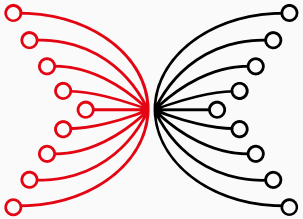
Generic programming

Haskell and Cryptocurrencies

Dr. Andres Löh, Well-Typed LLP
Dr. Lars Brünjes, IOHK

2018-03-01



INPUT | OUTPUT

Goals

- Introduce datatype-generic programming.
- GHC generics.
- `generics-sop`.

Motivation

User-defined datatypes are fantastic

- Easy to introduce.
- Distinguished from existing types by the compiler.
- Added safety.
- Can use domain-specific names for types and constructors.
- Make the program more readable.

User-defined datatypes are problematic

- New datatypes have no associated library.
- Cannot be compared for equality, cannot be (de)serialized, cannot be traversed, ...

User-defined datatypes are problematic

- New datatypes have no associated library.
- Cannot be compared for equality, cannot be (de)serialized, cannot be traversed, ...

Fortunately, there is `deriving`.

Derivable classes

In Haskell 2010:

```
Eq, Ord, Enum, Bounded, Read, Show
```

Derivable classes

In Haskell 2010:

```
Eq, Ord, Enum, Bounded, Read, Show
```

In GHC (in addition to the ones above):

```
Functor, Foldable, Traversable, Typeable,
Data, Generic
```

What about other classes?

For many additional classes, we can come up with an informal algorithm that explains how to derive them. we can intuitively derive instances.

But can we also do it in practice?

Options for deriving other classes

- Template Haskell.
- External preprocessor (such as `data-derive`).
- GHC `Generic` support.
- One of many other libraries for datatype-generic programming in Haskell.

Options for deriving other classes

- Template Haskell.
- External preprocessor (such as `data-derive`).
- GHC `Generic` support.
- One of many other libraries for datatype-generic programming in Haskell.

In this lecture, we'll look at the ideas of datatype-generic programming that are the foundation for the latter two approaches.

GHC generics – the user perspective

Using a generic function

Step 1

Define a new datatype and derive `Generic` for it.

```haskell
data MyType a b =
    Flag Bool
  | Combo (a, a)
  | Other b Int (MyType a a)
  deriving Generic
```

This requires the `DeriveGeneric` language extension and importing the `Generic` type class from `GHC.Generics`.

Using a generic function

Step 2a

Use a library that makes use of GHC `Generic` and give an empty instance declaration for a suitable type class:

```
import Data.Binary


instance
  (Binary a, Binary b) => Binary (MyType a b)
```

That is all. The generic implementation is derived by GHC and supplied automatically.

Using a generic function

Step 2b

With recent GHC versions, we can also use `deriving` instead of an empty instance declaration:

```
data MyType a b =
    Flag Bool
  | Combo (a, a)
  | Other b Int (MyType a a)
  deriving (Generic, Binary)
```

This requires the `DeriveAnyClass` extension. It only works in a truly robust way from GHC 8.2 onwards (although in some simple cases it may work with 8.0 or even 7.10).

Deriving `Generic` is still necessary!

Explaining how to derive a class

Equality as an example

Let's work with our "copy" of the `Eq` class:

```haskell
class Eq' a where
  eq' :: a -> a -> Bool
```

Equality as an example

Let's work with our "copy" of the `Eq` class:

```
class Eq' a where
  eq' :: a -> a -> Bool
```

Let's define some instances by hand.

