# Applying Type-Level and Generic Programming in Haskell

Andres Löh, Well-Typed LLP

September 3, 2015

# Contents

# 1 Introduction and type-level programming

*Please note that this text is still somewhat unfinished and rough in a few parts. Feedback and suggestions for improvement are always welcome.*

## 1.1 Motivation

### 1.1.1 What is datatype-generic programming?

The idea of datatype-generic programming is as follows. Assume you have a datatype `A` and consider a function on that datatype such as structural equality:

```
eq_A :: A -> A -> Bool
```

Given the definition of `A`, it's very easy and straight-forward to give the definition of $\texttt{eq}_{\texttt{A}}$. That's because there is an algorithm that we can describe informally: if the datatype admits different choices (i.e., *constructors*), then check that the two arguments are of the same constructor. If they are, then check the arguments of the constructor for equality pointwise.

If we can phrase the algorithm informally, we'd like to also be able to write it down formally. (Haskell allows to say `deriving Eq` on a datatype declaration, invoking compiler magic that conjures up a suitable definition of equality for that datatype, but that isn't a formally defined algorithm and only works for a handful of functions.)

So here is the general idea, expressed in terms of Haskell: assume there is a type class

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

that associates with a type `a` an isomorphic representation type `Rep a`, witnessed by the conversion functions `from` and `to`. Now if all `Rep a` types have a common structure, we can define a generic function

```
geq :: Generic a => Rep a -> Rep a -> Bool
```

that works for all representation types by induction over this common structure.

Then we can use the isomorphism and `geq` to get an equality function that works on any representable type:

```
eq :: Generic a => a -> a -> Bool
eq x y = geq (from x) (from y)
```

We can then build a limited amount of compiler magic into the language to derive the `Generic` instance for each representable type automatically, i.e., to automatically come up with a structural representation type the conversion functions, and we have reduced the need for having an

arbitrary amount of magically derivable type classes to the need to have just a single magically derivable type class, called `Generic`.

There is still a lot of flexibility in the details of how exactly the class `Generic` is defined, and even more importantly, what kind of representation we choose for `Rep`. This choice is often called the generic *view* or *universe*, and it is the main way in which the countless generic programming approaches that exist for Haskell (SYB, RepLib, instant-generics, regular, multirec, generic-deriving, generic-sop) are different from one another. This choice can have a significant effect:

- Depending on the choice of universe, some datatypes may be representable and others may not.
- Depending on the choice of universe, some generic functions may be definable and others may not definable (or more difficult to define, or less efficient).

In the context of this lecture series, we are going to look at one such universe, which is relatively recent, called generics-sop. I think it is quite elegant and easy to use, while still being expressive. We will also briefly look at the relation to other universes, and I will take that opportunity to argue that despite the arguments I just made, the choice of universe is less important than one might think.

In order to work with generics-sop, one has to use quite a few type system extensions offered by GHC, and it helps to understand the concepts involved. One may see this as a disadvantage, but here, we're going to see it as a welcome challenge: being able to program on the type level in GHC, while not a silver bullet, is an exciting area in itself, and useful for many purposes beyond datatype-generic programming, so we'll see it as a useful opportunity to familiarize ourselves with the necessary concepts.

Apart from the knowledge barrier, generics-sop is ready for use. It is released, available on Hackage, and is being used.

### 1.1.2 Applications of generic programming

There are many ways in which datatype-generic programs can be usefully implied. Next to equality and comparison functions, there are a large class of translation functions that translate between values of datatypes and other formats, such as human-readable text, binary encodings or data description languages such as JSON, XML or similar. Typical generic functions include also all sorts of navigation or traversal functions that access certain parts of values while leaving others alone, such as lenses.

In the latter part of this lecture series, we will discuss a number of applications of generics-sop.

### 1.1.3 A uniform view on data

We are now going to explain the main motivation behind the structural representation of Haskell datatypes that is chosen by generics-sop.

Let us start by looking at a few datatypes in Haskell:

```haskell
data Maybe a    = Nothing | Just a
data Either a b = Left a | Right b
data Group      = Group Char Bool Int
data Expr       = NumL Int | BoolL Bool | Add Expr Expr | If Expr Expr Expr
```

We observe: Haskell datatypes offer a *choice* between different constructors. Depending on the constructor chosen, we have to provide an appropriate *sequence* of arguments. In other words, each value of such a datatype is a constructor applied to a number of arguments. This is going to be our common structure:

```
C_i x_0 ... x_(n_i - 1)
```

If we know the type of the value, the names of its constructors do not really matter. We can simply number the constructors from `0` to `c - 1` where `c` is the total number of constructors for that datatype. Depending on the chosen number `n`, we know how many arguments $n_i$ we expect and what their types are.

Interestingly, this representation is quite close to the run-time representation of Haskell values. GHC stores each value starting with a pointer to some information that, among other things, contains a tag indicating the number of the constructor. Following this info pointer there is the "payload", which is just a sequence of pointers to the constructor arguments.

So it is plausible that something similar might work for us. But we need to assign types to this representation if we want to work with it on the Haskell surface level. Which means that we have to formalize concepts such as "the number of the constructor determines the number of arguments and their types".

Haskell does not have "full-spectrum" dependent types, but it is well-known that we can get very close with various GHC extensions. We will model the sequence of arguments of a constructor as a typed heterogeneous list. Then we will model the choice between the constructors as something like a pointer into a heterogeneous list. In the following, we will introduce the GHC concepts that we will rely on, such as GADTs, data kinds and constraint kinds.

Since type-level programming is a bit peculiar in Haskell, we'll move step by step: from normal list over length-indexed vectors to heterogeneous lists and ultimately the slightly more generalized *n*-ary products that we are planning to use. Several concepts will be introduced in the slightly simpler setting of length-indexed vectors before we try to apply them to *n*-ary products.

## 1.2 Kinds and data kinds

Haskell has a layered type system. Types are defined using a much more limited language than terms. Perhaps the most obvious omission is the lack of a plain lambda abstraction on the type level. Nevertheless, types are structured, and there is the possibility to apply types to one another, and to parameterize named types. Therefore, types need their own, non-trivial, types, which are called kinds.

### 1.2.1 Stars and functions

For a long time, the kind system of Haskell was quite uninteresting:

- The kind `*` is for types that classify terms (even though not every type of kind `*` need be inhabited). In particular, whenever we define a new datatype using `data`, its fully applied form is of kind `*`. This also implies that kind `*` is *open*. New members of kind `*` can be added via `data` at any time.
- If `k` and `l` are kinds, we can form the function kind `k -> l` to indicate types of kind `l` that are parameterized over types of kind `k`.

Examples:

- The types `Int`, `Double`, `Bool`, and `Char` are all of kind `*`.
- The types `Maybe`, `[]`, and `IO` are of kind `* -> *`, but e.g. `Maybe Int`, `[Double]` or `IO Bool` are of kind `*` again.
- The types `Either` or `(,)` are of kind `* -> * -> *`. They can be partially applied, so e.g. `Either Int` is of kind `* -> *`. And `Either Int Char` or `(Bool, Double)` are of kind `*` again.
- We can use the kind system to tell that `Either (Maybe [Int]) (Char, Bool)` is well-formed, but `(Maybe, IO)` is not.

### 1.2.2 Promoted data kinds

Nowadays, Haskell has many more kinds than just the above. In fact, we can define new kinds ourselves, via datatype promotion. Whenever we define a new datatype, we can *also* use it as a kind.

Example:

```
data Bool = False | True
```

This defines the datatype `Bool` with (data) constructors

```
False :: Bool
True  :: Bool
```

But promotion allows us to use everything also one level up, so `Bool` is a new *kind* with *type constructors*

```
'False :: Bool
'True  :: Bool
```

Just like there are only two values of type `Bool` (ignoring `undefined` or "bottom"), there are only two types of kind `Bool`. The quotes can be used to emphasize the promotion and are required where the code would otherwise be syntactically ambiguous, but are often just dropped.

So if `'False` is a type, are there any terms or values of type `'False`? The answer is no! The kind `*` remains the kind of types that classify values. All the other new kinds we now gain by promoting datatypes are uninhabited.

They are, however, still useful, as we shall see, because they can appear as the arguments and results of type-level functions, and they can appear as parameters of datatypes and classes. It's the latter use that we are for now most interested in.

## 1.3 Generalized algebraic data types (GADTs)

We'd like to model heterogeneous lists so that we can represent sequences of arguments of a data constructor as such lists. We'll do so by generalizing normal Haskell lists in several steps.

### 1.3.1 Vectors

The first step is to go to length-indexed lists, also known as *vectors*. For these, let's first define the natural numbers as a datatype as follows:

```
data Nat = Zero | Suc Nat
```

We are going to use this datatype in promoted form, so are going to see `Nat` as a kind, and

```
'Zero :: Nat
'Suc  :: Nat -> Nat
```

(again, the quotes are generally optional) as types. The definition of vectors is then as follows:

```
data Vec (a :: *) (n :: Nat) where
  VNil  :: Vec a Zero
  VCons :: a -> Vec a n -> Vec a (Suc n)
infixr 5 `VCons`
```

This definition uses a different, more flexible syntax for datatype definitions: we define `Vec :: * -> Nat -> *` with two constructors, by listing their types. This syntax gives us the option to restrict the result type. For example, `VNil` does not construct an arbitrary vector `Vec a n`, but only a `Vec a Zero`. Similarly, `VCons` cannot be used to construct a `Vec a Zero`, but only a `Vec a (Suc n)` for some `n`.

Note the role of `Nat` here, as just an index. As we have observed above, there are no inhabitants of type `Nat`, nor is there any need. We just use `Nat` to grant us more information about vectors. In a full-spectrum dependently typed language, we could use the *actual* natural numbers for this. In Haskell, we have to lift them to the type level, but apart from that technicality, the effect is very similar.

Let us look at an example:

```
type Three = Suc (Suc (Suc Zero))
vabc :: Vec Char Three
vabc = 'a' `VCons` 'b' `VCons` 'c' `VCons` VNil
```

The type of `vabc` can also be inferred. The type system keeps track of the length of our vectors, as desired.

