Datatype-generic Programming in Haskell
An introduction

Andres Löh

Well-Typed LLP

23 January 2012
Haven’t you ever wondered how deriving works?
Equality on binary trees

```
data T = L | N T T
```

Let’s try ourselves:
Equality on binary trees

data T = L | N T T

Let's try ourselves:

\[ \text{eqT} :: T \rightarrow T \rightarrow \text{Bool} \]

\[ \text{eqT} \ L \ L = \text{True} \]

\[ \text{eqT} \ (N \ x_1 \ y_1) \ (N \ x_2 \ y_2) = \text{eqT} \ x_1 \ x_2 \ \&\& \ \text{eqT} \ y_1 \ y_2 \]

\[ \text{eqT} \ _ \ _ = \text{False} \]
Equality on binary trees

```
data T = L | N T T
```

Let's try ourselves:

```
eqT :: T → T → Bool
eqT L L = True
eqT (N x₁ y₁) (N x₂ y₂) = eqT x₁ x₂ && eqT y₁ y₂
eqT _ _ = False
```

Easy enough, let’s try another . . .
Equality on another type

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

eqChoice :: Choice → Choice → Bool

eqChoice (I n1) (I n2) = eqInt n1 n2

eqChoice (C c1) (C c2) = eqChar c1 c2

eqChoice (B x1 b1) (B x2 b2) = eqChoice x1 x2 && eqBool b1 b2

eqChoice (S x1) (S x2) = eqChoice x1 x2

eqChoice = False

Do you see a pattern?
Equality on another type

\[
data \text{ Choice } = \text{ I Int} | \text{ C Char} | \text{ B Choice Bool} | \text{ S Choice}
\]

\[
eq\text{Choice} :: \text{Choice} \rightarrow \text{Choice} \rightarrow \text{Bool}
\]

\[
eq\text{Choice} (\text{I } n_1) (\text{I } n_2) = \text{eqInt } n_1 \ n_2
\]
\[
eq\text{Choice} (\text{C } c_1) (\text{C } c_2) = \text{eqChar } c_1 \ c_2
\]
\[
eq\text{Choice} (\text{B } x_1 \ b_1) (\text{B } x_2 \ b_2) = \text{eqChoice } x_1 \ x_2 \ &&
\]
\[
\text{eqBool } b_1 \ b_2
\]
\[
eq\text{Choice} (\text{S } x_1) (\text{S } x_2) = \text{eqChoice } x_1 \ x_2
\]
\[
eq\text{Choice } _ \ _ = \text{False}
\]
Equality on another type

\[
\textbf{data} \ \text{Choice} = \text{I Int} | \text{C Char} | \text{B Choice Bool} | \text{S Choice}
\]

\[
\text{eqChoice :: Choice} \rightarrow \text{Choice} \rightarrow \text{Bool}
\]

\[
\text{eqChoice} (\text{I } n_1) (\text{I } n_2) = \text{eqInt } n_1 \ n_2
\]

\[
\text{eqChoice} (\text{C } c_1) (\text{C } c_2) = \text{eqChar } c_1 \ c_2
\]

\[
\text{eqChoice} (\text{B } x_1 \ b_1) (\text{B } x_2 \ b_2) = \text{eqChoice } x_1 \ x_2 \ \&\& \ \text{eqBool } b_1 \ b_2
\]

\[
\text{eqChoice} (\text{S } x_1) (\text{S } x_2) = \text{eqChoice } x_1 \ x_2
\]

\[
\text{eqChoice} _ _ = \text{False}
\]

Do you see a pattern?
A pattern for defining equality

- How many cases does the function definition have?
- What is on the right hand sides?
A pattern for defining equality

- 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?
A pattern for defining equality

- 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

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

Like before, but with labels in the leaves.

How to define equality now?
More datatypes

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

Like before, but with labels in the leaves.

How to define equality now?

We need equality on `a`!
More datatypes

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

Like before, but with labels in the leaves.

How to define equality now?