Equality on binary trees

```
data T = A | N T T
```

```
instance Eq' T where
  eq'  A         A         = True
  eq' (N x1 y1) (N x2 y2) = eq' x1 x2 && eq' y1 y2
  eq'  _         _         = False
```

Equality on another type

```haskell
data Choice =
  I Int | C Char | B Choice Bool | S Choice
```

Equality on another type

```
data Choice =
  I Int | C Char | B Choice Bool | S Choice
```

Assuming instances for `Int`, `Char`, `Bool`:

```
instance Eq' Choice where
  eq' (I n1   ) (I n2   ) = eq' n1 n2
  eq' (C c1   ) (C c2   ) = eq' c1 c2
  eq' (B x1 b1) (B x2 b2) = eq' x1 x2 && eq' b1 b2
  eq' (S x1   ) (S x2   ) = eq' x1 x2
  eq' _         _         = False
```

What is the pattern?

- How many cases does the function definition have?
- What is on the right hand sides?

What is the pattern?

- How many cases does the function definition have?
- What is on the right hand sides?
- How many clauses are there in the conjunctions on each right hand side?

What is the pattern?

- How many cases does the function definition have?
- What is on the right hand sides?
- How many clauses are there in the conjunctions on each right hand side?

Relevant concepts:

- number of constructors in datatype,
- number of fields per constructor,
- recursion leads to recursion,
- other types lead to invocation of equality on those types.

More datatypes

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

More datatypes

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

```
instance Eq' a => Eq' (Tree a) where
  eq' (Leaf n1   ) (Leaf n2   ) = eq' n1 n2
  eq' (Node x1 y1) (Node x2 y2) =
    eq' x1 x2 && eq' y1 y2
  eq'  _           _            = False
```

Yet another equality function

This is often called a *rose tree*:

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

Yet another equality function

This is often called a *rose tree*:

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

Assuming an instance for lists:

```
instance Eq' a => Eq' (Rose a) where
  eq' (Fork x1 xs1) (Fork x2 xs2) =
    eq' x1 x2 && eq' xs1 xs2
```

More concepts

- Parameterization of types is reflected by parameterization of the functions (via constraints on the instances).
- Using parameterized types in other types then works as expected.

The equality pattern

In order to define equality for a datatype:

- introduce a parameter for each parameter of the datatype,
- introduce a case for each constructor of the datatype,
- introduce a final catch-all case returning `False`,
- for each of the other cases, compare the constructor fields pair-wise and combine them using `(&&)`,
- for each field, use the appropriate equality instance.

The equality pattern

In order to define equality for a datatype:

- introduce a parameter for each parameter of the datatype,
- introduce a case for each constructor of the datatype,
- introduce a final catch-all case returning `False`,
- for each of the other cases, compare the constructor fields pair-wise and combine them using `(&&)`,
- for each field, use the appropriate equality instance.

If we can describe it, *can we write a program to do it?*

Datatype-generic programming

The key idea of datatype-generic programming

- Represent a type `A` as an isomorphic type `Rep A`.

The key idea of datatype-generic programming

- Represent a type `A` as an isomorphic type `Rep A`.
- If a limited number of type constructors is used to build `Rep A`,

The key idea of datatype-generic programming

- Represent a type `A` as an isomorphic type `Rep A`.
- If a limited number of type constructors is used to build `Rep A`,
- then functions defined on each of these type constructors

The key idea of datatype-generic programming

- Represent a type `A` as an isomorphic type `Rep A`.
- If a limited number of type constructors is used to build `Rep A`,
- then functions defined on each of these type constructors
- can be lifted to work on the original type `A`

The key idea of datatype-generic programming

- Represent a type `A` as an isomorphic type `Rep A`.
- If a limited number of type constructors is used to build `Rep A`,
- then functions defined on each of these type constructors
- can be lifted to work on the original type `A`
- and thus on any representable type.

Many options

What to choose as representation `Rep` of a type?

Many options

What to choose as representation `Rep` of a type?

- There are many different options.
- The choice influences which datatypes can be represented, how easy it is to define certain generic functions, and how the resulting generic programming library looks and feels.
- This is one of the reasons why there are so many different approaches to datatype-generic programming in Haskell.

Many options

What to choose as representation `Rep` of a type?

