A concise introduction to `generics-sop`

Sridhar Ratnakumar

Contents

A fascinating aspect of Lisp-based programming languages is that code is data and data is code. This property, called homoiconicity, is what makes Lisp macros so powerful. This sort of runtime operation done on arbitrary datatypes is called polytypism or datatype genericity. In Haskell, the following two packages provide datatype genericity:

1. GHC Generics
2. generics-sop

While the former comes with `base`, the latter is much easier to write generics code in, and this blog post introduces `generics-sop`.

Motivation

Generic programming is useful in avoiding writing of boilerplate implementations for each datatype that are otherwise similar in some way. This could be a (polytypic) function or a typeclass instance. For example, instead of having to manually write `FromJSON` and `ToJSON` instances for each of your datatypes, you can use generics to derive them automatically. Other examples include pretty printers, parsers, equality functions and route encoders.

Basics

Before diving, we must understand the "SOP" in generics-**sop**.

Datatypes are SOPs under the hood

Haskell has two kinds of datatypes:

1. Algebraic data types, or ADTs
2. Newtypes

Both ADTs and newtypes are a "sum-of-product" (SOP) under the hood. When writing generics-sop code, we operate on these SOPs rather than directly on the datatype, because every datatype is "polymorphic" in their SOP representation. The basic idea is that if you can write a function `SOP -> a`, then you get `SomeDataType -> a` for free for *any* `SomeDataType`. This is called polytypism.

Consider the following ADT (from the `these` package):

```
-- `These` is like `Either`, but with a 3rd possibility of representing both values.
data These a b
  = This a
  | That b
  | These a b
```

Here, `These` is a **sum** type, with `This`, `That` and `These` being its three sum constructors. Each sum constructor themselves are **product** types - inasmuch as, say, the `a` and `b` in the 3rd constructor together represent a product type associated with *that* constructor. The type `These` is therefore a "sum of product".

SOPs are tables

To gain better intuition, we may visualize the `These` SOP in a table form:

Constructor	Arg 1	Arg 2	...
This	a		
That	b		
These	a	b	

As every Haskell datatype is a SOP, they can be (visually) reduced to a table like the above. Each row represents the sum constructor, and the individual cells to the right represent the arguments to the constructors (product type). We can drop the constructor *names* entirely and simplify the table as:

a	
b	
a	b

(*`These`* type visually represented as a table)

Every cell in this table is a unique type. To define this table in Haskell we could use type-level lists, specifically a type-level *list of lists*. The outer list represents the sum constructor, and the inner list represents the products. The *kind* of this table type would then be `[[Type]]`. Indeed this is what generics-sop uses. We can define the table type for `These` in Haskell as follows:

```
type TheseTable a b =
  '[ '[ a ]
     '[ b ]
     '[ a, b ]
  ]
```

If you are confused about this syntax, read the following "Interlude" section.

Interlude: a foray into type-level programming

What is a "kind"? Kinds are to types what types are to terms. For example, the *type of* of the term `"Hello world"` is `String`. The latter is a "type," whereas the former is a "term". Furthermore we can go one level up and ask - what the *kind of* of the type `String` is; and the answer is `Type`. We can clarify this further by explicitly annotating the kinds of types when defining them (just as we annotate the types of terms when defining them):

```
-- Here, we define a term (2nd line) and declare its type (1st line)
someBool :: Bool
someBool = True

-- Here, we define a type (2nd line) and declare its kind (1st line)
```

```
type Bool :: Type
data Bool = False | True
```

Parametrized types, such as `Maybe`, have a type-level function as their kind:

```
type Maybe :: Type -> Type
data Maybe a = Nothing | Just a
```

Here we say that "the type `Maybe` is of kind `Type -> Type`". In other words, `Maybe` is a *type-level function* that takes a type of kind `Type` as an argument and returns another type of the same kind `Type` as its result.