It's a bit annoying if we cannot show vectors in GHCi, so let's add a `deriving` declaration that magically gives us a `Show` instance for vectors:

```
deriving instance Show a => Show (Vec a n)
```

The true power of GADTs becomes visible if we consider pattern matching, such as in

```
vtail :: Vec a (Suc n) -> Vec a n
vtail (VCons x xs) = xs
```

or

```
vmap :: (a -> b) -> Vec a n -> Vec b n
vmap f VNil         = VNil
vmap f (VCons x xs) = f x `VCons` vmap f xs
```

In `vtail`, the type signature indicates that we receive a non-empty vector. This means we do not have to match on `Nil` – in fact, doing so would not be type-correct. We can write a safe and total `vtail` function.

In `vmap`, matching on `VNil` tells GHC that for this case `n` must be equal to `Zero`. GHC has its own syntax for type equality: it will learn that `n ~ Zero` holds for this case, and therefore, it will happily accept that we return `VNil` again. Trying to use `VCons` on the right hand side of the `VNil` case would cause a type error.

Similarly, in the `VCons`-case, GHC learns `n ~ Suc n'` for some `n'`. Then, because `vmap` preserves the length, `vmap f xs :: Vec a n'` and `VCons`-ing one element on that will yield a vector of length `n` again.

The type signatures for `vtail` and `vmap` are required. In general, pattern matching on a GADT requires an explicit type annotation.

### 1.3.2 Functions on vectors

Before we move on to heterogeneous lists, let's spend a bit more time trying to reimplement some functions we know from normal lists on vectors. Ultimately, our goal is to do the same for heterogeneous lists, but we'll need a couple of new concepts along the way.

Some of these concepts are needed just generally because we move more information to the type level. Others result from the further generalization from vectors to heterogeneous lists. Looking at vectors in more detail now allows us to separate these issues, rather than having to discuss and understand them all at the same time.

Consider `replicate`:

```
replicate :: Int -> a -> [a]
replicate n x
   | n <= 0    = []
   | otherwise = x : replicate (n - 1) x
```

A call to `replicate n x` creates a list with `n` copies of `x`. How would we do something similar for vectors?

The size of a vector is statically known, so we'll have to assume that the number of copies we want is also known to us at the type level. But how do we pass it? We certainly cannot use an `Int`, because that would live at the term level. Do we need an argument at all? Might we be able to write:

```
vreplicate :: a -> Vec a n
```

Unfortunately, no. This type suggests that the function is parametric in `n`, i.e., that we do not need to look at `n` at all. But we have to make a compile-time distinction based on the type `n`. One option is to use a type class:

```
class VReplicate (n :: Nat) where
  vreplicate :: a -> Vec a n
```

Haskell type classes naturally extend to the presence of data kinds.

Now it's easy to complete the definition of `VReplicate`:

```
instance VReplicate Zero where
  vreplicate _ = VNil
instance VReplicate n => VReplicate (Suc n) where
  vreplicate x = x `VCons` vreplicate x
```

This works. For example

```
GHCi> vreplicate 'x' :: Vec Char Three
```

While using type classes like this is a possibility, there also is a significant disadvantage: when following this pattern, we have to define new type classes for a lot of different functions, and the class constraints will appear in the types of all the functions that use them. In other words, we will leak information that should be internal to the implementation of a function in the types. Once we start optimizing a function or changing the algorithm or reimplementing it in terms of other, more general functions, we might have to adapt its type and therefore lots of other types in our program.

It's therefore worth to consider another option.

## 1.4 Singleton types

### 1.4.1 Singleton natural numbers

We can try to mirror the type of `replicate` more closely by creating a value that does nothing more than to help GHC to know what value of `n` we want. The idea is that we create a GADT `SNat n` such that `SNat n` has exactly one value for each `n`. Then we can use pattern matching on the `SNat n` to learn at run-time what `n` is, and branch on that.

One common approach to achieve this goal is to define:

```
data SNat (n :: Nat) where    -- preliminary
  SZero :: SNat Zero
  SSuc  :: SNat n -> SNat (Suc n)
```

For example,

```
sThree :: SNat Three
sThree = SSuc (SSuc (SSuc SZero))
```

However, we're going to take a slightly different approach here by defining `SNat` and a class that are mutually recursive:

```
data SNat (n :: Nat) where
  SZero :: SNat Zero
  SSuc  :: SNatI n => SNat (Suc n)
class SNatI (n :: Nat) where
  sNat :: SNat n
instance SNatI Zero where
  sNat = SZero
instance SNatI n => SNatI (Suc n) where
  sNat = SSuc
```

Using `SNat` and `SNatI`, we can define `vreplicate` also as follows:

```
vreplicate :: forall a n . SNatI n => a -> Vec a n
vreplicate x = case sNat :: SNat n of
  SZero -> VNil
  SSuc  -> x `VCons` vreplicate x
```

### 1.4.2 Evaluation

Comparing to the original solution of defining one class per function, the singletons approach has the advantage that we need only one class per type that we pattern match on – and we do not even necessarily need the class, if we're willing to pass the singleton explicitly instead. So this gives us rather more interface stability and exposes less of the implementation. There's also a disadvantage though: type class resolution happens at compile time. However, the value of type `SNat` is an actual run-time value, and pattern matching on it happens at run-time. So the singleton-based version of `vreplicate` does unnecessary work at run-time by performing case distinctions for which the outcome was already known at compile time.

A sufficiently optimizing compiler could in principle resolve all this and compile both versions to the same code. In practice, GHC does not, and while this is somewhat annoying, it can in some circumstances also be beneficial, because it actually provides the programmer with a (albeit rather subtle) choice: use type classes and get compile-time resolution (which is usually more efficient, but can actually in extreme cases produce lots of code), or use singleton and get run-time resolution (which usually does unnecessary work at run-time, but can be more compact).

### 1.4.3 Applicative vectors

Using `vreplicate`, vectors in principle admit the interface of an applicative functor:

```
class Functor f => Applicative f where
  pure  :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

It is well known that one possible instance for lists is that where `pure` produces an infinite list of copies, and `(<*>)` zips together the list, applying the list of functions pointwise to the list of arguments.

For vectors, we can implement the same idea, only that of course, the length of all the lists involved is statically fixed. The function `vreplicate` plays the role of `pure`, and

```
vapply :: Vec (a -> b) n -> Vec a n -> Vec b n
vapply VNil          VNil          = VNil
vapply (f `VCons` fs) (x `VCons` xs) = f x `VCons` vapply fs xs
```

is like `(<*>)`.

There's a relatively technical reason why we cannot make `Vec` an actual instance of `Applicative`: The parameters of `Vec` are in the wrong order for that.

## 1.5 Heterogeneous lists

A heterogeneous list is like a vector: a list-like strucuture with a statically known length. But in addition, we also know the type of each element! So indexing by a natural number is not enough. We have to know all the types of all the elements. So we index by a normal list of types.

### 1.5.1 Promoted lists and kind polymorphism

Normal term-level Haskell lists define a type constructor `[]` of kind `* -> *`, with constructors

```
[] :: [a]
(:) :: a -> [a] -> [a]
```

Just like we can promote types like `Bool` and `Nat`, we can also promote lists and treat `[]` as a *kind constructor*, and `[]` and `(:)` as types.

Note that even if seen as a type constructor, `(:)` has kind

```
(:) :: a -> [a] -> [a]
```

This means it is *kind-polymorphic*. We can build a type-level list of promoted Booleans as well as a type-level list of promoted natural numbers:

```
GHCi> :kind [True, False]
[True, False] :: [Bool]
GHCi> :kind [Zero, Three]
[Zero, Three] :: [Nat]
```

For a heterogeneous list, we want a type-level list of types, which has kind `[*]`:

```
GHCi> :kind [Char, Bool, Int]
[Char, Bool, Int] :: [*]
```

The definition is as follows:

```
data HList (xs :: [*]) where
  HNil  :: HList '[]
  HCons :: x -> HList xs -> HList (x ': xs)
infixr 5 `HCons`
```

In this case, the quotes in `'[]` and `':` are actually required by GHC, because we e.g. need to disambiguate between the list type constructor `[] :: * -> *` and the empty type-level list `'[] :: [*]`.

Let's consider our example datatype `Group` again:

```
data Group = Group Char Bool Int
```

Using `HList`, we can produce a sequence of a character, a Boolean, and an integer, as follows:

```
group :: HList '[Char, Bool, Int]
group = 'x' `HCons` False `HCons` 3 `HCons` HNil
```

Once again, we include the type signature for reference, but it could have been inferred.

### 1.5.2 Environments or $n$-ary products

It turns out that we will often need heterogeneous lists where each element is given by applying a particular type constructor to one of the types in the index list. Consider for example a heterogeneous list where all values are optional (`Maybe`), or all values are `IO` actions.

We therefore are going to define the following variant of `HList`, which is additional abstracted over a type constructor `f`, and has elements of the type `f x` where `x` is a member of the index list:

```
data NP (f :: k -> *) (xs :: [k]) where
  Nil  :: NP f '[]
  (:*) :: f x -> NP f xs -> NP f (x ': xs)
infixr 5 :*
```

Here, `NP` is for *n-ary product*, but I sometimes also call this type an *environment*, and the index list `xs` its *signature*.

Note that we've made `NP` kind-polymorphic. We do not require the signature to be of kind `[*]`, but allow it to be an arbitrary type-level list of kind `[k]`, as long as the "interpretation function" `f` maps `k` back to `*`. This is possible, because elements of the signature do not directly appear in an environment, but only as arguments to `f`. It also will turn out to be useful for us in a little while, but let's first consider scenarios where `k` is `*`.

The datatype `NP` is really a generalization of `HList`:

```
newtype I a = I {unI :: a}
```

implements an identity function on types. For any choice of `a` the type `I a` is isomorphic to type `a`.

And the type `NP I` is isomorphic to `HList`, as is witnessed by the following conversion functions:

```
fromHList :: HList xs -> NP I xs
fromHList HNil          = Nil
fromHList (x `HCons` xs) = I x :* fromHList xs

toHList :: NP I xs -> HList xs
toHList Nil         = HNil
toHList (I x :* xs) = x `HCons` toHList xs
```

Interestingly, there is no need to put these conversion functions into type classes (such as `hshow`), because there are no constraints on the type-level list `xs`.

Perhaps surprisingly, `NP` is also a rather direct generalization of (homogeneous) vectors:

```
newtype K a b = K {unK :: a}
```

implements a constant function on types. For any choice of `a` and `b`, the type `K a b` is isomorphic to type `a`.

An environment of type

```
NP (K Double) '[Char, Bool, Int]
```

is a list of three elements of type `Double`. The length of the type-level list determines the number of elements, however the types in the index list are irrelevant, as they're all ignored by `K`.

In practice, the function that turns out to be useful often is one that collapses an `NP` of `K`s into a normal list:

```
hcollapse :: NP (K a) xs -> [a]
hcollapse Nil         = []
hcollapse (K x :* xs) = x : hcollapse xs
```

## 1.6 Higher-rank types

Let's try to define a number of functions on environments. For vectors, we had implemented `vmap`, `vreplicate` and `vapply`, which together gave us (morally speaking) an instance of `Applicative` for vectors. Our aim is to provide a similar interface for environments.

### 1.6.1 Mapping over $n$-ary products

Let's start by looking at a generalization of `vmap`.