- There are many different options.
- The choice influences which datatypes can be represented, how easy it is to define certain generic functions, and how the resulting generic programming library looks and feels.
- This is one of the reasons why there are so many different approaches to datatype-generic programming in Haskell.

In the following, we will primarily look at one particular representation, called the binary sums of products representation.

Choice between constructors

Which type best encodes choice between constructors?

Choice between constructors

Which type best encodes choice between constructors?

Well, let's restrict to two constructors first.

Choice between constructors

Which type best encodes choice between constructors?

Well, let's restrict to two constructors first.

Booleans encode choice, but do not provide information what the choice is about.

Choice between constructors

Which type best encodes choice between constructors?

Well, let's restrict to two constructors first.

Booleans encode choice, but do not provide information what the choice is about.

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

Choice between constructors

Which type best encodes choice between constructors?

Well, let's restrict to two constructors first.

Booleans encode choice, but do not provide information what the choice is about.

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

Choice between three things:

```
type Either3 a b c = Either a (Either b c)
```

Combining constructor fields

Which type best encodes combining fields?

Combining constructor fields

Which type best encodes combining fields?

Again, let's just consider two of them.

Combining constructor fields

Which type best encodes combining fields?

Again, let's just consider two of them.

```
data (a, b) = (a, b)
```

Combining constructor fields

Which type best encodes combining fields?

Again, let's just consider two of them.

```
data (a, b) = (a, b)
```

Combining three fields:

```
type Triple a b c = (a, (b, c))
```

What about constructors without arguments?

We need another type.

What about constructors without arguments?

We need another type.

Well, how many values does a constructor without argument encode?

What about constructors without arguments?

We need another type.

Well, how many values does a constructor without argument encode?

```
data () = ()
```

Representing types

Representing types

To keep representation and original types apart, let's define isomorphic copies of the types we need:

```
data U       = U
data a :+: b = L a | R b
data a :*: b = a :*: b
```

Representing types

To keep representation and original types apart, let's define isomorphic copies of the types we need:

```
data U       = U
data a :+: b = L a | R b
data a :*: b = a :*: b
```

We can now get started:

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

How do we represent `Bool`?

Representing types

To keep representation and original types apart, let's define isomorphic copies of the types we need:

```
data U       = U
data a :+: b = L a | R b
data a :*: b = a :*: b
```

We can now get started:

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

How do we represent `Bool`?

```
type RepBool = U :+: U
```

A class for representable types

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

The type `Rep` is an example of an associated type.

A class for representable types

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

The type `Rep` is an example of an associated type.

Equivalent to defining `Rep` separately as a type family:

```haskell
type family Rep a
```

Representable Booleans

```
instance Generic Bool where
  type Rep Bool = U :+: U

  from False = L U
  from True  = R U

  to   (L U) = False
  to   (R U) = True
```

Representable lists

```
instance Generic [a] where
  type Rep [a] = U :+: (a :*: [a])
  from []                 = L U
  from (x : xs)           = R (x :*: xs)
  to   (L U             ) = []
  to   (R (x :*: xs)) = x : xs
```

Representable lists

```
instance Generic [a] where
  type Rep [a] = U :+: (a :*: [a])
  from []                 = L U
  from (x : xs)           = R (x :*: xs)
  to   (L U             ) = []
  to   (R (x :*: xs)) = x : xs
```

Note:

- shallow transformation,
- no constraint on `Generic a` required.

Representable trees

```
instance Generic (Tree a) where
  type Rep (Tree a) = a :+: (Tree a :*: Tree a)

  from (Leaf n    ) = L n
  from (Node x y  ) = R (x :*: y)

  to   (L n       ) = Leaf n
  to   (R (x :*: y)) = Node x y
```

Representable rose trees

```haskell
instance Generic (Rose a) where
  type Rep (Rose a) = a :*: [Rose a]
  from (Fork x xs) = x :*: xs
  to   (x :*: xs  ) = Fork x xs
```

Representing primitive types

We do not define instances of `Generic` for primitive types such as `Int`, `Char` or `Float`.