Finally, we are now in a position to understand the kind of `TheseTable` described in the prior section:

```
type TheseTable :: Type -> Type -> [[Type]]
type TheseTable a b =
  '[ '[ a ]
     '[ b ]
     '[ a, b ]
  ]
```

`[Type]` is the kind of type-level lists, and `[[Type]]` is the kind of type-level lists of lists. The tick (`'`) lifts a term into a type. So, while `True` represents a *term* of type `Bool`, `'True` on the other hand represents a *type* of *kind* `Bool`, just as `'[a]` represents a type of the kind `[Type]`. The tick "promotes" a term to be a type. See Datatype promotion in GHC user guide for details.

See An introduction to typeclass metaprogramming as well as Thinking with Types for more on type-level programming.

Let's play with SOPs

Enough theory, let's get our hands dirty in GHCi. If you use Nix, you can clone this repo and run `bin/repl` to get GHCi with everything configured ahead for you.

```
$ git clone https://github.com/srid/generics-sop-examples.git
$ cd ./generics-sop-examples
$ bin/repl
[1 of 1] Compiling Main             ( src/Main.hs, interpreted )
Ok, one module loaded.
*Main>
```

The project already has `generics-sop` and `sop-core` added to the .cabal file, so you should be able to import it:

```
> import Generics.SOP
```

We also have the `these` package added to the .cabal file because it provides the above `These` type from `Data.These` module. To explore the SOP representation of the `These` type, let us do some bootstrapping:

```
> import Data.These
> instance Generic (These a b) -- Derive generics-sop instance
> let breakfast = These "Egg" "Sausage" :: These String String
```

We derived `Generic` on the type and created a term value called `breakfast` (we are eating both eggs and sausages). To get the SOP representation of this value, we can use `from`:

```
> unSOP . from $ breakfast
S (S (Z (I "Egg" :* I "Sausage" :* Nil)))
```

Well explain the above value structure in a bit, but the key thing to realize is that this value corresponds to the 3rd row in the SOP table for `These`:

String	
String	
String	String

Because `breakfast` is a value of the 3rd constructor of `These` and it contains two values (the product of "Egg" and "Sausage"). The corresponding Haskell type for this table:

```
type TheseTable :: [[Type]]
type TheseTable =
  '[ '[ String ]
     '[ String ]
     '[ String, String ]
  ]
```

This type is automatically provided by `generics-sop` whenever we derive a `Generic` instance for the type in question. We did precisely that further above by evaluating `instance Generic (These a b)` in GHCi. Instead of manually defining `TheseTable` as above, deriving `Generic` gives it for free, in the form of `Code a` (viz. `Code (These a b)`).

```
> :k Code (These String String)
Code (These String String) :: [[Type]]
```

In brief, remember this: `Code a` gives us the SOP table *type* for the datatype `a`. Now how do we get the SOP table *value*? That is what `from` is for:

```
> :t (unSOP . from $ breakfast)
(unSOP . from $ breakfast)
  :: NS
       @[Type]
       (NP @Type I)
       (('':)
          @[Type]
          (('':) @Type [Char] ('[] @Type))
          (('':)
             @[Type]
             (('':) @Type [Char] ('[] @Type))
             (('':)
                @[Type]
                (('':) @Type [Char] (('':) @Type [Char] ('[] @Type)))
                ('[] @[Type]))))
```

That is quite a mouthful because type-level lists are not represented cleanly in GHCi. But we can reduce it (in our mind) to the following

```
> :t (unSOP . from $ breakfast)
(unSOP . from $ breakfast)
  :: NS (NP I) '[ [String], [String], [String, String] ]
```

Notice how this is more or less isomorphic to our `TheseTable` definition above. We will explain the `NS` and `NP` parts next.

Interlude: `NS` & `NP`

You wonder what the `NS (NP I)` part refers to in our table type above. `NS` is a n-ary sum; and `NP` an n-ary product. These are explained well in section 2 of Applying Type-Level and Generic Programming in Haskell, but for our purposes - you can treat `NS` as similar to the `Nat` type from the fin package, and `NP` as being similar to the `Vec` type from the vec package.