We need equality on \( a \)!

```haskell
eqTree :: (a → a → Bool) → Tree a → Tree a → Bool
eqTree eqa (Leaf n₁) (Leaf n₂) = eqa n₁ n₂
eqTree eqa (Node x₁ y₁) (Node x₂ y₂) = eqTree eqa x₁ x₂ && eqTree eqa y₁ y₂
eqTree eqa _ _ _ = False
```
Type classes

Note how the definition of `eqTree` is perfectly suited for a type class instance:

```haskell
instance Eq a ⇒ Eq (Tree a) where
    (==) = eqTree (==)
```
Type classes

Note how the definition of `eqTree` is perfectly suited for a type class instance:

```haskell
instance Eq a ⇒ Eq (Tree a) where
  (==) = eqTree (==)
```

In fact, type classes are usually implemented as dictionaries, and an instance declaration is translated into a dictionary transformer.
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:

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

Let’s assume we already have:

```haskell
eqList :: (a → a → Bool) → [a] → [a] → Bool
```

How to define `eqRose`?
Yet another equality function

This is often called a **rose tree**:

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

Let’s assume we already have:

```haskell
eqList :: (a → a → Bool) → [a] → [a] → Bool
```

How to define **eqRose**?

```haskell
eqRose :: (a → a → Bool) → Rose a → Rose a → Bool
eqRose eqa (Fork x₁ xs₁) (Fork x₂ xs₂) =
  eqa x₁ x₂ && eqList (eqRose eqa) xs₁ xs₂
```

No fallback case needed because there is only one constructor.
More concepts

- Parameterization of types is reflected by parameterization of the functions.
- Application of parameterized types is reflected by application of the functions.
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 function; combine equality functions and use the parameter functions as needed.
The equality pattern

An informal description

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 function; combine equality functions and use the parameter functions as needed.

If we can describe it, **can we write a program to do it?**
Interlude: type isomorphisms
Isomorphism between types

Two types $A$ and $B$ are called isomorphic if we have functions

\[ f :: A \to B \]
\[ g :: B \to A \]

that are mutual inverses, i.e., if

\[ f \circ g \equiv \text{id} \]
\[ g \circ f \equiv \text{id} \]
List and Snoc-lists are isomorphic

\[
\textbf{data} \ \text{SnocList} \ a = \text{Lin} \ | \ \text{SnocList} \ a \to a
\]
**Example**

Lists and Snoc-lists are isomorphic

```haskell
data SnocList a = Lin | SnocList a :> a

listToSnocList :: [a] → SnocList a
listToSnocList [] = Lin
listToSnocList (x : xs) = listToSnocList xs :> x

snocListToList :: SnocList a → [a]
snocListToList Lin = []
snocListToList (xs :> x) = x : snocListToList xs

We can prove that these are inverses.
```
The idea of datatype-generic programming

- Represent a type \( A \) as an isomorphic type \( \text{Rep} \ A \).

If a limited number of type constructors is used to build \( \text{Rep} \ A \), 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. In fact, we do not even quite need an isomorphic type. For a type \( A \), we need a type \( \text{Rep} \ A \) and from

\[
A \rightarrow \text{Rep} \ A
\]

and

\[
\text{Rep} \ A \rightarrow A
\]

such that

\[
\text{to} \circ \text{from} \equiv \text{id}
\]

We call such a combination an embedding-projection pair.
The idea of datatype-generic programming

- Represent a type $A$ as an isomorphic type $\text{Rep } A$.
- If a limited number of type constructors is used to build $\text{Rep } A$, 

In fact, we do not even quite need an isomorphic type. For a type $A$, we need a type $\text{Rep } A$ and from $A \rightarrow \text{Rep } A$ and to $\text{Rep } A \rightarrow A$ such that to $\circ$ from $\equiv \text{id}$. We call such a combination an embedding-projection pair.
The idea of datatype-generic programming

- Represent a type $A$ as an isomorphic type $\text{Rep} \ A$.
- If a limited number of type constructors is used to build $\text{Rep} \ A$,
- then functions defined on each of these type constructors
The idea of datatype-generic programming

- Represent a type $A$ as an isomorphic type $\text{Rep } A$.
- If a limited number of type constructors is used to build $\text{Rep } A$,
- then functions defined on each of these type constructors
- can be lifted to work on the original type $A$. 
The idea of datatype-generic programming

- Represent a type \( A \) as an isomorphic type \( \text{Rep} \ A \).
- If a limited number of type constructors is used to build \( \text{Rep} \ A \),
- 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.
The idea of datatype-generic programming

- Represent a type $A$ as an isomorphic type $\text{Rep} \ A$.
- If a limited number of type constructors is used to build $\text{Rep} \ A$,
- 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.