Representing primitive types

We do not define instances of `Generic` for primitive types such as `Int`, `Char` or `Float`.

This will imply that we cannot derive a generic function for such primitive types.

If we want a function to work on these types, we have to provide an appropriate class instance manually.

Back to equality

Intermediate summary

- We have defined class `Generic` that maps datatypes to representations built up from `U`, `(:+:)`, `(:*:)` and other datatypes.
- If we can define equality on the representation types, then we should be able to obtain a generic equality function.
- Let us apply the informal recipe from earlier.

A class for generic equality

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

Instance for sums

```
instance (GEq a, GEq b) => GEq (a :+: b) where
  geq (L a1) (L a2) = geq a1 a2
  geq (R b1) (R b2) = geq b1 b2
  geq  _      _     = False
```

Instance for products and unit

```
instance (GEq a, GEq b) => GEq (a :*: b) where
  geq (a1 :*: b1) (a2 :*: b2) =
    geq a1 a2 && geq b1 b2

instance GEq U where
  geq U U = True
```

Instances for primitive types

```
instance GEq Int where
  geq = ((==) :: Int -> Int -> Bool)
```

What now?

Dispatching to the representation type

```haskell
defaultEq ::
  (Generic a, GEq (Rep a)) => a -> a -> Bool
defaultEq x y = geq (from x) (from y)
```

Dispatching to the representation type

```haskell
defaultEq ::
  (Generic a, GEq (Rep a)) => a -> a -> Bool
defaultEq x y = geq (from x) (from y)
```

Defining generic instances is now trivial:

```haskell
instance           GEq Bool     where
  geq = defaultEq
instance GEq a => GEq [a]       where
  geq = defaultEq
instance GEq a => GEq (Tree a) where
  geq = defaultEq
instance GEq a => GEq (Rose a) where
  geq = defaultEq
```

Dispatching to the representation type

```haskell
defaultEq ::
  (Generic a, GEq (Rep a)) => a -> a -> Bool
defaultEq x y = geq (from x) (from y)
```

Or with the `DefaultSignatures` language extension:

```haskell
class GEq a where
  geq :: a -> a -> Bool
  default geq ::
    (Generic a, GEq (Rep a)) => a -> a -> Bool
  geq = defaultEq

instance GEq Bool
instance GEq a => GEq [a]
instance GEq a => GEq (Tree a)
instance GEq a => GEq (Rose a)
```

Dispatching to the representation type

```haskell
defaultEq ::
  (Generic a, GEq (Rep a)) => a -> a -> Bool
defaultEq x y = geq (from x) (from y)
```

Or with `DeriveAnyClass` / `StandaloneDeriving`:

```haskell
class GEq a where
  geq :: a -> a -> Bool
  default geq ::
    (Generic a, GEq (Rep a)) => a -> a -> Bool
  geq = defaultEq

deriving instance GEq Bool
deriving instance GEq a => GEq [a]
deriving instance GEq a => GEq (Tree a)
deriving instance GEq a => GEq (Rose a)
```

Isn’t this as bad as before?

Amount of work

Question

Haven't we just replaced some tedious work (defining equality for a type) by some other tedious work (defining a representation for a type)?

Amount of work

Question

Haven't we just replaced some tedious work (defining equality for a type) by some other tedious work (defining a representation for a type)?

Yes, *but*:

- The representation has to be given only once, and works for potentially many generic functions.
- Since there is a single representation per type, it could be generated automatically by some other means (compiler support, TH).
- In other words, it's sufficient if we can use `deriving` on class `Generic`.

So can we derive `Generic`?

So can we derive `Generic`?

Yes (with `DeriveGeneric`) ...

So can we derive `Generic`?

Yes (with `DeriveGeneric`) …

…but the representations in GHC are not quite as simple as we've pretended before:

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

So can we derive `Generic`?