The difference is that unlike `Vec` (a homogenous list), `NP` is a heterogenous list whose element types are specified by a type-level list.

```
> :k NP I '[String, Int]
NP I '[String, Int] :: Type
```

Just like `Vec` can enforce the size of its homogenous list, `NP` is a heterogenous list of exactly size 2. However, unlike `Vec` we also say that the first element is of type `String` and the second (and the last) element is of type `Int` (hence a heterogenous list). To create a value of this heterogeneous list:

```
> I "Meaning" :* I 42 :* Nil  :: NP I '[String, Int]
I "Meaning" :* I 42 :* Nil
```

This syntax should be unsurprising because `Nil` and `(:*)` are the constructors of the `NP` type:

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

(View haddocks)

The `I` is the identity functor, but it could also be something else like `Maybe` :

```
> Nothing :* Just 42 :* Nil  :: NP Maybe '[String, Int]
Nothing :* Just 42 :* Nil
```

`NS` is the same, except now we are representing the same characteristics (heterogeneity) but for the sum type instead of a product type. A sum of length 'n' over some functor 'f':

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

(View haddocks)

When the value is `Z` it indicates the first sum constructor; when it is `S . Z` it is the second, and so on. Our `breakfast` value above uses `These` , which is the 3rd constructor. So, to construct the SOP representation of this value directly, we would use `S . S . Z` . This is exactly what we saw above (repeated here):

```
-- Note the `S . S . Z`
> unSOP . from $ breakfast
S (S (Z (I "Egg" :* I "Sausage" :* Nil)))
>
> :t (unSOP . from $ breakfast)
(unSOP . from $ breakfast)
  :: NS (NP I) '[ [String], [String], [String, String] ]
```

`NS` 's functor is a `NP I` , and so the inner value of that sum choice is an n-ary product (remember: we are working with a sum-of-product), whose value is `I "Egg" :* I "Sausage" :* Nil` . Putting that product inside a sum, we get `S (S (Z (I "Egg" :* I "Sausage" :* Nil)))` .

Code as data; data as code

Ther SOP representation of `These` can be manually constructed. First we build the constructor arguments (product), and then we build the constructor itself (sum):

```
> let prod = I "Egg" :* I "Sausage" :* Nil :: NP I '[String, String]
> let sum = S $ S $ Z prod :: NP I '[[String], [String], [String, String]]
> :t sum
sum :: NS (NP I) '[[String], [String], [String, String]]
```

And from this representation, we can produce a value of type `These` easily, using `to` :

```
> to @(These String String) (SOP sum)
These "Egg" "Sausage"
```

Let us stop for a moment and reflect on what we just did. By treating the type-definition of `These` ("code") as a generic `SOP` table ("data") – i.e., code as data – we are able to generate a value ("code") for that type ("data") – ie., data as code – but without using the constructors of that type. This is generic programming in Haskell; you program *generically*, without being privy to the actual type used.

This concludes the section on playing with SOPs. Now let us do something useful.

Example 1: generic equality

GHC's stock deriving can be used to derive instances for builtin type classes, like `Eq` , on user-defined datatypes. This works for builtin type classes, but generics-sop (as well as GHC.Generics) comes in handy when you want

to derive generically for arbitrary typeclasses. For a moment, let us assume that GHC had no support for stock deriving; how will we derive our `Eq` instance?

We want a function `geq` that takes *any* datatype `a` (this makes the function polytypic), and does equality check on its arguments. In effect, we want:

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

This function can be further broken down to operate on SOP structures directly, so as to "forget" the specific `a`:

```
geq :: forall a. Generic a => a -> a -> Bool
geq x y = geq' @a (unSOP $ from x) (unSOP $ from y)

geq' :: NS (NP I) (Code a) -> NS (NP I) (Code a) -> Bool
geq' = undefined
```

Our problem has now been reduced to operating on SOP tables, and our task is to implement `geq'`.

At this point you are probably thinking we can just case-match on the arguments, but remember that the n-ary sum type `NS` is a GADT (its type index is dependent on the sum constructor). We have to instead case-match at the *type-level* as it were. This is what type-classes are for. The general pattern is that when wanting a `foo` that case-match'es at type-level, we write a type-class `Foo` and write instances for each case-match pattern.