```
vmap :: (a -> b) -> Vec a n -> Vec b n
```

Note that `vmap` preserves the length index, but changes the type of elements.

For `NP`, we have no single type of elements, but different types. While we might want to change the type of elements, we better not change it in completely unpredictable ways, otherwise we cannot do anything useful with the list anymore. Here, it turns out to be useful that we've made the step from `HList` to `NP`, because we already have a type constructor `f` that

tells us how to interpret each element in the signature. So let us generalize `vmap` to `hmap` in the following way: the function `hmap` will also preserve the index, but change the type constructor.

This turns out to be very flexible in practice. For example, with just the two instantiations of `f` we have considered so far, `K` and `I`, we can model a type-preserving map as a function

```
NP I xs -> NP I xs
```

and a type-unifying map as a function

```
NP I xs -> NP (K a) xs
```

and a map that actually produces values of different types out of a homogeneous list as a function

```
NP (K a) xs -> NP I xs
```

But note that we can also plug other type constructors into an `NP`, including GADTs, so in principle we can express rather strange maps, as long as we are willing to explain them to the type system.

So let's look at the code we would expect `hmap` to have, and think about the type signature afterwards:

```
hmap m Nil       = Nil
hmap m (x :* xs) = m x :* hmap m xs
```

Ok, so `hmap` clearly takes an environment and produces one again, with equally many elements, as was the plan:

```
hmap :: ... -> NP f xs -> NP g xs
```

Now it seems that we need a function of type `f x -> g x` as an argument. But for what `x`? If you look at the second case, then `m` will be applied to all elements of the list `xs`. So it's not enough if `m` works for one particular `x` – it has to work for at least all the `xs` that appear in `xs`! If `m` even happens to be polymorphic in `x`, then that is clearly good enough:

```
hmap :: (forall x . f x -> g x) -> NP f xs -> NP g xs
```

This is an example of a *rank-2-polymorphic* type. The argument function to `hmap` must itself be polymorphic.

If a Haskell function is polymorphic, the caller of the function has the freedom to choose the type on which to use the function, whereas the callee must make no assumptions whatsoever about the type. If an argument function is polymorphic, the situation is reversed: now the caller has the obligation to pass in a polymorphic function, and the callee may flexibly use it on any type desired.

Let us look at an example (using a `Show` instance for `NP` that we'll only define later):

```
group' :: NP I '[Char, Bool, Int]
group' = I 'x' :* I False :* I 3 :* Nil
```

```
example :: NP Maybe '[Char, Bool, Int]
example = hmap (Just . unI) group'


GHCi> example
```

While we cannot map plain `Just` over `group'`, we can still get quite close. We have to unwrap the elements from the `I` constructor before we can re-wrap them in `Just`.

### 1.6.2 Applicative $n$-ary products

Now that we've seen `hmap`, let's see if we can come up with functions corresponding to `pure` and `<*>` as well.

Let's recall `vreplicate`:

```
vreplicate :: forall a n . SNatI n => a -> Vec a n
vreplicate x = case sNat :: SNat n of
  SZero -> VNil
  SSuc  -> x `VCons` vreplicate x
```

The role of `n` is now played by the signature `xs`, and the role of `a` by the type constructor `f`.

So we need two ingredients to make this work:

- A sufficiently polymorphic value of type `f x` so that it can be used for any `x` in the signature of `xs`. We'll use the type `forall a . f a` here.
- A singleton for lists.

Following the same principles we used for `SNat`, we try to define:

```
data SList (xs :: [k]) where
  SNil  :: SList '[]
  SCons :: SListI xs => SList (x ': xs)
class SListI (xs :: [k]) where
  sList :: SList xs
instance SListI '[] where
  sList = SNil
instance SListI xs => SListI (x ': xs) where
  sList = SCons
```

This isn't quite ideal, because we say nothing about the elements of the list. In principle, we'd want `SCons` to include both a singleton for the head and a singleton for the tail.

For now, we can actually define:

```
hpure :: forall f xs . SListI xs => (forall a . f a) -> NP f xs
hpure x = case sList :: SList xs of
  SNil  -> Nil
  SCons -> x :* hpure x
```

It is reassuring to see that the code is essentially the same as for `vreplicate`.

Here are two examples:

```
GHCi> hpure Nothing :: NP Maybe   '[Char, Bool, Int]

GHCi> hpure (K 0)   :: NP (K Int) '[Char, Bool, Int]
```

### 1.6.3 Lifted functions

As a final step, let us tackle `hap`, the function corresponding to `vapply`:

```
vapply :: Vec (a -> b) n -> Vec a n -> Vec b n
vapply VNil           VNil           = VNil
vapply (f `VCons` fs) (x `VCons` xs) = f x `VCons` vapply fs xs
```

It is relatively obvious that we'll have to turn this into

```
hap :: NP...xs -> NP f xs -> NP g xs
```

But what goes into the place of `...`? We need the first vector to contain at the position indexed by `x` a function of type `f x -> g x`. So the interpretation function we're looking for is something like `\x -> (f x -> g x)`, only that we do not have type-level lambda available in Haskell.

The standard trick is to define a `newtype`, which will go along with some manual wrapping and unwrapping, but is isomorphic to the desired type:

```
newtype (f -.-> g) a = Fn {apFn :: f a -> g a}
infixr 1 -.->
```

We can then define

```
hap :: NP (f -.-> g) xs -> NP f xs -> NP g xs
hap Nil       Nil       = Nil
hap (f :* fs) (x :* xs) = apFn f x :* hap fs xs
```

Let's look at an example:

```
lists :: NP [] '[String, Int]
lists = ["foo", "bar", "baz"] :* [1 .. 10] :* Nil

numbers :: NP (K Int) '[String, Int]
numbers = K 2 :* K 5 :* Nil

fn_2 :: (f a -> f' a -> f'' a)
     -> (f -.-> f' -.-> f'') a
fn_2 f = Fn (\x -> Fn (\y -> f x y))

take' :: (K Int -.-> [] -.-> []) a
take' = fn_2 (\(K n) xs -> take n xs)

GHCi> hpure take' `hap` numbers `hap` lists
```

### 1.6.4 Another look at `hmap`

For normal lists (assuming the `ZipList` applicative functor instance), we have the identity

```
map f = pure f <*> xs
```

And indeed, the corresponding identity does also hold for environments:

```
hmap' :: SListI xs => (forall a . f a -> g a) -> NP f xs -> NP g xs
hmap' f xs = hpure (Fn f) `hap` xs
```

is the same as `hmap`. The only difference is an additional (and rather harmless) constraint on `SListI`.

## 1.7 Abstracting from classes and type functions

Can we also apply `hmap` or `hmap'` to `show`, thereby turning an `NP I` into an `NP (K String)`? Unfortunately not: for example, the call

```
hmap (K . show . unI) group'
```

results in a type error, because we have

```
K . show . unI ::  forall x . Show x => I x -> K String x
```

which does not match

```
                   forall x .           f x -> g        x
```

due to the class constraint on `Show`. But it would be useful to allow this, and in fact, there are other functions with other class constraints that we might also want to map over an environment, provided that all elements in the environment are actually instances of the class.

### 1.7.1 The kind `Constraint`

Fortunately, GHC these days also allows us to abstract over classes in this way. It turns out that classes can be seen as types with yet again a different kind. Where `data` introduces types of kind `*`, classes introduce types of kind `Constraint`. So like `*`, the kind `Constraint` is open.

Examples:

- Classes such as `Show` or `Eq` or `Ord` all have kind `* -> Constraint`. They are parameterized by a type of kind `*`, and if applied, form a class constraint.
- Classes such as `Functor` or `Monad` are of kind `(* -> *) -> Constraint`. They are parameterized by type constructors of kind `* -> *`.
- A multi-parameter type class such as `MonadReader` has kind `* -> (* -> *) -> Constraint`. It needs two parameters, one of kind `*` and one of kind `* -> *`.

In addition, we can use tuple syntax to create empty constraints and combine constraints in a syntactically somewhat ad-hoc fashion:

```
type NoConstraint        = (() :: Constraint)
type SomeConstraints a   = (Eq a, Show a)
type MoreConstraints f a = (Monad f, SomeConstraints a)
```

Note that there are no such things as "nested" constraint sets – collections of constraints are automatically flattened, `(, )` acts like a union operation here.

### 1.7.2 Type functions

One thing we would like to express is that a constraint holds for all elements of a type-level list. It is a part of what we need in order to write a variant of `hmap` that is compatible with constrained functions.

To do this, we are going to define a *type family* – essentially a function on types, combining a parameterized constraint and a list of parameters into a single constraint:

```
type family All (c :: k -> Constraint) (xs :: [k]) :: Constraint where
  All c '[]       = ()
  All c (x ': xs) = (c x, All c xs)
```

We can use a rather cryptic command in GHCi to expand type families for us and see `All` in action:

```
GHCi> :kind! All Eq '[Int, Bool]
All Eq '[Int, Bool] :: Constraint
 = (Eq Int, (Eq Bool, ()))
```

Once again, note that while GHCi shows this as a nested tuple, it is actually flattened into a single set of constraints.

We can use `All` to write a function that turns all elements in a heterogeneous list (a `HList`, not an `NP`) to strings via `show` and concatenates them:

```
hToString :: All Show xs => HList xs -> String
hToString HNil         = ""
hToString (HCons x xs) = show x ++ hToString xs
```

Let's see why this works: When pattern matching on `HCons`, GHC learns that `xs ~ y ': ys`. Therefore, we have by applying the definition of `All`:

```
All Show xs ~ All Show (y ': ys) ~ (Show y, All Show ys)
```

The call `show x` requires `Show y`, and the recursive call to `hToString xs` requires `Show ys`, so everything is fine.

### 1.7.3 Composing constraints

In order to do the same for `NP`, we need one extra ingredient. If we have e.g. an `NP f '[Int, Bool]`, we do not actually need

```
(Show Int, Show Bool)
```

but rather

```
(Show (f Int), Show (f Bool))
```

If we had function composition on the type level, we could also write

```
((Show `Compose` f) Int, (Show `Compose` f) Bool)
```

and express this as `All (Show `Compose` f) Bool`.

In general, function composition on the type level is difficult to define in Haskell – once again because of the lack of a type-level lambda. And `newtype`-wrapping isn't helpful here, because we need to produce a `Constraint`, and a `newtype` – just like `data` – always produces something of kind `*`.

However, it turns out that for the special case of composition where the result is a `Constraint`, we can use `class` to define it:

```
class (f (g x)) => (f `Compose` g) x
instance (f (g x)) => (f `Compose` g) x
```

Note that the inferred kind for `Compose` is

```
Compose :: (b -> Constraint) -> (a -> b) -> a -> Constraint
```

Constraints defined via `class` can – unlike type synonyms and type families – be partially applied, so we can actually apply `Compose` as desired in connection with `All`:

```
GHCi> :kind! All (Show `Compose` I) '[Int, Bool]
All (Show `Compose` I) '[Int, Bool] :: Constraint
 = (Compose Show I Int, (Compose Show I Bool, ()))
```

With this, we can define `hToString` for `NP`

```
hToString' :: All (Show `Compose` f) xs => NP f xs -> String
hToString' Nil       = ""
hToString' (x :* xs) = show x ++ hToString' xs
```

and try it:

```
GHCi> hToString' group'
```

This means we now also have the necessary ingredient to have GHC derive a proper `Show` instance for `NP` for us, because `show` is nothing but a more sophisticated version of our `hToString'` function with the same type requirements:

```
deriving instance (All (Compose Show f) xs) => Show (NP f xs)
```

### 1.7.4 Proxies

We'll now use abstraction over constraints to define a very useful variant of `hpure` that instead of taking an argument of type

```
forall a .        f a
```

takes an argument of type

```
forall a . c a => f a
```

for some parameterized constraint `c`. So we're aiming for `hcpure` with a type as follows (I'm putting the type of `hpure` next to it for comparison):

```
hpure  :: SListI xs            => (forall a .        f a) -> NP f xs
hcpure :: (SListI xs, All c xs) => (forall a . c a => f a) -> NP f xs
```

However, there is one more problem that we can observe without actually defining the function. Let's just say

```
hcpure :: (SListI xs, All c xs) => (forall a . c a => f a) -> NP f xs
hcpure = undefined
```

and try to call e.g.

```
GHCi> hcpure (I minBound) :: NP I '[Char, Bool]
```

Note that

```
I minBound :: Bounded a => I a
```

so it seems to have the correct type. Nevertheless, we get a type error from GHC complaining that it cannot properly match up `Bounded` with `c`. In fact, GHC generally refuses to infer contraint variables from matching available constraints, because there are situations where this can be ambiguous.

We can help GHC by providing a dummy parameter to the function – a so-called *proxy* – that has a trivial run-time representation and does nothing else but to fix the value of a type variable.

```
data Proxy (a :: k) = Proxy
```

A proxy works for arguments of any kind, so in particular for kind `* -> Constraint` at which we need it here. Because `data`-defined type are by construction injective, for GHC, knowing `Proxy a` is always sufficient to infer `a`.

### 1.7.5 A variant of `hpure` for constrained functions

So we'll actually define

```
hcpure :: forall c f xs . (SListI xs, All c xs)
       => Proxy c -> (forall a . c a => f a) -> NP f xs
```

```
hcpure p x = case sList :: SList xs of
  SNil  -> Nil
  SCons -> x :* hcpure p x
```

This works, as we can test:

```
GHCi> hcpure (Proxy :: Proxy Bounded) (I minBound) :: NP I '[Char, Bool]

GHCi> hcpure (Proxy :: Proxy Show) (Fn (K . show . unI)) `hap` group'
```

The second example shows that the composition of `hcpure` with the old `hap` also gives us the desired mapping of constrained functions over environments:

```
hcmap :: (SListI xs, All c xs)
      => Proxy c -> (forall a . c a => f a -> g a) -> NP f xs -> NP g xs
hcmap p f xs = hcpure p (Fn f) `hap` xs
```

# 2 Generic programming with $n$-ary sums of products

In the first part, we've taken a tour through Haskell's type-level programming extensions, driven mostly by the desire to cover all the concepts that we need in order to understand the generics-sop library.

It is no surprise that we've spent a lot of time on the type `NP` of $n$-ary products, because we use this type to represent the fields of a constructor.

In this chapter, we'll also introduce $n$-ary sums, which we'll use to represent the choice between constructors. We can then explain how most Haskell datatypes can easily be transformed into sums of products.

## 2.1 Recap: $n$-ary products `NP`

Let's repeat the definition of $n$-ary products:

```
data NP (f :: k -> *) (xs :: [k]) where
  Nil  :: NP f '[]
  (:*) :: f x -> NP f xs -> NP f (x ': xs)
infixr 5 :*
```

The most important functions that we've considered were:

```
hpure  :: SListI xs             => (forall a . f a) -> NP f xs
hcpure :: (SListI xs, All c xs) => Proxy c -> (forall a . c a => f a) -> NP f xs
hap    ::                          NP (f -.-> g) xs -> NP f xs -> NP g xs
```

These give us a generalization of the applicative functor interface, and allow us to define other functions such as `hmap` and `hcmap` easily:

```
hmap  :: (SListI xs)
      => (forall x . f x -> g x) -> NP f xs -> NP g xs
hcmap :: (SListI xs, All c xs)
      => Proxy c -> (forall a . c a => f a -> g a) -> NP f xs -> NP g xs
```

We had also considered

```
hcollapse :: NP (K a) xs -> [a]
```

that can collapse a homogeneous product into a normal list.

## 2.2 Generalizing choice

We moved from homogeneous lists to homogeneous length-indexed vectors, then to heterogeneous lists and ultimately to `NP`.

### 2.2.1 Choosing from a list

We'll try to go through a similar progression for choices or sums. A Haskell list is a sequence of arbitrarily many elements of type `a`. Let's also define a type that is a choice between arbitrarily many elements of type `a`:

```
data LChoice a = LCZero a | LCSuc (LChoice a)
```

If we imagine arbitrarily elements of type `a` being represented by a list of `a`s, then `LChoice a` gives us the index of a particular element in that list paired with the element being chosen. The constructors are named `LCZero` and `LCSuc` so that we can view an easily view values of type `LChoice a` as a zero-based index, where `LCZero` contains the payload.

Just to make the relation to the following datatypes more obvious, let's rewrite `LChoice` equivalently in GADT syntax:

```
data LChoice (a :: *) where
  LCZero :: a -> LChoice a
  LCSuc  :: LChoice a -> LChoice a
```

### 2.2.2 Choosing from a vector

If we now generalize from lists to vectors, we'll get a choice that is additionally indexed by the number of elements that we can choose from

```
data VChoice (a :: *) (n :: Nat) where
  VCZero :: a -> VChoice a (Suc n)
  VCSuc  :: VChoice a n -> VChoice a (Suc n)
```

We can choose the first element (via `VCZero`) from any non-empty vector – so `VCZero` forces the length index to be of form `Suc n`.

If we have an index into an `n`-element vector, we can also produce an index into an `Suc n`-element vector by skipping the first element. This is what `VCSuc` does. Once again, we pair the choice with the actual element of type `a` being chosen.

Note that there are no possible choices of an `a` from an empty vector, which is reflected by the fact that there are no constructors of `VChoice` targeting the index `Zero`.

### 2.2.3 Choosing from a heterogeneous list

The next step is to generalize from vectors to `HList`, so we'll have an index that is a list of types:

```
data HChoice (xs :: [*]) where
  HCZero :: x -> HChoice (x ': xs)
  HCSuc  :: HChoice xs -> HChoice (x ': xs)
```

Now the element chosen, which is still stored in `HCZero`, is taken from the index list. In analogy with `VChoice`, we have no constructors targeting the empty index list.

### 2.2.4 Choosing from an environment

For $n$-ary products, we decided that it's additionally helpful to build type constructor application into the type. We'll perform the same generalization for our choice, and arrive at our type of $n$-ary sums:

```
data NS (f :: k -> *) (xs :: [k]) where
  Z :: f x -> NS f (x ': xs)
  S :: NS f xs -> NS f (x ': xs)
```

The payload is now of type `f x`, where `x` is from the index list. Just as for `NP`, we can actually be kind-polymorphic in the index list, because only elements of type `f x` directly appear as data in the structure.

Once again, if we choose `f = I`, the type is equivalent to `HChoice`, and if we choose `f = K a`, we get something that is much like `VChoice`, where the elements in the index list are ignored, and just the length of it determines the number of `a`s we can choose from.

Let's look at an example. Let's consider the type

```
type ExampleChoice = NS I '[Char, Bool, Int]
```

So we can choose a character with index `0`, a Boolean with index `1`, or an integer with index `2`:

```
c0, c1, c2 :: ExampleChoice
c0 = Z (I 'x')
c1 = S (Z (I True))
c2 = S (S (Z (I 3)))
```

It's useful for experimenting if we have a `Show` instance for `NS` available. We can let GHC derive one in much the same way as we have for `NP` – we require that all elements that can appear in the `NS` have show instances themselves:

```
deriving instance All (Show `Compose` f) xs => Show (NS f xs)
```

## 2.3 Sums of products

Having defined both `NS` and `NP`, we are now at a point where we can define the uniform generic representation that we are going to use for datatypes.

### 2.3.1 Representing expressions

Let's recap from the very beginning. If we have a datatype such as e.g.

