Left (l, (r, ())))
  to :: Rep (Tree a) -> Tree a
  to (Left (x, ()))              = Leaf x
  to (Right (Left (l, (r, ())))) = Node l r
```

Not all that much has changed: we just changed the nesting of the various constructs somewhat, and shuffled things around a bit. But the essence is the same, and all the rest of the development would go through as before.

2.7 Lists of lists

At this time, `SumRepresentation` and `ProductRepresentation` are so close to the shape of lists that it seems like we should just try to use the standard promoted type-level lists:

```
type SumRepresentation     = [ProductRepresentation]
type ProductRepresentation = [Type]
```

Everything else works as before; but the right hand side of `Code` in the `Generic` instances gets substantially more readable.

```
type family InterpretSum (r :: SumRepresentation) :: Type where
  InterpretSum (r : s) = Either (InterpretProduct r) (InterpretSum s)
  InterpretSum '[]     = Void
type family InterpretProduct (r :: ProductRepresentation) :: Type where
  InterpretProduct (a : s) = (a, InterpretProduct s)
  InterpretProduct '[]     = ()
instance Generic (Tree a) where
  type Code (Tree a) = '[ '[a], '[Tree a, Tree a] ]
  from :: Tree a -> Rep (Tree a)
  from (Leaf x)   = Left (x, ())
  from (Node l r) = Right (Left (l, (r, ())))
  to :: Rep (Tree a) -> Tree a
  to (Left (x, ()))              = Leaf x
  to (Right (Left (l, (r, ())))) = Node l r
```

We can try to extend the uniformity of our current representation to the singletons. Rather than having two different singleton types for sums and products, let us see if we can do with a single `SList` type:

```
data SList (c :: a -> Constraint) (xs :: [a]) where
  SCons :: c x => SList c xs -> SList c (x : xs)
  SNil :: SList c '[]
class IsList (c :: a -> Constraint) (xs :: [a]) where
  list :: SList c xs
instance (c x, IsList c xs) => IsList c (x : xs) where
  list = SCons list
instance IsList c '[] where
  list = SNil
```

This works because we can nest the calls to `IsList`:

```
type IsRepresentation c xss = IsList (IsList c) xss
```

The constraint `IsRepresentation` expresses that all of the *inner* lists satisfy constraint `c`, which is what we typically need for a sum-of-products structure.

2.8 Zipping products

Consider `geqProduct` in the current setup:

```
geqProduct :: SList Eq xs
                -> InterpretProduct xs -> InterpretProduct xs -> Bool
geqProduct (SCons xs) (a1, r1) (a2, r2) = a1 == a2 && geqProduct xs r1 r2
geqProduct SNil       ()       ()       = True
```

We are zipping two products together using a binary operator (`==`), and can try to abstract from that. For example like this:

```
zipProduct :: SList c xs -> (forall x. c x => x -> x -> r)
                -> InterpretProduct xs -> InterpretProduct xs -> [r]
zipProduct (SCons xs) op (a1, r1) (a2, r2) = op a1 a2
                                               : zipProduct xs op r1 r2
zipProduct SNil       _  ()       ()       = []
```

We could then rewrite `geqProduct` as follows:

```
geqProduct :: SList Eq xs
                -> InterpretProduct xs -> InterpretProduct xs -> Bool
geqProduct xs a1 a2 = and (zipProduct xs (==) a1 a2)
```

This is nice, because it is plausible we could reuse an operation such as `zipProduct` in *other* generic functions. However, `zipProduct` still feels a bit ad-hoc, because it treats the inputs and outputs differently: the inputs are product structures, the output is a plain list, because `(==)` produces a constant `Bool` result.

What if we wanted to zip products while e.g. defining a generic semigroup append operation for product types, based on `(<>)`? Compare:

```
(<>) :: Semigroup a => a -> a -> a
(==) :: Eq a => a -> a -> Bool
```

We would then need a different `zipProduct` function, one which also produces a product, like this:

```
zipProduct' :: SList c xs -> (forall x. c x => x -> x -> x)
   -> InterpretProduct xs -> InterpretProduct xs -> InterpretProduct xs
zipProduct' (SCons xs) op (a1, r1) (a2, r2) =
  (op a1 a2, zipProduct' xs op r1 r2)
zipProduct' SNil       _  ()       ()       = ()
```

And what if we had a binary operation that would result in `Maybe a` or `IO a`? Then we would need yet another zip-like function.

Another option is to try to generalise our product (and sum) structure to capture all these patterns, by allowing each element to be wrapped in an arbitrary type constructor.

Exercise 15. Define `enum` in this setting.

2.9 GADTs for sums and products

To expand on what we just discussed, we would ultimately like to rewrite `Interpret` as follows:

```
type Interpret xss = InterpretSum (InterpretProduct Identity) xss
```

Both `InterpretSum` and `InterpretProduct` gain an additional argument, which is a type constructor. We can use this to explicitly denote here that normally, we want to treat every element of the sum as a product, and every element of the product as

a plain type (`Identity`). By plugging in other type constructors, we can get other interesting behaviour.

For example, a

```
Product (Const Bool) xs
```

would denote a product where every element is a `Bool` (as it might arise as the result of zipping two products using (`==`)), and a

```
Product Maybe xs
```

would be a product where every element is wrapped in an application of `Maybe`.

However, if we try to embed this parameterisation over type constructors into our current approach, it seems we require partial application of the `InterpretProduct` type family, at least – and partial application of type families is not supported by GHC.

Perhaps this is a good moment to explore another option for the interpretations. When we first introduced the interpretation type family, we have been discussing that one could also use a data family or a GADT.

Let us try this, and introduce dedicated GADTs for sums and products. After all, n-ary heterogeneous sums and products seem like structures worthy of having a name of their own. A first approach for sums could look like this:

```
data Sum (xss :: [[Type]]) where
  Zero :: Product xs -> Sum (xs : xss)
  Suc  :: Sum xss -> Sum (xs : xss)
```

Here, `Zero` plays the role of selecting the first out of the given choices (previously done using `Left`), and `Suc` plays the role of selecting one of the remaining choices (if there are choices left; previously done using `Right`).

Note that there is no constructor producing a `Sum` for an empty list. This corresponds to the situation that the last option in each nested `Either` structure was now always `Void`, because we had already adopted a list-like structure.

However, it feels unsystematic to require sums to always lock sums to be indexed by a list of lists of types, and its contents to be a `Product` (which we yet have to define). This is not compatible with our desire to write `Interpret` as outlined above. It seems better to abstract over the contents of a sum *and how to interpret them*, by defining:

```
data Sum (f :: k -> Type) (xs :: [k]) where
  Zero :: f x -> Sum f (x : xs)
  Suc  :: Sum f xs -> Sum f (x : xs)
```

Now `f` can be a product, if we so desire, but it can also be something else. Let us define `Product` following the same approach:

```
data Product (f :: k -> Type) (xs :: [k]) where
  Nil  :: Product f '[]
  Cons :: f x -> Product f xs -> Product f (x : xs)
```

This is easy to recognise as a heterogeneous list, but with every element wrapped in an application of `f`.

Going back to our starting point, we can now say

```
type Interpret xss = Sum (Product Identity) xss
```

Note that `Identity` is a newtype, and our `Sum` and `Product` GADTs have slightly different constructor names, so our `Generic` class changes one final time:

```
type Rep a = Interpret (Code a)

class Generic a where
  type Code a :: [[Type]]
  from :: a -> Rep a
  to :: Rep a -> a
```

Here is the instance for trees:

```
instance Generic (Tree a) where
  type Code (Tree a) = '[ '[a], '[Tree a, Tree a] ]
  from :: Tree a -> Rep (Tree a)
  from (Leaf x)   = Zero (Identity x `Cons` Nil)
  from (Node l r) = Suc (Zero (Identity l `Cons` (Identity r `Cons` Nil)))
  to :: Rep (Tree a) -> Tree a
  to (Zero (Identity x `Cons` Nil))                          = Leaf x
  to (Suc (Zero (Identity l `Cons` (Identity r `Cons` Nil)))) = Node l r
```

Exercise 16. Port the `Generic` instance for `Term` from Exercise 1 to this setting.

2.10 Zipping products, revisited

Recall we previously had defined two versions for zipping product structures: `zipProduct` to zip with an operator producing a result of a constant type, yielding a list of results, and `zipProduct'` to zip with an operator that does not change the type, yielding another product.

With the abstraction over a type constructor in place, we can define a single generalisation of both functions:

```
zipProduct :: SList c xs -> (forall x. c x => f x -> g x -> h x)
                -> Product f xs -> Product g xs -> Product h xs
zipProduct (SCons xs) op (Cons fx fxs) (Cons gx gxs) =
  Cons (op fx gx) (zipProduct xs op fxs gxs)
zipProduct SNil       _  Nil           Nil           = Nil
```

We can now also define a general operation to collapse a product into a list if all its components are in fact of a constant type:

```
collapseProduct :: SList c xs -> Product (Const a) xs -> [a]
collapseProduct (SCons xs) (Cons (Const a) as) = a : collapseProduct xs as
collapseProduct SNil       Nil                 = []
```

We can now define the product part of generic equality in terms of these:

```
geqProduct :: SList Eq xs ->
              Product Identity xs -> Product Identity xs -> Bool
geqProduct xs as bs =
  and (collapseProduct xs (zipProduct xs (\ a b -> coerce (a == b)) as bs))
```

We are now in a situation where we can compose generic functions from more general, high-level functions that operate on the sum and product structures.

Exercise 17. Define a function

```
mapProduct :: SList c xs -> (forall x. c x => f x -> g x)
                -> Product f xs -> Product g xs
```

that maps an operation over a single product, rather than zipping two products. Think of a generic function where this would be useful.

Exercise 18. Define a function

```
collapseSum :: SList c xs -> Sum (Const a) xs -> a
```

that extracts the payload of type `a` from a suitable sum.

2.11 Creating products

Sums and products offer a rich structure, with many useful operations. Introducing them all is beyond the scope of this text, but let us look at a few more examples.

A simpler form of zipping is mapping, which works for both products and sums. With products, we can also *create* a structure by uniformly applying an operation for every component:

```
pureProduct :: SList c xs -> (forall x. c x => f x) -> Product f xs
pureProduct (SCons xs) op = Cons op (pureProduct xs op)
pureProduct SNil       _  = Nil
```

We can, for example, use `zipProduct` and `pureProduct` to generically lift a monoid structure from the components to any product type. Here is how to generalise `mempty`:

```
gmempty :: forall a xs. (Generic a, Code a ~ '[xs], IsList Monoid xs) => a
gmempty = to (Zero (pureProduct (list @_ @Monoid) (Identity mempty)))
```

This is a one-liner now! Note that we require here that the type is a product type by stating that the outer list of the code must have a single element. We can therefore safely apply the first constructor `Zero` generically.

We could further abstract here and define `IsProductType` in a similar style as we defined `IsEnumType` earlier:

```
type IsProductType a xs = (Generic a, Code a ~ '[xs])
```

Exercise 19. Define the generalisation of the monoidal (or semigroup) append operation to product types in the same style as `gmempty`.

2.12 Generic enumeration revisited

This is how we could define `enum` directly in our current style:

```
type IsEnumType a = (Generic a, IsList ((~) '[]) (Code a))

enum :: IsEnumType a => [a]
enum = to <$> genum (list @_ @((~) '[]))

genum :: SList ((~) '[]) xss -> [Sum (Product Identity) xss]
genum (SCons xs) = Zero Nil : (Suc <$> genum xs)
genum SNil       = []
```

This version is already rather nice. However, we can still ask the question whether there are more general concepts at play here.

We can, for example, provide a general operation that produces all injections into a sum, as a product – this is a generalisation of `genum`, because each of these injections can still be applied to arguments:

```
newtype Injection f xs x = Injection (f x -> Sum f xs)

injections :: forall c f xs. SList c xs -> Product (Injection f xs) xs
injections (SCons xs) =
  Cons (Injection Zero) (mapProduct xs shiftInjection (injections @c @f xs))
injections SNil       = Nil

shiftInjection :: Injection f xs x -> Injection f (y : xs) x
shiftInjection (Injection inj) = Injection (Suc . inj)
```

Here, we are using `mapProduct` which we have not explicitly shown.

Now if we have suitable arguments for all the injections as a product, we can apply them:

```
applyInjections :: forall c f xs. SList c xs
                    -> Product f xs -> Product (Const (Sum f xs)) xs
applyInjections xs args =
  zipProduct xs
    (\ (Injection inj) arg -> Const (inj arg))
    (injections @c @f xs) args
```

Applying the injections is itself another applications of `zipProduct`.

Now `genum` is a special case of `applyInjections` where the product of arguments is empty in every case. We can reuse `pureProduct` to create this product easily:

```
genum :: SList ((~) '[]) xss -> [Sum (Product Identity) xss]
genum xss =
  collapseProduct xss (applyInjections xss (pureProduct xss Nil))
```

Exercise 20. Given

```
data Projection f xs x = MkProjection (Product f xs -> f x)
```

define

```
projections :: forall c f xs. SList c xs -> Product (Projection f xs) xs
```

similar to `injections`, only here to compute all the projections from a product.

Exercise 21. Can you use `projections` to define a product of "getters" generically for Haskell product datatypes?

2.13 About metadata

Somewhere along the way, we dropped the constructor names. How can we bring them back, to write functions such as `conName` (or, of course, more realistic functions that make use metadata)? Do we have to give up the relative simplicty of our current codes as lists of lists of types, to add the symbols back in?

The answer is no. We have so much information about the *shape* of the datatypes now that we can supply metainformation separately, and merge it in when needed.

For example, we can provide the names of the constructors just as a product of the right size:

```
class Generic a => HasDatatypeInfo a where
  conNames :: Proxy a -> Product (Const String) (Code a)
```

Now, this is an additional class next to `Generic` that we have to manually instantiate for each datatype (or via Template Haskell, or via built-in GHC support), but given that this information was previously contained in `Generic` itself, this is not significantly more work than before:

```
instance HasDatatypeInfo (Tree a) where
  conNames _ = Const "Leaf" `Cons` (Const "Node" `Cons` Nil)
instance HasDatatypeInfo Colour where
  conNames _ =
    Const "Red" `Cons` (Const "Green" `Cons` (Const "Blue" `Cons` Nil))
```

Our function `conName` can then be written in terms of a – generally useful – function `select` that combines a product with a sum by selecting the component of the product that is indicated by the sum and combines the payloads as needed:

```
select :: SList c xs -> (forall x. c x => f x -> g x -> h x)
            -> Product f xs -> Sum g xs -> Sum h xs
select (SCons xs) op (Cons fx fxs) (Zero gx) = Zero (op fx gx)
select (SCons xs) op (Cons fx fxs) (Suc gxs) = Suc (select xs op fxs gxs)
conName :: forall c a. (HasDatatypeInfo a, IsList c (Code a)) => a -> String
conName a = collapseSum xs (select xs const (conNames (Proxy @a)) (from a))
  where
    xs = list @_ @c
```

This is using `collapseSum` from Exercise 18.

Exercise 22. Port `conIx` to this setting.

Exercise 23. Define a parser for enumeration types in this setting, as in Exercise 6.

2.14 Goal: `generics-sop`

We are now very close to what `generics-sop` is doing. Once again, the major discrepancies are in the naming.

What we call `Product` and `Sum`, `generics-sop` calls `NP` and `NS`, respectively. What we call `IsList` is called `All` in `generics-sop`. The explicit singleton form `SList` does not carry a constraint parameter in `generics-sop`. The operations come with constrained and unconstrained versions, and there are also overloaded versions available that work over sums, products and even sums of products. Also, there is systematic handling of metadata available.

However, apart from these minor details, the approach we have arrived at and `generics-sop` are the same.

Of course, developing `generics-sop` from scratch is, in a way, easier and more direct than the journey that we have taken. On the other hand, I hope to have made the relation to `generics-sop` a bit clearer.

Also, many of the intermediate steps (most of which can be applied in a different order as well) lead to interesting design points in the generic programming space as well, that have perhaps not been sufficiently explored. A special mention should perhaps go to `simplistic-generics`, which is a recent library offering one of these intermediate points, where we can define generic functions via `case` using a closed description of the representation types, but binary sums and products are being used just as in `GHC.Generics`.

The strength of `generics-sop` is the conciseness of the resulting generic functions, the clear structure being provided, and the encouragement of using combinators and reusing code between different generic functions.

Its potential weakness is efficiency: while `GHC.Generics` is using classes prominently and has some hope of being inlined, `generics-sop` in the current form uses term-level patter matching and chains of recursive functions on GADT-based representations, with basically no hope of optimising any of this overhead away. By yet more variations in the design, we could try to recover some of the advantages `GHC.Generics` has in this regard. However, I am convinced that not even `GHC.Generics` has a satisfactory story for optimisation, and if one wants reliable performance of generic functions, one has to apply *staging*. There is a version of `generics-sop` in the works that employs *Typed Template Haskell* to combine its high-level interface with guaranteed optimisation behaviour, where the code written will in general be equally fast as hand-written code. But this is a story for another day (and, due to some changes being required, unfortunately, also for a future GHC release).

Exercise 24. Define generic equality and enumeration in actual `generics-sop`.

Exercise 25. Find out how metadata is handled in `generics-sop` and define a function that makes use of metadata.

Exercise 26. Take a look at `simplistic-generics` and find out where approximately fits within the journey we have taken.

Exercise 27. Define at least one more datatype-generic programming approach by combining the steps we have seen in a different way or different order.

For example, can you define a generic programming approach in the style of `GHC.Generics`, i.e., one that still uses type-class based dispatching, but that still makes use of a list-like sum and product encoding, and that uses an explicit layering?