Naive implementation

For pedagogic reasons, we begin with a naive implementation of `geq'` to illustrate the above. We need a `sumEq` function that checks equality of first constructor and then recurses for others; it will case-match on the outer list. Likewise, for each sum constructor, we will need a `prodEq` that checks equality of its products; and it does so, similarly, by checking equality of the first product and then recurses for the rest; it will case-match on the inner list.

```
geq' :: SumEq (Code a) => NS (NP I) (Code a) -> NS (NP I) (Code a) -> Bool
geq' = sumEq

-- `xss` is a type-level list of lists; `Code a`
class SumEq xss where
  sumEq :: NS (NP I) xss -> NS (NP I) xss -> Bool

instance SumEq '[] where
  sumEq = \case

instance (ProdEq xs, SumEq xss) => SumEq (xs ': xss) where
  -- Both values are the same constructor; so check equality on their products,
  -- using `prodEq`.
  sumEq (Z x) (Z y) = prodEq x y
  -- Recurse on next sum constructor.
  sumEq (S x) (S y) = sumEq x y
  -- Mismatching sum constructor; equality check failed.
  sumEq _ _ = False

class ProdEq xs where
  prodEq :: NP I xs -> NP I xs -> Bool

instance ProdEq '[] where
  prodEq Nil Nil = True

instance (Eq x, ProdEq xs) => ProdEq (x ': xs) where
  -- First product argument should be equal; then we recurse for rest of arguments.
  prodEq (x :* xs) (y :* ys) = x == y && prodEq xs ys
```

Notice how in the first instance for `SumEq` we are "pattern matching" as it were at the type-level and defining the implementation for the scenario of zero sum constructors (not inhabitable). Then, inductively, we define the next instance using recursion. When both arguments are at `Z`, we match their products, using `prodEq`

which is defined similarly. Otherwise, we recurse into the successor constructor (the `x` in `S x`). The story for `ProdEq` is similar.

Finally, we can test that it works:

```
> geq (This True) (That False)
False
> geq (These 42 "Hello") (These 42 "Hello" :: These Int String)
True
```

We just implemented an equality function that works for *any* datatype (with `Generic` instance).

Combinator-based implementation

Hopefully, the above naive implementation is illustratory of how one can "transform" SOP structures straightforwardly using typeclasses. N-ary sums and products need to be processed at type-level, so it is not uncommon to write new type-classes to dispatch on their constructors, as shown above. Typically, however, you do not have to do that because `generics-sop` provides combinators for common operations. Here, we will rewrite the above implementation using these combinators.

The combinators are explained in depth in ATLGP. We will introduce a few in this post. The particular combinators we need for `geq` are:

Combinator	Description	Typeclass it replaces
`hcliftA2`	Lift elements of a NP or NS using given function	`ProdEq`
`hcollapse`	Convert heterogenous structure into homogenous value	`ProdEq`
`ccompare_NS`	Compare two `NS` values	`SumEq`

To appreciate the value of these particular combinators, notice the 3rd column indicating the type-class it intends to replace. Withtout further ado, here is the new (compact) implementation:

```
geq :: forall a. (Generic a, All2 Eq (Code a)) => a -> a -> Bool
geq x y = geq' @a (from x) (from y)

geq' :: All2 Eq (Code a) => SOP I (Code a) -> SOP I (Code a) -> Bool
geq' (SOP c1) (SOP c2) =
  ccompare_NS (Proxy @(All Eq)) False eqProd False c1 c2
  where
    eqProd :: All Eq xs => NP I xs -> NP I xs -> Bool
    eqProd p1 p2 =
      foldl' (&&) True $
        hcollapse $ hcliftA2 (Proxy :: Proxy Eq) eqTerm p1 p2
      where
        eqTerm :: forall a. Eq a => I a -> I a -> K Bool a
        eqTerm a b =
          K $ a == b
```

This code introduces two more aspects to `generics-sop`:

- **Constraint propagation**: When generically transforming SOP structures we want to be able to "propagate" inner constraints outwardly, and this is what the `Proxy` class is being used for here. `All c xs` simply is an alias for `(c x1, c x2, ...)` where `xs` is a type-level list; likewise, `All2 c xss` is `c x11, c x12, ...` where `xss` is type-level list of lists (ie., `Code a ~ xss`). Clearly, we want the `Eq` constraint in table elements to apply to the whole table row and thereon to the table itself. And `All2 Eq (Code a)` on `geq'` specifies this.
- **Constant functor**: The constant functor `K` is defined as `data K a b = K a`; it "discards" the second type parameter, always containing the first. Where you see `K Bool a` we are discarding the polymorphic `a` (the type of the cell in the table), and returning the (constant) type `Bool`. When we transform the structure to be over `K` (using `hcliftA2`), we are essentially making the structure *homogenous* in its elements, which in turn allows us to "collapse" it using `hcollapse` to get a single value out of it (which is what we need to be the result of `geq`).

This is just a brief taste of generics-sop combinators. Read ATLGP for details, and we shall introduce more in the examples below.

Interlude: Specialized combinators Most combinators are polymorphic over the containing structure, and as such their type signatures can be pretty complex to understand. For this reason, you might want to begin with using their *monomorphized* versions which have simpler type signatures. For example, the polymorphic combinator `hcollapse` has the following signature that makes it possible to work with any structure (`NS` or a `NP` , etc).

```
hcollapse :: SListIN h xs => h (K a) xs -> CollapseTo h a
```

This signature is not particularly easier to understand if you are not very familiar with the library. But the monomorphized versions, such as that for `NS` , are more straightforward to understand:

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

These specialized versions typically are suffixed as above (i.e., `_NP`).

Example 2: route encoding

In the first example above we demonstrated how to use generics-sop to generically implement `eq` . Here, we will show a more interesting example. Specifically, how to represent routes for a statically generated site using algebraic data types. We will derive encoders (`route -> FilePath`) for them automatically using generics-sop.

Imagine you are writing a static site in Haskell[1] for your blog posts. Each "route" in that site corresponds to a generated `.html` file. We will use ADTs to represent the routes:

```
data Route
  = Route_Index -- index.html
  | Route_Blog BlogRoute -- blog/*

data BlogRoute
  = BlogRoute_Index -- blog/index.html
  | BlogRoute_Post PostSlug -- blog/${slug}.html

newtype PostSlug = PostSlug {unPostSlug :: Text}
```

To compute the path to the `.html` file for each route, we need a function `encodeRoute :: r -> FilePath` . It is worth creating a typeclass for it because we can recursively encode the ADT:

```
-- Class of routes that can be encoded to a filename.
class IsRoute r where
  encodeRoute :: r -> FilePath
```

Manual implementation

Before writing generic implementation, it is always useful to write the implementation "by hand". Doing so enables us to begin to build an intuition for what the generic version will look like.

```
-- This instance will remain manual.
instance IsRoute PostSlug where
  encodeRoute (PostSlug slug) = T.unpack slug <> ".html"

-- These instances eventually will be generalized.
instance IsRoute BlogRoute where
  encodeRoute = \case
    BlogRoute_Index -> "index.html"
    BlogRoute_Post slug -> "post" </> encodeRoute slug

instance IsRoute Route where
  encodeRoute = \case
    Route_Index -> "index.html"
    Route_Blog br -> "blog" </> encodeRoute br
```

There is nothing we can do about `PostSlug` instance because it is not an ADT, but we *do* want to generically implement `encodeRoute` for both `BlogRoute` and `Route` generically.

[1] Using generators like Hakyll or Ema

Identify the general pattern

After writing the implementation manually, the next step is to make it as general as possible. Try to extract the "general pattern" behind these manual implementations. Looking at the instances above, we can conclude the general pattern as follows:

- To encode `Foo_Bar` in a datatype `Foo`, we drop the `Foo_`, and take the `Bar`. Then we convert it to `bar.html`.
- If a sum constructor has arguments, we check that it has exactly one argument (arity <=1). Then call `encodeRoute` on that argument, and append it to the constructor's encoding using `/`.
  - For example, to encode `BlogPost_Post (PostSlug "hello")` we first encode the constructor as `"post"`, then encode the only argument as `encodeRoute (PostSlug "hello")` which reduces to `"hello.html"`, thus producing the encoding `"post/hello.html"`. Finally when encoding `Route_Blog br` - this gets encoded ito `"blog/post/hello.html"`, inductively.