```
data Expr = NumL Int | BoolL Bool | Add Expr Expr | If Expr Expr Expr
```

then each value of that type is of form

$$C_i\ x_0 \ldots x_{n_i - 1}$$

where $C_i$ is one of the constructors, in this case either NumL, BoolL, Add or If. And the xs are the arguments of the chosen constructor.

We use NS to represent the choice between the constructors, and we have four things to choose from. We use NP to represent the sequence of arguments. The products contain actual elements of the argument types. So the representation type corresponding to Expr is

```
type RepExpr = NS (NP I) (' [ '[Int]
                           , '[Bool]
                           , '[Expr, Expr]
                           , '[Expr, Expr, Expr]
                           ])
```

The index is now a list of list of types – a table that, except for the names of the datatype and the constructors, completely specifies the structure of the datatype. The outer list reflects the sum structure. Each list element corresponds to one constructor. The inner lists reflect the product structures, and each element corresponds to one argument.

Note that we only represent the top-level structure of an expression. A RepExpr contains values of the original type Expr. We never look at the arguments of a constructor further. Conversion between a datatype and its structural representation will always be shallow and non-recursive. If needed, we can and will apply such conversions multiple times while traversing a recursive strucure.

Let's consider an example value of type Expr:

```
exampleExpr :: Expr
exampleExpr = If (BoolL True) (NumL 1) (NumL 0)
```

The corresponding RepExpr is:

```
exampleRepExpr :: RepExpr
exampleRepExpr = S (S (Z (I (BoolL True) :* I (NumL 1) :* I (NumL 0) :* Nil))))
```

In the following, we're sometimes going to use a more concise syntax in order to represent values of types NS and NP:

```
exampleRepExpr :: RepExpr
exampleRepExpr = C3 [I (BoolL True), I (NumL 1), I (NumL 0)]
```

I.e., we'll collapse the sequence of constructor applications S (S (Z...)) into $C_2$ and use ordinary list syntax for the $n$-ary product.

It is easy to see that RepExpr is isomorphic to Expr:

```
fromExpr :: Expr -> RepExpr
fromExpr (NumL n)      = Z          (I n :* Nil)
fromExpr (BoolL b)     = S (Z       (I b :* Nil))
fromExpr (Add e1 e2)   = S (S (Z    (I e1 :* I e2 :* Nil)))
fromExpr (If e1 e2 e3) = S (S (S (Z (I e1 :* I e2 :* I e3 :* Nil))))

toExpr :: RepExpr -> Expr
toExpr (Z          (I n :* Nil))                    = NumL n
toExpr (S (Z       (I b :* Nil)))                   = BoolL b
toExpr (S (S (Z    (I e1 :* I e2 :* Nil))))         = Add e1 e2
toExpr (S (S (S (Z (I e1 :* I e2 :* I e3 :* Nil))))) = If e1 e2 e3
```

### 2.3.2 The `Generic` class

In the beginning, we presented the following class `Generic` to capture the idea of datatype-generic programming:

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

We're now at a point where we can define the actual class `Generic` that is used in generics-sop. Because all representations will be of the form

```
NS (NP I) c
```

for some code `c :: [[*]]`, we decide to define a `newtype` and a type synonym:

```
newtype SOP f a = SOP {unSOP :: NS (NP f) a}
type Rep a = SOP I (Code a)
```

and include `Code` in the class rather than `Rep`. Additionally, we're going to demand that both the outer list and all inner lists of the code have implicit singletons available. This isn't strictly speaking necessary, but it's harmless and will reduce the number of constraints we need to put on function definitions later:

```
class (SListI (Code a), All SListI (Code a)) => Generic (a :: *) where
  type Code a :: [[*]]
  from :: a -> Rep a
  to   :: Rep a -> a
```

The instantiation of `Generic` to e.g. `Expr` is straight-forward and mechanical:

```
instance Generic Expr where
  type Code Expr = ' [ '[Int]
                     , '[Bool]
                     , '[Expr, Expr]
                     , '[Expr, Expr, Expr]
                     ]
  from x = SOP (fromExpr x)
  to (SOP x) = toExpr x
```

### 2.3.3 Other `Generic` instances

The presence of datatype parameters does not make the definition of a `Generic` instance harder, and we can easily provide `Generic` instances even for built-in types such as Haskell lists:

```
instance Generic [a] where
  type Code [a] = '['[], '[a, [a]]]
```

```
from []       = SOP (Z    Nil)
from (x : xs) = SOP (S (Z (I x :* I xs :* Nil)))
to (SOP (Z    Nil))                  = []
to (SOP (S (Z (I x :* I xs :* Nil)))) = x : xs
```

The `Generic` instances are the boilerplate we still have to write in order to be able to benefit from the generics-sop approach. We'll see that once we have a `Generic` instance, we get all generic functions we write for free for all these types.

But the `Generic` instances themselves are easy to automate. In practice, we have a number of options:

- The first and generally worst option is to write `Generic` instances by hand. This is quickly getting tedious, but in principle gives us more flexibility, because we could decide to define non-standard `Generic` instances.
- The second is to use Template Haskell to derive `Generic` instances for us. This is easy to do, and provided by the generics-sop library. We can just put

  ```
  deriveGeneric ''Expr
  ```

  after the `data` declaration of `Expr` and get the `Generic` instance derived for us automatically.
- The third option is to extend the compiler with support to `derive` instances of `Generic`, just like it can derive instances of `Eq` and `Show`. Then we could simply write

  ```
  data Expr = ...
    deriving (..., Generic)
  ```

  and would also get the instance for us automatically. This option deserves a bit more discussion.

#### 2.3.4 Support for `Generic` in the compiler

As discussed in the beginning, there are many – somewhat competing – approaches to generic programming in Haskell, which use slightly different views on the datatypes. While it is conceivable that GHC supports one or even two of these directly, it's not feasible to support all of them directly. Each new class that has built-in `deriving` requires a compiler change and subsequent maintenance of the feature.

In fact, GHC has first-class support for two generic programming approaches already. For SYB, it allows `deriving Data`, and for generic-deriving, it allows `deriving Generic` – but for a different class also named `Generic`.

Now, an interesting realization is that with all the type-level programming features that GHC offers and we have discussed, we actually have sufficient machinery to turn the representations GHC has built-in support for into other generic representations programmatically! This makes extensive use of type classes and typ families that turn one structural representation into another, and at the same time provide functions that convert values from one representation into values in another (and back).

It is e.g. possible to convert from GHC's own `Generic` class into the generics-sop-provided `Generic` class using this technique. This is also implemented in the generics-sop library. To use this approach, we have to first import GHC's `Generic` class in qualified form:

```
import qualified GHC.Generics as GHC
```

Then, when defining a datatype such as `Expr`, we tell GHC to derive its own `Generic` class for this type:

```
data Expr = ...
  deriving (..., GHC.Generic)
```

Finally, we can just provide an empty instance for generics-sop's `Generic` class:

```
instance Generic Expr
```

The implementation of generics-sop has default definitions for `Generic` that will use and translate from `GHC.Generic`.

This technique of translating different generic programming representations into each other that we have dubbed "Generic Generic Programming" has a significant impact: GHC should no longer strive to provide built-in support for the "one true" or "best" approach to generic programming, nor do we need a "one size fits all" approach to generic programming. GHC instead should provide support for a representation that captures as much information as possible, even if that means that the representation is difficult to use. But it means we can translate easily from that representation to various others, forgetting unnecessary things along the way. Generic programming libraries also no longer have to compete for the one or two spots of first-class support, but can all co-exist with each other and be used together by users in one and the same program.

## 2.4 Defining generic equality

Let us now look at defining one standard example of datatype-generic programming in our approach – generic equality.

### 2.4.1 A bottom-up approach

Let's start bottom-up, by comparing two $n$-ary products:

```
geqNP :: All Eq xs => NP I xs -> NP I xs -> Bool
geqNP Nil          Nil          = True
geqNP (I x1 :* xs1) (I x2 :* xs2) = x1 == x2 && geqNP xs1 xs2
```

This is rather straight-forward. We traverse both producs in parallel, and compare all the contained elements for equality pointwise. We fall back to the `Eq` class for the elements, which means we have to require an `Eq` instance for all types that occur in the signature, using `All`.

(In fact, we can do this in a slightly more high-level way, and will return to this point in a moment.)

Next, lets' try to compare to $n$-ary sums of $n$-ary products:

```
geqNS :: ... => NS (NP I) xss -> NS (NP I) xss -> Bool
geqNS (Z np1) (Z np2) = geqNP np1 np2
geqNS (S ns1) (S ns2) = geqNS ns1 ns2
geqNS _       _       = False
```

The idea is also clear. We traverse the two sums and check whether both are mapping to the same choice. If it's obvious that they don't, we return `False`. Otherwise, we call `geqNP` on the products contained within.

### 2.4.2 The `All2` constraint

The problem here is that due to `geqNP`, we need a constraint that all elements of all the *inner* lists in the list of lists `xss` are instances of `Eq`. If we had partial parameterization of type families available, this would be easy to do:

```
All (All Eq) xss
```

However, GHC does not admit that. Instead, we simply define a separate type family `All2` that works on lists of lists. This is not nice, but neither is it tragic, because due to the two-layer structure of our datatypes `All` and `All2` are the only cases we'll ever need.