Yes (with `DeriveGeneric`) …

…but the representations in GHC are not quite as simple as we've pretended before:

```
class Generic a where
  type Rep a :: * -> *

  from :: a -> Rep a x
  to   :: Rep a x -> a
```

Representation types are actually of kind `* -> *`.

An extra argument?

- It's a pragmatic choice.
- Facilitates some things, because we also want to derive classes parameterized by type constructors (such as `Functor`).
- For now, let's just try to "ignore" the extra argument.

Simple vs. GHC representation

Old:

```
type instance Rep (Tree a) = a :+: (Tree a :*: Tree a)
```

New:

```
type instance Rep (Tree a) =
  D1 (MetaData "Tree" "Main" "main" False)
    (C1 (MetaCons "Leaf" PrefixI False)
       (S1 (MetaSel Nothing NoSourceUnpackedness NoSourceStrictness DecidedLazy)
          (Rec0 a)
       )
      :+:
     C1 (MetaCons "Node" PrefixI False)
       (S1 (MetaSel Nothing NoSourceUnpackedness NoSourceStrictness DecidedLazy)
          (Rec0 (Tree a))
         :*:
        S1 (MetaSel Nothing NoSourceUnpackedness NoSourceStrictness DecidedLazy)
          (Rec0 (Tree a))
       )
    )
```

Simple vs. GHC representation

Old:

```
type instance Rep (Tree a) = a :+: (Tree a :*: Tree a)
```

New:

```
type instance Rep (Tree a) =


                a

      :+:

       (
                 Tree a
          :*:

                 Tree a
       )
```

Familiar components

Everything is now lifted to kind `* -> *`:

```
data U1       a = U1
data (f :+: g) a = L1 (f a) | R1 (g a)
data (f :*: g) a = f a :*: g a
```

All the extra noise

- Every ocurrence of a type as an argument is wrapped in an extra application of `Rec0`.
- This is to detect where the “structure” combinators end and the new type starts.

All the extra noise

- Every ocurrence of a type as an argument is wrapped in an extra application of `Rec0`.
- This is to detect where the “structure” combinators end and the new type starts.
- All the metadata is included a type-level information, too.
- This includes names of the datatypes, constructors and record selectors, but also fixity info for infix constructors, and strictness information.
- This is for defining generic functions that make use of this info (consider e.g. `Show` or `Read`).

Extra constructors

```
type Rec0 = K1 R

newtype K1 i c p = K1 {unK1 :: c}
```

There is no other instantiation of `K1` than `R` actually being used anymore.

Extra constructors

```
type Rec0 = K1 R

newtype K1 i c p = K1 {unK1 :: c}
```

There is no other instantiation of `K1` than `R` actually being used anymore.

```
type D1 = M1 D  -- for datatypes
type C1 = M1 C  -- for constructors
type S1 = M1 S  -- for selectors

newtype M1 i (c :: Meta) f p = M1 {unM1 :: f p}
```

Used to capture various kinds of metadata.

Adapting the equality class(es)

Works on representation types:

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

Works on “normal” types:

```
class GEq a where
  geq :: a -> a -> Bool
  default geq ::
    (Generic a, GEq' (Rep a)) => a -> a -> Bool
  geq x y = geq' (from x) (from y)
```

Instance for `GEq Int` and other primitive types as before.

Adapting the equality class(es) – contd.

```
instance (GEq' f, GEq' g) => GEq' (f :+: g) where
  geq' (L1 x) (L1 y) = geq' x y
  geq' (R1 x) (R1 y) = geq' x y
  geq' _      _      = False
```

Similarly for `:*:` and `U1`.

Adapting the equality class(es) – contd.

```
instance (GEq' f, GEq' g) => GEq' (f :+: g) where
  geq' (L1 x) (L1 y) = geq' x y
  geq' (R1 x) (R1 y) = geq' x y
  geq' _      _      = False
```

Similarly for `:*:` and `U1`.