In fact, we do not even quite need an isomorphic type.

For a type $A$, we need a type $\text{Rep} \ A$ and $\text{from} :: A \rightarrow \text{Rep} \ A$
and $\text{to} :: \text{Rep} \ A \rightarrow A$ such that

$\text{to} \circ \text{from} \equiv \text{id}$

We call such a combination an embedding-projection pair.
Choice between constructors

Which type best encodes choice between constructors?

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

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

Choice between three things:

```latex
type Either 3 a b c = Either a (Either b c)
```
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 a
```
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.

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

Choice between three things:

```haskell
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:

\[
type \text{ Triple } a \ b \ c = (a, (b, c))
\]
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
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:

\[
\begin{align*}
\textbf{data} & \quad \mathbb{U} = \mathbb{U} \\
\textbf{data} & \quad a :: b = \mathbb{L} a \mid \mathbb{R} b \\
\textbf{data} & \quad a ::* b = a ::* b
\end{align*}
\]

We can now get started:

\[
\begin{align*}
\textbf{data} & \quad \textbf{Bool} = \text{False} \mid \text{True}
\end{align*}
\]

How do we represent \texttt{Bool}?

\[
\begin{align*}
\textbf{type} & \quad \text{RepBool} = \mathbb{U} :: \mathbb{U}
\end{align*}
\]
A class for representable types

class Generic a where
  type Rep a
  from :: a → Rep a
  to :: Rep a → a
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 **associated type**. GHC allows us to define datatypes and type synonyms within classes, depending on the class parameter(s).
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 associated type. GHC allows us to define datatypes and type synonyms within classes, depending on the class parameter(s).

This is 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 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
```

**Question**

Are `Bool` and `Rep Bool` isomorphic?
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 that the representation of recursive types mentions the original types – if needed, we can apply the transformation multiple times.

Note further that we do not require `Generic a`.
Representable lists

```haskell
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 that the representation of recursive types mentions the original types – if needed, we can apply the transformation multiple times.
Representable lists

```haskell
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 that the representation of recursive types mentions the original types – if needed, we can apply the transformation multiple times.

Note further that we do not require `Generic a`. 
Representable trees

```haskell
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

For some types, it does not make sense to define a structural representation – for such types, we will have to define generic functions directly.

```haskell
instance Generic Int where
type Rep Int = Int
  from = id
  to   = id
```
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.
Equality on sums

eqSum :: (a → a → Bool) →
      (b → b → Bool) →
      a :+: b → a :+: b → Bool

eqSum eqa eqb (L a₁) (L a₂) = eqa a₁ a₂
eqSum eqa eqb (R b₁) (R b₂) = eqb b₁ b₂
eqSum eqa eqb _ _ = False
Equality on products

eqProd :: (a → a → Bool) →
   (b → b → Bool) →
   a :: b a :: b → Bool

eqProd eqa eqb (a₁ :: b₁) (a₂ :: b₂) =
eqa a₁ a₂ && eqb b₁ b₂
Equality on units

eqUnit :: U → U → Bool

eqUnit U U = True
What now?
A class for generic equality

```haskell
class GEq a where
  geq :: a → a → Bool
```

Instances for primitive types:

```haskell
instance GEq Int where
  geq = eqInt
```
A class for generic equality

class GEq a where
  geq :: a → a → Bool

instance (GEq a, GEq b) ⇒ GEq (a :+: b) where
  geq = eqSum geq geq

instance (GEq a, GEq b) ⇒ GEq (a :*: b) where
  geq = eqProd geq geq

instance GEq U where
  geq = eqUnit
A class for generic equality

class GEq a where
geq :: a → a → Bool

instance (GEq a, GEq b) ⇒ GEq (a :+: b) where
  geq = eqSum geq geq

instance (GEq a, GEq b) ⇒ GEq (a :+: b) where
  geq = eqProd geq geq

instance GEq U where
  geq = eqUnit

Instances for primitive types:

instance GEq Int where
  geq = eqInt
Dispatching to the representation type

\[
eq :: (\text{Generic } a, \text{G\text{Eq }} (\text{Rep } a)) \Rightarrow a \rightarrow a \rightarrow \text{Bool}
\]

\[
eq x y = \text{geq (from } x) \text{ (from } y)
\]
Dispatching to the representation type

\[\text{eq} :: (\text{Generic } a, \text{GEq (Rep } a)) \Rightarrow a \to a \to \text{Bool}\]

\[\text{eq } x \; y = \text{geq (from } x) \; (\text{from } y)\]