```
type family All2 (f :: k -> Constraint) (xss :: [[k]]) :: Constraint where
  All2 f '[]         = ()
  All2 f (xs ': xss) = (All f xs, All2 f xss)
```

With `All2`, we can now complete `geqNS`:

```
geqNS :: All2 Eq xss => NS (NP I) xss -> NS (NP I) xss -> Bool
geqNS (Z np1) (Z np2) = geqNP np1 np2
geqNS (S ns1) (S ns2) = geqNS ns1 ns2
geqNS _       _       = False
```

### 2.4.3 Completing the definition

From here, we can go to `geqSOP` simply by stripping the `SOP` constructors:

```
geqSOP :: All2 Eq xss => SOP I xss -> SOP I xss -> Bool
geqSOP (SOP sop1) (SOP sop2) = geqNS sop1 sop2
```

And now, it is easy to define a generic equality that works for all representable types:

```
geq :: (Generic a, All2 Eq (Code a)) => a -> a -> Bool
geq x y = geqSOP (from x) (from y)
```

If we now wanted to use `geq` to define equality for a datatype, such as `Expr`, we could do so instead of invoking `deriving Eq` by giving a trivial instance declaration:

```
instance Eq Expr where
  (==) = geq
```

Let's try this:

```
GHCi> geq (Add (NumL 3) (NumL 5)) (Add (NumL 3) (NumL 5))

GHCi> geq (Add (NumL 3) (NumL 5)) (Add (NumL 3) (NumL 6))
```

### 2.4.4 Improving the product case

Not without reason, we spent a lot of time defining useful functions over `NP`. One of the advantages of the generics-sop approach is that it encourages defining high-level operations that work on `NP` and `NS`, and then using a composition of such operators to define (parts of) generic functions.

Let's look again at `geqNP`:

```
geqNP :: All Eq xs => NP I xs -> NP I xs -> Bool
geqNP Nil          Nil          = True
geqNP (I x1 :* xs1) (I x2 :* xs2) = x1 == x2 && geqNP xs1 xs2
```

We can see this as a composition of several different operations: first, we zip the two products together using `(==)`, getting a homogeneous product of `Bool`s. We can thus collapse that product into a `[Bool]` and then just fold with `(&&)` and `True` over that list.

The zipping we need here turns out to be a straight-forward generalization of `hcmap`, again easy to define using `hcpure` and `hap`:

```
hczipWith :: (All c xs, SListI xs)
          => Proxy c
          -> (forall a . (c a) => f a -> g a -> h a)
          -> NP f xs -> NP g xs -> NP h xs
hczipWith p f xs1 xs2 = hcpure p (fn_2 f) `hap` xs1 `hap` xs2
```

We can now redefine `geqNP` as follows:

```
geqNP' :: (SListI xs, All Eq xs) => NP I xs -> NP I xs -> Bool
geqNP' xs ys = and
             $ hcollapse
             $ hczipWith (Proxy :: Proxy Eq) aux xs ys
  where
    aux (I x) (I y) = K (x == y)
```

The only tiny annoyance is that we have to unwrap and rewrap the arguments of `(==)` from with `I` and `K` constructors as appropriate. We also pay the price of an additional `SListI xs` constraint that we then also have to add to the other functions that are used to define `geq` – but ultimately, it does not change anything, because we had already imposed `SListI` constraints as superclass constraints on `Generic`.

## 2.5 Generic producers

### 2.5.1 Generically producing default values

We'll try to define a generic function that produces a value of a given datatype. This functionality is for example offered by the data-default package that is available on Hackage. It defines a class

```
class Default a where
  def :: a
```

We will try to define a generic version of `def` that we can use to define trivial instances for types that are instances of `Generic`:

```
gdef :: (Generic a, Code a ~ (xs ': xss), All Default xs) => a
gdef = to (SOP gdefNS)
```

It may be surprising that we're not following the pattern we discovered for `geq` and have the type

```
gdef :: (Generic a, All2 Default (Code a)) => a
```

But let's think for a moment what `gdef` should actually do? We have to choose a constructor and then choose default values for the arguments of that constructor. There are different strategies how we could choose a constructor, but one obvious strategy is to just take the first.

Now, we can only choose the first constructor if there is at least one! The constraint `Code a ~ (xs ': xss)` makes this requirement precise. In turn, we then do not really need that *all* arguments of all constructors have a `Default` instance. We only require this for the arguments of the first constructor `xs`, and therefore `All Default xs` is sufficient.

Completing the definition is straight-forward:

```
gdefNS :: forall xs xss . (SListI xs, All Default xs) => NS (NP I) (xs ': xss)
gdefNS = case sList :: SList xs of
  SCons -> Z (hcpure (Proxy :: Proxy Default) (I def))
```

With `Z`, we choose the first constructor, and using `hcpure`, we'll create a sequence of `def`s:

```
C₀ [def, ...]
```

If we want `gdef` for example on `Expr`, we need to have a `Default` instance on `Int`, because the first constructor is an `Int`. Let's define this manually:

```
instance Default Int where
  def = 0
```

Then:

```
GHCi> gdef :: Expr
```

### 2.5.2 Using default signatures

GHC also allows us to provide a generic default definition for a class such as `Default`:

```
class Default a where
  def :: a
  default def :: (Generic a, Code a ~ (xs ': xss), All Default xs) => a
  def = gdef
```

Then we can simply give an empty instance declaration to make use of the default:

```
instance Default Expr
```

In future versions of GHC, it will become possible to use `deriving Default` on `Expr` in a situation like this.

### 2.5.3 Generating all possible constructors

Let's consider a variant of `gdef` that produces a list of default values, one for each constructor:

```
gdefAll :: (Generic a, All2 Default (Code a)) => [a]
gdefAll = map (to . SOP) gdefAllNS
```

We don't change the `Default` class for this example, where `def` will still return a single default value. But now we do in fact require that all arguments to all constructors are an instance of `Default`.

```
gdefAllNS ::  forall xss . (SListI xss, All SListI xss, All2 Default xss)
          => [NS (NP I) xss]
gdefAllNS = case sList :: SList xss of
  SNil  -> []
  SCons -> Z (hcpure (Proxy :: Proxy Default) (I def)) : map S gdefAllNS
```

The function `gdefAllNS` is a rather straight-forward definition of the function we want: we use the list singleton to figure out whether we need to build an empty or non-empty list. If the list is non-empty, then the first element will be constructed by `Z`, the others will be constructed using `S`. For the product structure, we use the same call to `hcpure` that we've already used in `gdefNS`.

We can test this function on `Expr` again:

```
instance Default Bool where
  def = False


GHCi> gdefAll :: [Expr]
```

Note that the final value produces is not actually type-correct. But obviously, our datatype allows us to represent type-incorrect values in the expression language, so we get no corresponding guarantees.

## 2.6 Products of products and injections

If we had a product of products, rather than a sum of products, based on the code of a datatype, then we could call `def` for every position, and would get a table of `def` calls. Using ordinary list syntactic sugar, for `Expr` we'd get something like this:

```
[[def], [def], [def, def], [def, def, def]]
```

Then if we would have such a table, we could combine it with all possible choices and obtain:

```
[C₀ [def], C₁ [def], C₂ [def, def], C₃ [def, def, def]]
```

This would essentially correspond to the same list of values that we're generating in `gdefAllNS`, split into several reusable steps.

### 2.6.1 Products of products

We can define a product of products much like `SOP`:

```
newtype POP f a = POP {unPOP :: NP (NP f) a}
```

Creating the table is morally just a nested call to `hcpure`. But for similar reasons that we had to define `All2`, we cannot just reuse `hcpure` twice, but have to mirror the construction on the outer level using `All2` in the place of `All`:

```
hcpure_POP ::  forall c f xss . (SListI xss, All SListI xss, All2 c xss)
           => Proxy c -> (forall a . c a => f a) -> POP f xss
hcpure_POP p x = POP (case sList :: SList xss of
  SNil  -> Nil
  SCons -> hcpure p x :* unPOP (hcpure_POP p x))
```

The call

```
hcpure_POP (Proxy :: Proxy Default) (I def) :: POP I (Code Expr)
```

now creates the above-mentioned table of shape

```
[[I def], [I def], [I def, I def], [I def, I def, I def]]
```

Taking the instances of `Default` into account, it will actually generate:

```
[[I 0], [I False], [I (NumL 0), I (NumL 0)], [I (NumL 0), I (NumL 0), I (NumL 0)]]
```

(Although the value GHCi would actually print is far more ugly, due to GHC not using the compact list notation.

### 2.6.2 Injections

Consider the `Either` type, which represents a binary choice:

```
data Either a b = Left a | Right b
```

where

```
Left  :: a -> Either a b
Right :: b -> Either a b
```

are the two *injections* into the `Either` type. They're of different, but related types. In fact, we can put them into a single environment with a little bit of tweaking:

```
eitherInjections :: NP (I -.-> K (Either a b)) '[a, b]
eitherInjections = Fn (K . Left  . unI)
                :* Fn (K . Right . unI)
                :* Nil
```

Let's try to do something similar with the injections into an $n$-ary sum:

```
injections :: forall xs f . SListI xs => NP (f -.-> K (NS f xs)) xs
injections = case sList :: SList xs of
  SNil  -> Nil
  SCons -> Fn (K . Z) :* hmap (Fn . ((K . S . unK) .) . apFn) injections
```

Looking at `injections` in the generic setting and keeping in mind that we are representing datatypes as sums of products, we can think of `f` being instantiated to `NP I`. Then `injections` computes (modulo isomorphism) an environment containing all the constructor functions of a datatype, looking roughly as follows:

```
[Fn (K . C₀), Fn (K . C₁), ..., Fn (K . Cₙ₋₁)]
```

If we have suitable arguments for each of the constructors, we can apply all the constructor functions to the arguments. Since all constructors inject into the same type, we obtain a homogeneous list that we can collapse:

```
apInjs :: SListI xs => NP f xs -> [NS f xs]
apInjs np = hcollapse (injections `hap` np)
```

Once again, `f` here will usually be instantiated to an `NP`, so that we define the following variant of `apInjs` that takes a product of product (a table of constructor arguments for all of the constructors of a datatype) to a list of `SOP`s (a list of constructed values, containing one entry per constructor):

```
apInjs_POP :: SListI xs => POP f xs -> [SOP f xs]
apInjs_POP (POP pop) = map SOP (apInjs pop)
```

### 2.6.3 Redefining `gdefAll`

It is now possible to redefine `gdefAll` as follows:

```
gdefAll' :: (Generic a, All2 Default (Code a)) => [a]
gdefAll' = map to (apInjs_POP (hcpure_POP (Proxy :: Proxy Default) (I def)))
```

As indicated above, we're creating a two-dimensional table of `def` calls, suitable as arguments for each of the constructors. We're then using `apInjs_POP` to actually apply the constructors, and finally convert back from the generic representation using `to`.

The result is exactly the same as for `gdefAll`:

```
GHCi> gdefAll' :: [Expr]
```

## 2.7 Abstracting common patterns

Now that we're no longer just considering `NP`, but also `NS`, `SOP` and `POP`, we see that these four types support many similar functions.

In order to not have lots of very similarly named functions, we'll therefore explore how we can define suitable type classes.

### 2.7.1 A class for `hpure` and `hcpure`

We've just defined `hcpure_POP` which is very similar to `hcpure_NP` on `NP` (we've called the latter just `hcpure` before, but I'm now going to refer to it as `hcpure_NP` to emphasize, and to free up the name `hcpure` for an overloaded version).