An instance for constant types:

```
instance GEq a => GEq' (K1 t a) where
  geq' (K1 x) (K1 y) = geq x y
```

Adapting the equality classes – contd.

For equality, we ignore all meta information:

```
instance GEq' f => GEq' (M1 t i f) where
  geq' (M1 x) (M1 y) = geq' x y
```

All meta information is grouped under a single datatype, so that we can easily ignore it all if we want to, such as here.

Adapting the equality classes – contd.

For equality, we ignore all meta information:

```
instance GEq' f => GEq' (M1 t i f) where
  geq' (M1 x) (M1 y) = geq' x y
```

All meta information is grouped under a single datatype, so that we can easily ignore it all if we want to, such as here.

Functions such as `show` and `read` can be implemented generically by accessing meta information.

Constructor classes

To cover classes such as `Functor`, `Traversable`, `Foldable` generically, we need a way to map between a type *constructor* and its representation:

```
class Generic1 f where
  type Rep1 f :: * -> *

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

Use the same representation type constructors, plus

```
data Par1 p      = Par1  {unPar1  :: p      }
data Rec1 f p    = Rec1  {unRec1  :: f p    }
data (:.:) f g p = Comp1 {unComp1 :: f (g p)}
```

GHC is able to derive `Generic1`, too.

A short glimpse at generics-sop

GHC Generics – a summary

- It is very useful that we can extend the set of derivable type classes and add functionality that works on many datatypes.
- Many packages on Hackage use this feature.
- For the user, this makes instantiating type classes to new types extremely easy in most cases – they can simply provide an empty instance, or, with `DeriveAnyClass`, include the class in a `deriving` clause.

GHC Generics – a summary

- It is very useful that we can extend the set of derivable type classes and add functionality that works on many datatypes.
- Many packages on Hackage use this feature.
- For the user, this makes instantiating type classes to new types extremely easy in most cases – they can simply provide an empty instance, or, with `DeriveAnyClass`, include the class in a `deriving` clause.
- However, there are also some disadvantages …

Disadvantages of GHC generics

- The representations are quite complicated (extra type argument for `Generic1`, all the metadata noise).
- Types are always sums of products, but the kinds do not enforce this. E.g., empty products are represented by `U1`, which otherwise does not occur. The `:*:` constructor only occurs if there are at least two fields.
- We cannot directly express that certain functions should only be applicable to certain sub-classes of datatypes, such as datatypes with only a single constructor.
- Generic functions are always defined by defining class instances for all the representation type constructors, which does not exactly encourage code reuse among generic functions.

Introducing `generics-sop`

The `generics-sop` library addresses these problems:

- It uses n-ary sums and products in the representation, rather than binary sums and products.
- It enforces that there is one sum layer on top of one product layer on top of occurrences of argument types.
- Metadata is separated from the normal representation and can be ignored as long as it is not needed by a generic function.
- Higher-order functions are provided that can be reused to build generic functions by composing existing ones.
- The price is that `generics-sop` requires many recent type system extensions (that we already discussed).

A different `Generic` class

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

type Rep a = SOP I (Code a)
```

A different `Generic` class

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

type Rep a = SOP I (Code a)
```

The representation `Rep` is a `SOP` (for sum of products) instantiated to the identity type `I` and the `Code` – a type-level list of lists.

Sums

The choice between several constructors is expressed using the following GADT – `NS` is for n-ary sum:

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

Sums

The choice between several constructors is expressed using the following GADT – `NS` is for n-ary sum:

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

- There is no constructor targeting `NS f '[]`.
- The `Z` constructor targets the first option.
- The `S` constructor targets any other option.

Types of the sum constructors

```
GHCi> :t Z
Z :: f x -> NS f (x ': xs)

GHCi> :t S . Z
S . Z :: f x1 -> NS f (x ': x1 ': xs)