Defining generic instances is now trivial:

\[
\text{instance GEq Bool where}\n\quad \text{geq } = \text{eq}
\]

\[
\text{instance GEq } a \Rightarrow \text{GEq } [a] \quad \text{where}\n\quad \text{geq } = \text{eq}
\]

\[
\text{instance GEq } a \Rightarrow \text{GEq (Tree } a) \quad \text{where}\n\quad \text{geq } = \text{eq}
\]

\[
\text{instance GEq } a \Rightarrow \text{GEq (Rose } a) \quad \text{where}\n\quad \text{geq } = \text{eq}
\]
Dispatching to the representation type

\[
\text{eq} :: (\text{Generic } a, \text{GEq } (\text{Rep } a)) \Rightarrow a \rightarrow a \rightarrow \text{Bool} \\
\text{eq } x \ y = \text{geq } (\text{from } x) (\text{from } y)
\]

Or with the DefaultSignatures language extension:

\[
\textbf{class } \text{GEq } a \textbf{ where} \\
\quad \text{geq} :: a \rightarrow a \rightarrow \text{Bool} \\
\quad \textbf{default} \ \text{geq} :: (\text{Generic } a, \text{GEq } (\text{Rep } a)) \Rightarrow a \rightarrow a \rightarrow \text{Bool} \\
\quad \text{geq } = \text{eq} \\
\textbf{instance } \text{GEq } \text{Bool} \\
\textbf{instance } \text{GEq } a \Rightarrow \text{GEq } \text{[a]} \\
\textbf{instance } \text{GEq } a \Rightarrow \text{GEq } \text{(Tree } a) \\
\textbf{instance } \text{GEq } a \Rightarrow \text{GEq } \text{(Rose } a)
\]
Have we won
or
have we lost?
<table>
<thead>
<tr>
<th>Question</th>
</tr>
</thead>
<tbody>
<tr>
<td>Haven’t we just replaced some tedious work (defining equality for a type) by some other tedious work (defining a representation for a type)?</td>
</tr>
</tbody>
</table>
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`.
Other generic functions
We want to define

```haskell
data Bit = O | I

encode :: (Generic a, GEncode (Rep a)) \Rightarrow a \rightarrow [Bit]
decode :: (Generic a, GDecode (Rep a)) \Rightarrow BitParser a

type BitParser a = [Bit] \rightarrow Maybe (a, [Bit])
```

such that encoding and then decoding yields the original value.
What about constructor names?

Seems that the representation we have does not provide constructor name info.
What about constructor names?

Seems that the representation we have does not provide constructor name info.

So let us extend the representation:

```haskell
data C c a = C a
```

Note that `c` does not appear on the right hand side.
What about constructor names?

Seems that the representation we have does not provide constructor name info.

So let us extend the representation:

```haskell
data C c a = C a
```

Note that `c` does not appear on the right hand side.

But `c` is supposed to be in this class:

```haskell
class Constructor c where
    conName :: t c a \to String
```
Trees with constructors

data TreeLeaf
instance Constructor TreeLeaf where
  conName _ = "Leaf"

data TreeNode
instance Constructor TreeNode where
  conName _ = "Node"
Trees with constructors

```haskell
data TreeLeaf
instance Constructor TreeLeaf where
  conName _ = "Leaf"

data TreeNode
instance Constructor TreeNode where
  conName _ = "Node"

instance Generic (Tree a) where
  type Rep (Tree a) = C TreeLeaf a :+: C TreeNode (Tree a :+: Tree a)

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

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

Well-Typed
Defining functions on constructors