```
hcpure_POP :: (SListI xss, All SListI xss, All2 c xss)
           => Proxy c -> (forall a . c a => f a) -> POP f xss
hcpure_NP  :: (SListI xs, All c xs)
           => Proxy c -> (forall a . c a => f a) -> NP  f xs
```

We also have comparable non-constrained versions for both `NP` and `POP`:

```
hpure_POP :: (SListI xss, All SListI xss) => (forall a . f a) -> POP f xss
hpure_NP  :: (SListI xs)                  => (forall a . f a) -> NP  f xs
```

These have different constraints, but are nevertheless easy to abstract into a class:

```
class HPure (h :: (k -> *) -> (l -> *)) where
  hcpure :: (SListIMap h x, AllMap h c x)
         => Proxy c -> (forall a . c a => f a) -> h f x
  hpure  :: (SListIMap h x)
         =>            (forall a . f a) -> h f x
```

We define two type families `SListIMap` and `AllMap` – both also parameterized by the type `h` in question – that capture the slightly different constraints we need:

```
type family SListIMap (h :: (k -> *) -> (l -> *))
                      (xs :: l)
                       :: Constraint
type instance SListIMap NP  xs  = SListI  xs
type instance SListIMap POP xss = SListI2 xss
```

where

```
class (SListI xss, All SListI xss) => SListI2 xss
instance (SListI xss, All SListI xss) => SListI2 xss
```

And

```
type family AllMap (h :: (k -> *) -> (l -> *))
                   (c :: k -> Constraint)
                   (xs :: l)
                    :: Constraint
type instance AllMap NP  c xs  = All  c xs
type instance AllMap POP c xss = All2 c xss
```

With all these definitions in place, it is then trivial to define the instances for `NP` and `POP`:

```
instance HPure NP  where
  hcpure = hcpure_NP
  hpure  = hpure_NP
instance HPure POP where
  hcpure = hcpure_POP
  hpure  = hpure_POP
```

Actually, for `POP`, we are still lacking the definition of `hpure_POP`:

```
hpure_POP ::  forall xss f . (SListI xss, All SListI xss)
          => (forall a . f a) -> POP f xss
hpure_POP x = POP (hcpure (Proxy :: Proxy SListI) (hpure x))
```

Once again, it is somewhat unfortunate that the definition is not completely symmetric, and we use the `hcpure` of an `hpure` – but at least we can use a previously defined function in this case.

### 2.7.2 A class for `hap`

We've also seen `hap_NP` (previously called `hap`):

```
hap_NP :: NP (f -.-> g) xs -> NP f xs -> NP g xs
```

We can easily define a version of this operator that operates on `POP`s:

```
hap_POP :: SListI xss => POP (f -.-> g) xss -> POP f xss -> POP g xss
hap_POP (POP pop1) (POP pop2) = POP (hpure (fn_2 hap) `hap` pop1 `hap` pop2)
```

Interestingly, we can also combine an `NP` with an `NS` into another `NS`. We are applying one of the functions contained in the `NP` with the argument contained in the `NS`:

```
hap_NS :: NP (f -.-> g) xs -> NS f xs -> NS g xs
hap_NS (f :* _) (Z x) = Z (apFn f x)
hap_NS (_ :* fs) (S ns) = S (hap_NS fs ns)
```

Similarly, we can combine a `POP` with a `SOP`:

```
hap_SOP :: SListI xss => POP (f -.-> g) xss -> SOP f xss -> SOP g xss
hap_SOP (POP pop) (SOP sop) =
  SOP (hpure (fn_2 hap) `hap_NP` pop `hap_NS` sop)
```

Let's collect the type signatures we have:

```
hap_NP  ::               NP  (f -.-> g) xs  -> NP  f xs  -> NP  g xs
hap_NS  ::               NP  (f -.-> g) xs  -> NS  f xs  -> NS  g xs
hap_POP :: SListI xss => POP (f -.-> g) xss -> POP f xss -> POP g xss
hap_SOP :: SListI xss => POP (f -.-> g) xss -> SOP f xss -> SOP g xss
```

In order to collect these four into one class, we define a type family that maps sums to products and products into itself:

```
type family Prod (h :: (k -> *) -> (l -> *)) :: (k -> *) -> (l -> *)
type instance Prod NP  = NP
type instance Prod NS  = NP
type instance Prod POP = POP
type instance Prod SOP = POP
```

In addition, we are going to be overly conservative with the singletons we require and reuse `SListIMap` even though the definitions given above have fewer constraints. (In fact, we could completely get rid of them for all four `hap` versions, if we directly defined them via pattern matching on the arguments.)

```
class (HPure (Prod h), Prod (Prod h) ~ Prod h)
       => HAp (h :: (k -> *) -> (l -> *)) where
  hap :: SListIMap (Prod h) xs => Prod h (f -.-> g) xs -> h f xs -> h g xs
```

The superclass constraints here are mainly in order to save a number of constraints on other functions. With the `HAp` class in place, we have the following four instances:

```
instance HAp NP  where hap = hap_NP
instance HAp NS  where hap = hap_NS
instance HAp POP where hap = hap_POP
instance HAp SOP where hap = hap_SOP
```

Given these, we can define a general `hmap` and `hzipWith` (and similarly `hcmap` and `hczipWith`):

```
hmap :: (SListIMap (Prod h) xs, HAp h)
     => (forall a . f a -> g a) -> h f xs -> h g xs
```

```
hmap f xs = hpure (Fn f) `hap` xs
hzipWith :: (SListIMap (Prod h) xs, HAp (Prod h), HAp h)
         => (forall a . e a -> f a -> g a)
         -> Prod h e xs -> h f xs -> h g xs
hzipWith f xs ys = hpure (fn_2 f) `hap` xs `hap` ys
```

### 2.7.3 A class for `hcollapse`

Homogeneous structures of all four types can be collapsed as follows:

```
class HCollapse (h :: (k -> *) -> (l -> *)) where
  hcollapse :: SListIMap h xs => h (K a) xs -> CollapseTo h a
```

where

```
type family CollapseTo (h :: (k -> *) -> (l -> *)) (a :: *) :: *
type instance CollapseTo NP  a = [a]
type instance CollapseTo NS  a = a
type instance CollapseTo POP a = [[a]]
type instance CollapseTo SOP a = [a]
```

If applied to sums, the sum structure is just discarded. The definitions are straight-forward.

# 3 Applications

In the following, we will have a look at a few additional applications of our approach. We will also consider in more detail how the generics-sop approach is quite helpful when it comes to defining variants of existing generic functions that can be configured via user-defined metadata. We'll also look at how actual datatype metadata (such as names) can be used in generic functions.

## 3.1 Producing test data

We've explained how to produce a default value generically. A situation where we have to perform a rather similar yet tedious task is for QuickCheck: In order to use QuickCheck on functions with inputs of a type `a`, we need to know how to produce test data of type `a`, which is captured by the `Arbitrary` class:

```
class Arbitrary a where
  arbitrary :: Gen a
  shrink    :: a -> [a]
```

The type `Gen` is an abstract datatype for *generators* of type `a`. QuickCheck comes with a little library of basic generators, and provides us with a `Monad` (and hence `Applicative`) interface for `Gen`.

## 3.2 Defining `arbitrary` generically

Let us first consider `arbitrary`. We can start in exactly the same way as we did for `gdefAll'`: by building a table of recursive `arbitrary` invocations for all possible constructor arguments, and injecting them:

```
apInjs_POP (hcpure (Proxy :: Proxy Arbitrary) arbitrary)
   :: (SListI xss, All SListI xss, All2 Arbitrary xss) => [SOP Gen xss]
```

How do we turn an `SOP` of generators into a generator for `SOP`s?

## 3.3 Sequencing

It turns out that all four of our types support a sequencing operation:

```
hsequence_NP  :: (Applicative f)
              => NP  f xs  -> f (NP  I xs)
hsequence_NS  :: (Applicative f)
              => NS  f xs  -> f (NS  I xs)
```

```
hsequence_POP :: (SListI xss, All SListI xss, Applicative f)
              => POP f xss -> f (POP I xss)
hsequence_SOP :: (SListI xss, All SListI xss, Applicative f)
              => SOP f xss -> f (SOP I xss)
```

In fact, this is just a special case of a more general sequencing operation:

```
hsequence_NP'  :: (Applicative f)
               => NP  (f :.: g) xs  -> f (NP  g xs)
hsequence_NS'  :: (Applicative f)
               => NS  (f :.: g) xs  -> f (NS  g xs)
hsequence_POP' :: (SListI xss, All SListI xss, Applicative f)
               => POP (f :.: g) xss -> f (POP g xss)
hsequence_SOP' :: (SListI xss, All SListI xss, Applicative f)
               => SOP (f :.: g) xss -> f (SOP g xss)
```

where `:.:` is composition of type constructors via a `newtype`:

```
newtype (f :.: g) x = Comp {unComp :: f (g x)}
```

We're going to show the definition just for the `NP` case:

```
hsequence_NP' Nil           = pure Nil
hsequence_NP' (Comp x :* xs) = (:*) <$> x <*> hsequence_NP' xs
```

We can once again capture all of these in a class:

```
class HAp h => HSequence (h :: (k -> *) -> (l -> *)) where
  hsequence' :: (SListIMap h xs, Applicative f) => h (f :.: g) xs -> f (h g xs)
instance HSequence NP where
  hsequence' = hsequence_NP'
```

The definition of `hsequence` can be given in terms of `hsequence'` for all members of the class:

```
hsequence :: (HSequence h, SListIMap h xs, SListIMap (Prod h) xs, Applicative f)
          => h f xs -> f (h I xs)
hsequence = hsequence' . hmap (Comp . fmap I)
```

## 3.4 Completing `arbitrary`

Before, we had

```
apInjs_POP (hcpure (Proxy :: Proxy Arbitrary) arbitrary)
 :: (SListI xss, All SListI xss, All2 Arbitrary xss) => [SOP Gen xss]
```

We're now going to apply `hsequence_SOP` to the elements of the list, and can even convert back to the original types:

```
map (fmap to . hsequence) (apInjs_POP (hcpure (Proxy :: Proxy Arbitrary) arbitrary))
 :: (Generic a, All2 Arbitrary (Code a)) => [Gen a]
```

At this point, we can use the QuickCheck function `oneof`

```
oneof :: [Gen a] -> Gen a
```

that will just randomly choose one of the given generators. All that then remains to complete the definition is to map `to` over `Gen` in order to translate back from the structural representation to the actual datatype.