GHCi> :t S . S . Z
S . S . Z :: f x2 -> NS f (x1 ': x ': x2 ': xs)
```

(The numbers assigned by GHCi are unfortunately not intuitive at all.)

Products

The sequence of arguments of a single constructor is expressed using the following GADT – `NP` is for n-ary product:

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

infixr 5 :*
```

Products

The sequence of arguments of a single constructor is expressed using the following GADT – `NP` is for n-ary product:

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

infixr 5 :*
```

This type is (apart from the flipped order of type arguments) the same as `Env` that we have discussed before.

Sums of products

```
newtype SOP f xss = SOP {unSOP :: NS (NP f) xss}
```

A sum of products is an `NS`, where the elements are of type `NP`, and the elements of that are of type `f`.

Sums of products

```
newtype SOP f xss = SOP {unSOP :: NS (NP f) xss}
```

A sum of products is an `NS`, where the elements are of type `NP`, and the elements of that are of type `f`.

An `SOP` is parameterized over a type-level lists of lists. The outer list corresponds to the choice between constructors, the inner lists correspond to the arguments of each constructor.

Example: `Bool`

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

  from False = SOP (Z    Nil )
  from True  = SOP (S (Z Nil))

  to (SOP (Z    Nil) ) = False
  to (SOP (S (Z Nil))) = True
```

Example: `Tree`

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

```haskell
instance Generic (Tree a) where
  type Code (Tree a) = '['[a], '[Tree a, Tree a]]
  from (Leaf x)   = SOP (Z    (I x :* Nil))
  from (Node l r) = SOP (S (Z (I l :* I r :* Nil)))
  to (SOP (Z    (I x :* Nil)))         = Leaf x
  to (SOP (S (Z (I l :* I r :* Nil)))) = Node l r
```

A generic monoid instance for products

Revisiting monoids

```haskell
class Monoid a where
  mempty  :: a
  mappend :: a -> a -> a
```

Revisiting monoids

```haskell
class Monoid a where
  mempty  :: a
  mappend :: a -> a -> a
```

We can easily derive a `Monoid` instance for all product types if their components are `Monoid`s, by lifting the `mzero` and `mappend` operations “component-wise”.

Restricting to product types

We can easily express what it means to be a product type in the vocabulary of `generics-sop`:

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

This is a synonym of kind `Constraint`.

It restricts the outer list of the code to have a single element, so the type must have a single constructor.

`mempty` using `generics-sop`

```
gmempty ::
  (IsProductType a xs, All Monoid xs) => a
gmempty =
    to . SOP . Z
  $ cpure_NP
      (Proxy  @ Monoid)
      (I mempty)
```

`mempty` using `generics-sop`

```
gmempty ::
  (IsProductType a xs, All Monoid xs) => a
gmempty =
    to . SOP . Z
  $ cpure_NP
      (Proxy @ Monoid)
      (I mempty)
```

- `to` turns the representation into an `a`,
- `SOP` is just wrapping the sum of products,
- `Z` creates a value of the first (and only) constructor.

`mempty` using `generics-sop`

```
gmempty ::
  (IsProductType a xs, All Monoid xs) => a
gmempty =
    to . SOP . Z
  $ cpure_NP
      (Proxy  @ Monoid)
      (I mempty)
```

```
cpure_NP ::
  All c xs
  => Proxy c
  -> (forall x . c x => f x) -> NP f xs
```

is a *constrained* form of `replicate`.

`mempty` using `generics-sop`

```
gmempty ::
  (IsProductType a xs, All Monoid xs) => a
gmempty =
    to . SOP . Z
  $ cpure_NP
      (Proxy  @ Monoid)
      (I mempty)
```

The `All` constraint is the same we discussed in the context of environments already:

```
All c xs
```

says that `c` holds for all elements of `xs`.