Write the generic version

Having identified the general pattern, we are now able to write the generic version of `encodeRoute`. Keep in mind the above pattern while you follow the code below.

```
gEncodeRoute :: Generic r => r -> FilePath
gEncodeRoute = undefined
```

To derive route encoding from the constructor name, we need the datatype metadata (provided by `HasDatatypeInfo`) from generics-sop. `constructorInfo . datatypeInfo` gives us the constructor information, from which we will determine the final route encoding using the `hindex` combinator. Effectively, this enables us to produce `"foo.html"` from a sum constructor like `Route_Foo`.

```
gEncodeRoute :: forall r.
  (Generic r, All2 IsRoute (Code r), All IsRouteProd (Code r), HasDatatypeInfo r) =>
  r -> FilePath
gEncodeRoute x = gEncodeRoute' @r (from x)

gEncodeRoute' :: forall r.
  (All2 IsRoute (Code r), All IsRouteProd (Code r), HasDatatypeInfo r) =>
  SOP I (Code r) -> FilePath
gEncodeRoute' (SOP x) =
  -- Determine the contructor name and then strip its prefix.
  let ctorNames :: [ConstructorName] =
        hcollapse $ hmap (K . constructorName) $ datatypeCtors @r
      ctorName = ctorNames !! hindex x
      ctorSuffix = ctorStripPrefix @r ctorName
  -- Encode the product argument, if any, otherwise end the route string with ".html"
   in case hcollapse $ hcmap (Proxy @IsRouteProd) encProd x of
        Nothing -> ctorSuffix <> ".html"
        Just p -> ctorSuffix </> p
  where
    encProd :: (IsRouteProd xs) => NP I xs -> K (Maybe FilePath) xs
    encProd =
      K . hcollapseMaybe . hcmap (Proxy @IsRoute) encTerm
    encTerm :: IsRoute b => I b -> K FilePath b
    encTerm =
      K . encodeRoute . unI

datatypeCtors :: forall a. HasDatatypeInfo a => NP ConstructorInfo (Code a)
datatypeCtors = constructorInfo $ datatypeInfo (Proxy @a)

ctorStripPrefix :: forall a. HasDatatypeInfo a => ConstructorName -> String
ctorStripPrefix ctorName =
  let name = datatypeName $ datatypeInfo (Proxy @a)
   in maybe (error "ctor: bad naming") (T.unpack . T.toLower) $
        T.stripPrefix (T.pack $ name <> "_") (T.pack ctorName)
```

`hcollapse` should be familiar; and `hcmap` is just an alias of `hcliftA` (analogous to `hcliftA2` used in

the above example). New here is `hcollapseMaybe` which is a custom version of `hcollapse` we defined to constrain the number of products to be either zero or one (as it would not make sense for a route tye otherwise); its full implementation[2] is available in the source.

Finally, we make use of `DefaultSignatures` to provide a default implementation in the `IsRoute` class:

```
class IsRoute r where
  encodeRoute :: r -> FilePath
  default encodeRoute ::
    (Generic r, All2 IsRoute (Code r), HasDatatypeInfo r) =>
    r ->
    FilePath
  encodeRoute = gEncodeRoute
```

This, in turn, allows us to derive `IsRoute` arbitrarily via `DeriveAnyClass`, which is to say that we get our `IsRoute` instances for "free":

```
data Route
  = Route_Foo
  | Route_Blog BlogRoute
  deriving stock (GHC.Generic, Eq, Show)
  deriving anyclass (Generic, HasDatatypeInfo, IsRoute)`
```

`encodeRoute Route_Foo` now returns `"foo.html"` and `encodeRoute $ Route_Blog BlogRoute_Index` returns `"blog/index.html"`, all without needing boilerplate implementation.