```
garbitrary :: (Generic a, All2 Arbitrary (Code a)) => Gen a
garbitrary = oneof
           $ map (fmap to . hsequence)
           $ apInjs_POP
           $ hcpure (Proxy :: Proxy Arbitrary)
           $ arbitrary
```

QuickCheck provides us with a function

```
sample :: Show a => Gen a -> IO ()
```

that we can use to quickly test a generator. It will run the generator to produce a few random values and print these. However, if we try to say

```
instance Arbitrary Expr where
  arbitrary = garbitrary
```

and then

```
GHCi> sample (arbitrary :: Gen Expr)
```

we may in theory get lucky, but it's far more likely that we'll see GHCi starting to print very long – possibly infinite – expressions back to us. Why is this happening? Each constructor is chosen with roughly equal probability. Two of the four constructors terminate immediately, but `Add` has two subexpressions, and `If` three. So the chance that we terminate after generating one constructor is one half. Otherwise we'll have two or three subtrees. The chance that we'll terminate on the next level is smaller, because it will only happen if we choose a literal for each of the subtrees! So we're quite likely to end up with more and more subtrees, and will never finish.

There are different solutions to this problem.

### 3.4.1 Changing the probabilities of the constructors

As a first attempt, let's configure the `garbitrary` function with weights, so that we can then use QuickCheck's `frequency` function:

```
frequency :: [(Int, Gen a)] -> Gen a
```

The function `frequency` is a generalization of `oneof` where every generator can be weighted. If all weights are equal, the function behaves like `oneof`.

Rather than trying to compute the weights automatically, we are for now going to try to accepts them as inputs. Often, we like to configure generic functions using data that is itself

dependent on the shape of the datatype we are operating on. In our case, we want to have as many weights as we have constructors. Fortunately, given our framework, this is easy: we can just use a

```
NP (K Int) (Code a)
```

This will have as many `Int`s as the outer list of the `Code` has elements, which is exactly the number of constructors.

```
garbitraryWithFreqs :: (Generic a, All2 Arbitrary (Code a))
                    => NP (K Int) (Code a) -> Gen a
garbitraryWithFreqs freqs =
     frequency
   $ hcollapse
   $ hzipWith (\(K x) (K y) -> K (x, fmap to (hsequence (SOP y))))
       freqs
       (injections `hap` unPOP (hcpure (Proxy :: Proxy Arbitrary) arbitrary))
```

Here, we decide not to use `apInjs_POP`, because it collapses the list too early. Instead, we manually apply `injections` to the products of recursive `arbitrary` calls. We then zip this up with the input frequencies, and finally collapse it and feed it to the `frequency` function.

We can now make `Expr` an instance of `Arbitrary`, but decide the frequencies we want:

```
instance Arbitrary Expr where
  arbitrary = garbitraryWithFreqs (K 5 :* K 5 :* K 2 :* K 1 :* Nil)
```

By making the choice for the leaf constructors much more likely than that for the branching constructors, we are effectively solving the problem of generating infinite values:

```
GHCi> sample (arbitrary :: Gen Expr)
BoolL True
NumL (- 2)
NumL (- 1)
BoolL True
If (BoolL False) (BoolL False) (BoolL True)
BoolL False
NumL 12
Add (BoolL True) (BoolL False)
NumL (- 3)
BoolL True
BoolL True
```

### 3.4.2 Computing arities of constructors

Now, as a next step, let's write a generic "function" that computes the arities of the constructors of a type. This information depends only on the type, not on a concrete value:

```
garities :: forall a . (Generic a) => Proxy a -> NP (K Int) (Code a)
garities _ = hcmap (Proxy :: Proxy SListI) (K . length . hcollapse)
           $ unPOP $ hpure (K ())
```

We do need a proxy here. This time, it's because the type variable `a` appears only in the constraint `Generic` and as an argument to the type family `Code`. Type families need not be injective (and in fact, `Code` is not), so GHC cannot infer `a` without a proxy.

Let's look at an example:

```
GHCi> garities (Proxy :: Proxy Expr)
```

We can now use the arities to compute weights.

### 3.4.3 Sizing the generator

We are going to use a slightly different approach here, and use the

```
sized :: (Int -> Gen a) -> Gen a
```

function provided by QuickCheck that lets us access an internal `size` parameter. We can also use

```
resize :: Int -> Gen a -> Gen a
```

to reset the size for a subcomputation. So in the following version, we'll check if the requested size is `0`. If so, we will only allow constructors that do not further increase the branching factor (i.e., with arity `0` or `1`). Otherwise, we choose equally between all available constructors.

```
garbitrarySized ::  forall a . (Generic a, All2 Arbitrary (Code a)) => Gen a
garbitrarySized = sized go
  where
    go n = oneof (map snd (filtered table))
      where
        table :: [(Int, Gen a)]
        table = hcollapse
              $ hzipWith aux
                  (garities (Proxy :: Proxy a))
                  (injections `hap` unPOP (hcpure (Proxy :: Proxy Arbitrary) arbitrary))
        aux :: forall x . K Int x -> K (NS (NP Gen) (Code a)) x -> K (Int, Gen a) x
        aux (K arity) (K gen) = K (arity, resize (n `div` arity)
                                          (fmap to (hsequence (SOP gen))))
        filtered | n <= 0    = filter ((<= 1) . fst)
                 | otherwise = id
```

We can now say

```
instance Arbitrary Expr where
  arbitrary = garbitrarySized
```

and – at least for expressions – get reliably terminating random values.

## 3.5 Lenses

## 3.6 Metadata

If one datatype differs from another only in its name, the names of its constructors or the names of its record selectors, then it will have the same code. This is on purpose. As we have seen, many datatype-generic functions can be defined in a way completely agnostic of such metadata, and it is one of the advantages of generics-sop that such functions do not have to worry about it at all.

We can even define a safe coercion function between structurally compatible types:

```
gcoerce :: (Generic a, Generic b, Code a ~ Code b) => a -> b
gcoerce = to . from
```

(This function is rather limited, as it only works for types which have an equal `Code`. It is certainly possible to tweak the function to take into account a certain amount of expansion and perhaps permutation.)

However, we now want to look at functions that *do* need metadata. A classic example are Haskell's `read` and `show` functions. But many other serialization formats also need to know something about the names involved. And sometimes, we might want to know even more, such as the module in which a datatype has been defined, or whether an operator priority has been declared for a constructor.

### 3.6.1 The `DatatypeInfo` type

We store the info about a datatype in a data structure indexed by its code. As we have demonstrated in a few examples earlier – such as the configurable `arbitrary` function – this allows us to use it as an additional input to a particular generic function while still being guaranteed by the type system that it matches up with the structure we are working on.

```
data DatatypeInfo (code :: [[*]]) where
  ADT     :: ModuleName -> DatatypeName -> NP ConstructorInfo xss -> DatatypeInfo xss
  Newtype :: ModuleName -> DatatypeName -> ConstructorInfo '[x] -> DatatypeInfo '['[x]]

type ModuleName      = String
type DatatypeName    = String
type ConstructorName = String
type FieldName       = String
```

We distinguish between datatypes defined via `data` and `newtype`. The latter are restricted to having a single constructor with a single field, which is reflected in the type.

Information about constructors looks as follows:

```
data ConstructorInfo (xs :: [*]) where
  Constructor :: ConstructorName -> ConstructorInfo xs
  Infix       :: ConstructorName -> Associativity -> Fixity -> ConstructorInfo '[x, y]
  Record      :: ConstructorName -> NP FieldInfo xs -> ConstructorInfo xs

data Associativity = LeftAssociative | RightAssociative | NotAssociative
type Fixity = Int
```

Here, we distinguish between normal constructors, constructors with an infix declaration and constructors declared using record notation. We restrict infix constructors to having two arguments, although this is actually incorrect, as Haskell allows constructors with more than two arguments to have infix declarations, and also record constructors to have infix declarations.

For records, we additionally store the field labels:

```
data FieldInfo (x :: *) = FieldInfo FieldName
```

As a final ingredient, we have a class that allows us to obtain the metadata for a specific datatype:

```
class Generic a => HasDatatypeInfo a where
  datatypeInfo :: Proxy a -> DatatypeInfo (Code a)
```

### 3.6.2 Example metadata

The role of `HasDatatypeInfo` is that it is an extension of the `Generic` class. The same issues we've discussed regarding the generation of `Generic` instances also apply to `HasDatatypeInfo`. In particular, generics-sop provides Template Haskell code that can automatically derive `HasDatatypeInfo` for user-defined datatypes, and it also implements a "Generic Generic Programming" translation that converts the metadata provided for GHC's built-in generic representation into metadata suitable for generics-sop.

Let us nevertheless manually define the metadata for our `Expr` type, to get a feeling for how it looks:

```
exprInfo :: DatatypeInfo (Code Expr)
exprInfo =
  ADT "LectureNotes" "Expr"
      $ Constructor "NumLit"
      :* Constructor "BoolLit"
      :* Constructor "Add"
      :* Constructor "If"
      :* Nil
instance HasDatatypeInfo Expr where
  datatypeInfo _ = exprInfo
```

For comparison, let us also look at a simple record type:

```
data Person = Person {name :: String, age :: Int, address :: String}


personInfo :: DatatypeInfo (Code Person)
personInfo =
  ADT "LectureNotes" "Person"
      $ Record "Person"  ( FieldInfo "name"
                         :* FieldInfo "age"
                         :* FieldInfo "address"
                         :* Nil
```

```
                        )
    :* Nil
instance HasDatatypeInfo Person where
  datatypeInfo _ = personInfo
```

### 3.6.3 Useful helper functions

It is helpful to have functions available that can extract certain information more directly:

```
datatypeName :: DatatypeInfo xss -> DatatypeName
datatypeName (ADT     _ n _) = n
datatypeName (Newtype _ n _) = n

constructorInfo :: DatatypeInfo xss -> NP ConstructorInfo xss
constructorInfo (ADT     _ _ i) = i
constructorInfo (Newtype _ _ i) = i :* Nil

constructorName :: ConstructorInfo xs -> ConstructorName
constructorName (Constructor n) = n
constructorName (Infix n _ _) = n
constructorName (Record n _)  = n

constructorNames :: SListI xss => DatatypeInfo xss -> NP (K ConstructorName) xss
constructorNames = hmap (K . constructorName) . constructorInfo
```