```haskell
instance (GShow a, Constructor c) ⇒ GShow (C c a) where
  gshow c@(C a)
    | null args = conName c
    | otherwise = "(" ++ conName c ++ " " ++ args ++ ")"
where args = gshow a
```

Defining functions on constructors

```haskell
instance (GShow a, Constructor c) \Rightarrow GShow (C c a) where
  gshow c@(C a)
    | null args = conName c
    | otherwise = "(" ++ conName c ++ " " ++ args ++ ")"
where args = gshow a

instance (GEq a) \Rightarrow GEq (C c a) where
  geq (C x) (C y) = geq x y
```
A library for generic programming

What we have discussed so far is a slightly simplified form of a library available on Hackage called `generic-deriving`.

- **Generic** instances for most prelude types.
- Since ghc-7.2.1, `DeriveGeneric` language extension to derive `Generic` class automatically.
- A number of example generic functions.
- Additional markers in the representation to distinguish positions of type variables from other fields.
- Even closer to what we discussed is the `instant-generics` library, but it offers “only” Template Haskell support for generating the representations.
Is this the only way?
Many design choices

No!

There are lots of approaches (too many) to generic programming in Haskell.
Many design choices

No!

There are lots of approaches (too many) to generic programming in Haskell.

- The main question is exactly how we represent the datatypes – we have already seen what kind of freedom we have.
- The view dictates which datatypes we can represent easily, and which generic functions can be defined.
Other notable approaches
Constructor-based views

The **Scrap your boilerplate** library takes a very simple view on values:

\[ C \ x_1 \ldots x_n \]

Every value in a datatype is a constructor applied to a number of arguments.
Other notable approaches
Constructor-based views

The **Scrap your boilerplate** library takes a very simple view on values:

\[ C \ x_1 \ldots \ x_n \]

Every value in a datatype is a constructor applied to a number of arguments.

Using SYB, it is easy to define traversals and queries.
Other notable approaches
Children-based views

The **Uniplate** library is a simplification of SYB that just shows how in a recursive structure we can get to the children, and back from the children to the structure.

\[
\text{uniplate} :: \text{Uniplate } a \Rightarrow a \rightarrow ([a], [a] \rightarrow a)
\]
Other notable approaches

Children-based views

The Uniplate library is a simplification of SYB that just shows how in a recursive structure we can get to the children, and back from the children to the structure.

\[
\text{uniplate} :: \text{Uniplate } a \Rightarrow a \rightarrow ([a], [a] \rightarrow a)
\]

While a bit less powerful than SYB, this is one of the simplest Generic Programming libraries around, and allows to define the same kind of traversals and queries as SYB.
The *regular* and *multirec* libraries work with representations that abstract from the recursion by means of a fixed-point combinator, in addition to revealing the sums-of-product structure.
Other notable approaches

Fixed-point views

The **regular** and **multirec** libraries work with representations that abstract from the recursion by means of a fixed-point combinator, in addition to revealing the sums-of-product structure

```haskell
data Fix f = In (f (Fix f))
out (In f) = f
```
Other notable approaches

Fixed-point views

The **regular** and **multirec** libraries work with representations that abstract from the recursion by means of a fixed-point combinator, in addition to revealing the sums-of-product structure.

```
data Fix f = ln (f (Fix f))
out (ln f) = f
```

Using a fixed-point view, we can more easily capture functions that make use of the recursive structure of a type, such as folds and unfolds (catamorphisms and anamorphisms).
Dependently typed programming languages such as Agda allow types to depend on terms. For example,

\[ \text{Vec Int 5} \]

could be a vector of integers of length 5.
Dependently typed programming languages such as Agda allow types to depend on terms. For example,

```
Vec Int 5
```

could be a vector of integers of length 5.

We can also compute types from values, then. So we can define grammars of types as normal datatypes, and interpret them as the types they describe.
Dependently typed programming languages such as Agda allow types to depend on terms. For example,

Vec Int 5

could be a vector of integers of length 5.

We can also compute types from values, then. So we can define grammars of types as normal datatypes, and interpret them as the types they describe.

Makes it easy to play with many different views (universes).
Other topics

There is more than we can cover in this lecture:

▶ Looking at all the other GP approaches closely.
▶ Comparison with template meta-programming.
▶ Efficiency of generic functions.
▶ Type-indexed types.
▶ ...