Example 3: route decoding

As a final example, we shall demonstrate what it takes to *construct* new values. Naturally, our `IsRoute` class above needs a new method, `decodeRoute` for the reverse conversion (perhaps you want to check the validity of links in the generated HTML):

```
class IsRoute r where
  -- | Encode a route to file path on disk.
  encodeRoute :: r -> FilePath
  -- | Decode a route from its encoded filepath
  decodeRoute :: FilePath -> Maybe r

gDecodeRoute :: forall r.
  (Generic r, All2 IsRoute (Code r), HasDatatypeInfo r) =>
  FilePath -> Maybe r
gDecodeRoute fp = undefined
```

[2] In particular, we create a `HCollapseMaybe` constraint that limits `hcollapse` to work on at most 1 product:

```
class HCollapseMaybe h xs where
  hcollapseMaybe :: SListIN h xs => h (K a) xs -> Maybe a

instance HCollapseMaybe NP '[] where
  hcollapseMaybe _ = Nothing

instance HCollapseMaybe NP '[p] where
  hcollapseMaybe (K x :* Nil) = Just x

instance (ps ~ TypeError ('Text "Expected at most 1 product")) => HCollapseMaybe NP (p ': p1 ': ps) where
  hcollapseMaybe _ = Nothing -- Unreachable

class (All IsRoute xs, HCollapseMaybe NP xs) => IsRouteProd xs

instance (All IsRoute xs, HCollapseMaybe NP xs) => IsRouteProd xs
```

Then we change `encProd` to be:

```
encProd :: (IsRouteProd xs) => NP I xs -> K (Maybe FilePath) xs
encProd =
  K . hcollapseMaybe . hcmap (Proxy @IsRoute) encTerm
```

While propagating the `All IsRouteProd (Code r)` constraint all the way up.

`SList`

Generically constructing values is a little more involved, where it useful to know about singleton for type-level lists, `SList`.

```
data SList :: [k] -> Type where
  SNil  :: SList '[]
  SCons :: SListI xs => SList (x ': xs)

-- | Get hold of an explicit singleton (that one can then
-- pattern match on) for a type-level list
--
sList :: SListI xs => SList xs
sList = ...
```

We require heavy use of `sList` to generically implement `decodeRoute`. `sList` pretty much allows us to "case-match" on the type-level list and build our combinators accordingly, as we will see below.

Anamomrphism combinators

To implement `decodeRoute` generically, we are looking to construct a `NS (NP I) (Code r)` depending on which constructor the first path segment of `fp` matches with. Then, we recurse into constructing the inner route for the sum constructor's (only and optional) product type. This recursive building of values is called anamorphism. In particular, we need two anamorphisms: one for the outer sum and another for the inner product.

`generics-sop` already provides `cana_NS` and `cana_NP` as anamorphisms for `NS` and `NP` respectively. However we need a slightly different version of them, to return `Maybe` values instead. We shall define them (prefixed with `m`) accordingly as follows (note the use of `sList`):

```
-- | Like `cana_NS` but returns a Maybe
mcana_NS ::
  forall c proxy s f xs.
  (All c xs) =>
  proxy c ->
  (forall y ys. c y => s (y ': ys) -> Either (Maybe (f y)) (s ys)) ->
  s xs ->
  Maybe (NS f xs)
mcana_NS _ decide = go sList
  where
    go :: forall ys. (All c ys) => SList ys -> s ys -> Maybe (NS f ys)
    go SNil _ = Nothing
    go SCons s = case decide s of
      Left x -> Z <$> x
      Right s' -> S <$> go sList s'

-- | Like `cana_NP` but returns a Maybe
mcana_NP ::
  forall c proxy s f xs.
  (All c xs) =>
  proxy c ->
  (forall y ys. (c y, SListI ys) => s (y ': ys) -> Maybe (f y, s ys)) ->
  s xs ->
  Maybe (NP f xs)
mcana_NP _ uncons = go sList
  where
    go :: forall ys. (All c ys) => SList ys -> s ys -> Maybe (NP f ys)
    go SNil _ = pure Nil
    go SCons s = do
      (x, s') <- uncons s
      xs <- go sList s'
      pure $ x :* xs
```