`mempty` using `generics-sop`

```
gmempty ::
  (IsProductType a xs, All Monoid xs) => a
gmempty =
    to . SOP . Z
  $ cpure_NP
      (Proxy  @ Monoid)
      (I mempty)
```

```
cpure_NP (Proxy  @ Monoid) (I mempty)
```

creates a product of `I mempty` applications, using `mempty` for every argument of the constructor.

`mappend` using `generics-sop`

```
gmappend ::
  (IsProductType a xs, All Monoid xs)
   => a -> a -> a
gmappend x y =
    to . SOP . Z
   $ go (unZ (unSOP (from x))) (unZ (unSOP (from y)))
  where
    go = czipWith_NP (Proxy  @ Monoid) (mapIII mappend)
```

`mappend` using `generics-sop`

```haskell
gmappend ::
  (IsProductType a xs, All Monoid xs)
   => a -> a -> a
gmappend x y =
    to . SOP . Z
   $ go (unZ (unSOP (from x))) (unZ (unSOP (from y)))
  where
    go = czipWith_NP (Proxy  @ Monoid) (mapIII mappend)
```

```haskell
unZ :: NS f '[x] -> f x
unZ (Z x) = x
```

extracts from a single-choice sum.

`mappend` using `generics-sop`

```
gmappend ::
  (IsProductType a xs, All Monoid xs)
   => a -> a -> a
gmappend x y =
    to . SOP . Z
   $ go (unZ (unSOP (from x))) (unZ (unSOP (from y)))
  where
    go = czipWith_NP (Proxy  @ Monoid) (mapIII mappend)
```

```
czipWith_NP ::
  All c xs
   => Proxy c
   -> (forall x . c x => f x -> g x -> h x)
   -> NP f xs -> NP g xs -> NP h xs
```

is a *constrained* `zipWith`.

`mappend` using `generics-sop`

```
gmappend ::
  (IsProductType a xs, All Monoid xs)
   => a -> a -> a
gmappend x y =
    to . SOP . Z
   $ go (unZ (unSOP (from x))) (unZ (unSOP (from y)))
  where
    go = czipWith_NP (Proxy  @ Monoid) (mapIII mappend)
```

```
mapIII :: (a -> b -> c) -> I a -> I b -> I c
mapIII f (I a) (I b) = I (f a b)
```

lifts a function to `I` types.

`mappend` using `generics-sop`

```
gmappend ::
  (IsProductType a xs, All Monoid xs)
   => a -> a -> a
gmappend x y =
    to . SOP . Z
   $ go (unZ (unSOP (from x))) (unZ (unSOP (from y)))
  where
    go = czipWith_NP (Proxy @ Monoid) (mapIII mappend)
```

So the function `gmappend` just combines corresponding constructor arguments using recursive invocations of `mappend`.

Testing

```haskell
data Test = MkTest [Char] (Sum Int)
```

Testing

```haskell
data Test = MkTest [Char] (Sum Int)
```

```haskell
instance Monoid Test where
  mempty = gmempty
  mappend = gmappend
```

Testing

```haskell
data Test = MkTest [Char] (Sum Int)
```

```haskell
instance Monoid Test where
  mempty = gmempty
  mappend = gmappend
```

```
GHCi> mempty :: Test
MkTest [] (Sum 0)
GHCi> mappend
  (MkTest "x" (Sum 3)) (MkTest "y" (Sum 7))
MkTest "xy" (Sum 10)
```

Conclusions

- Generic programming is a powerful programming technique.
- GHC Generics is more widely used, but `generics-sop` leads perhaps to more concise programs.
- There are also many other interesting packages in the area (most notably `uniplate` and `syb`).

Outlook: Template Haskell (TH)

- A full meta-programming extension to Haskell.
- Has the full syntax tree. Can do much more.
- You have to do more work to derive classes using TH.
- It's trickier to get it right. Corner cases. Name manipulation.
- Datatype-generic functions are type-checked, TH only checks the generated code.