Implement `gDecodeRoute`

Now we are ready to use a combination of `sList`, `mcana_NS` and `mcana_NP` to implement `gDecodeRoute`:

```
gDecodeRoute :: forall r.
  (Generic r, All IsRouteProd (Code r), All2 IsRoute (Code r), HasDatatypeInfo r) =>
  FilePath -> Maybe r
gDecodeRoute fp = do
  -- We operate on first element of the filepath, and inductively decode the rest.
  basePath : restPath <- pure $ splitDirectories fp
  -- Build the sum using an anamorphism
  to . SOP
    <$> mcana_NS @IsRouteProd @_ @_ @(NP I)
      Proxy
      (anamorphismSum basePath restPath)
      (datatypeCtors @r)
  where
    -- The `base` part of the path should correspond to the constructor name.
    anamorphismSum :: forall xs xss.
     IsRouteProd xs =>
     FilePath ->
     [FilePath] ->
     NP ConstructorInfo (xs ': xss) ->
     Either (Maybe (NP I xs)) (NP ConstructorInfo xss)
    anamorphismSum base rest (p :* ps) =
      fromMaybe (Right ps) $ do
        let ctorSuffix = ctorStripPrefix @r (constructorName p)
        Left <$> case sList @xs of
          SNil -> do
            -- For constructors without arguments, we simply expect the `rest`
            -- of the path to be empty.
            guard $ ctorSuffix <> ".html" == base && null rest
            pure $ Just Nil
          SCons -> do
            -- For constructors with an argument, we ensure that the constructor
            -- name matches the base part and then recurse into decoding the
            -- argument itself.
            guard $ ctorSuffix == base
            pure $
              mcana_NP @_ @_ @_ @I
                (Proxy @IsRoute)
                anamorphismProduct
                Proxy
      where
        anamorphismProduct :: forall y1 ys1.
         (IsRoute y1, SListI ys1) =>
         Proxy (y1 ': ys1) -> Maybe (I y1, Proxy ys1)
        anamorphismProduct Proxy = case sList @ys1 of
          -- We "case match" on the rest of the products, to handle the scenario
          -- of there being exactly one product.
          SNil -> do
            -- Recurse into the only product argument
            guard $ not $ null rest
            r' <- decodeRoute @y1 $ joinPath rest
            pure (I r', Proxy)
          SCons ->
            -- Not reachable, due to HCollapseMaybe constraint
            Nothing
```

We split the path `fp` and process the first path segment by matching it with one of the sum constructors. In `anamorphismSum` we handle the two cases of null product constructor and singleton product constructor (

`mcana_NS` is responsible for recursing into other sum constructors). For null product, we match the file path with "${constructorSuffix}.html" and return immediately. For a single product case, we use `mcana_NP` to build the product. `anammorphismProduct` uses `sList` to case match on the rest of the products (i.e. 2nd, etc.) - and calls `decodeRoute` on the first product only if the rest is empty, which in turn requires us to the `IsRoute` constraint all the way above.

Finally, we use `DefaultSignatures` to specify a default implementation in `IsRoute` class.

Putting it all together

We can test that our code works in ghci:

```
> import RouteEncoding
> encodeRoute Route_Index
"index.html"
> decodeRoute @Route $ encodeRoute Route_Index
Just Route_Index
```

To be completely sure, we can test it with inductive route values:

```
> encodeRoute $ Route_Blog $ BlogRoute_Post "hello"
"blog/post/hello.html"
> decodeRoute @Route "blog/post/hello.html"
Just (Route_Blog (BlogRoute_Post "hello"))
>
```

This concludes our introduction to `generics-sop` .

Further information

- Source code for this blog post
- This ZuriHac talk provides a good introduction to generics-sop
- Applying Type-Level and Generic Programming in Haskell by Andres Löh acts as a lengthy tutorial cum documentation for generics-sop