Datatype-generic Programming in Haskell

An introduction

Andres Löh

Well-Typed LLP

30 May 2011

Haven't you ever wondered how **deriving** works?

Equality on binary trees

$\mathbf{data}\ \mathsf{T} = \mathsf{L} \mid \mathsf{N}\ \mathsf{T}\ \mathsf{T}$

Let's try ourselves:

Equality on binary trees

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

Let's try ourselves:

$$
\begin{array}{llll}
\mathrm{eqT} :: \mathrm{T} \to \mathrm{T} \to \mathrm{Bool} & & & \\
\mathrm{eqT}\ \ \mathrm{L} & \mathrm{L} & = & \mathrm{True} \\
\mathrm{eqT}\ (\mathrm{N}\ x_1\ y_1) & (\mathrm{N}\ x_2\ y_2) & = & \mathrm{eqT}\ x_1\ x_2 \wedge \mathrm{eqT}\ y_1\ y_2 \\
\mathrm{eqT}\ \ \_ & \_ & = & \mathrm{False}
\end{array}
$$

Equality on binary trees

$$\textbf{data}\ T = L \mid N\ T\ T$$

Let's try ourselves:

$$
\begin{aligned}
&\mathrm{eqT} :: T \to T \to \mathrm{Bool} \\
&\mathrm{eqT}\ \ L \qquad\qquad\quad L \qquad\qquad\ = \mathrm{True} \\
&\mathrm{eqT}\ (N\ x_1\ y_1)\ (N\ x_2\ y_2) = \mathrm{eqT}\ x_1\ x_2 \wedge \mathrm{eqT}\ y_1\ y_2 \\
&\mathrm{eqT}\ \ \_ \qquad\qquad\quad \_ \qquad\qquad\ = \mathrm{False}
\end{aligned}
$$

Easy enough, let's try another ...

Equality on another type

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

Equality on another type

$$\textbf{data}\ \mathrm{Choice} = \mathrm{I\ Int} \mid \mathrm{C\ Char} \mid \mathrm{B\ Choice\ Bool} \mid \mathrm{S\ Choice}$$

$$\begin{array}{llll}
\mathrm{eqChoice} :: \mathrm{Choice} \to \mathrm{Choice} \to \mathrm{Bool} & & & \\
\mathrm{eqChoice}\ (\mathrm{I}\ n_1) & (\mathrm{I}\ n_2) & = & \mathrm{eqInt}\ n_1\ n_2 \\
\mathrm{eqChoice}\ (\mathrm{C}\ c_1) & (\mathrm{C}\ c_2) & = & \mathrm{eqChar}\ c_1\ c_2 \\
\mathrm{eqChoice}\ (\mathrm{B}\ x_1\ b_1) & (\mathrm{B}\ x_2\ b_2) & = & \mathrm{eqChoice}\ x_1\ x_2 \wedge \mathrm{eqBool}\ b_1\ b_2 \\
\mathrm{eqChoice}\ _ & _ & = & \mathrm{False}
\end{array}$$

Equality on another type

$$\textbf{data}\ \text{Choice} = \text{I Int} \mid \text{C Char} \mid \text{B Choice Bool} \mid \text{S Choice}$$

$$\begin{array}{l}
\text{eqChoice} :: \text{Choice} \to \text{Choice} \to \text{Bool} \\
\text{eqChoice}\ (\text{I}\ n_1)\ (\text{I}\ n_2) = \text{eqInt}\ n_1\ n_2 \\
\text{eqChoice}\ (\text{C}\ c_1)\ (\text{C}\ c_2) = \text{eqChar}\ c_1\ c_2 \\
\text{eqChoice}\ (\text{B}\ x_1\ b_1)\ (\text{B}\ x_2\ b_2) = \text{eqChoice}\ x_1\ x_2 \wedge \text{eqBool}\ b_1\ b_2 \\
\text{eqChoice}\ _\ _ = \text{False}
\end{array}$$

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

```
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

```
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

$$\textbf{data}\ \mathrm{Tree}\ a = \mathrm{Leaf}\ a \mid \mathrm{Node}\ (\mathrm{Tree}\ a)\ (\mathrm{Tree}\ a)$$

Like before, but with labels in the leaves.

How to define equality now?

We need equality on a!

$$\begin{array}{llll}
\mathrm{eqTree} :: (a \to a \to \mathrm{Bool}) \to \mathrm{Tree}\ a \to \mathrm{Tree}\ a \to \mathrm{Bool} & & & \\
\mathrm{eqTree}\ \mathrm{eqa}\ (\mathrm{Leaf}\ n_1\ ) & (\mathrm{Leaf}\ n_2\ ) & = & \mathrm{eqa}\ n_1\ n_2 \\
\mathrm{eqTree}\ \mathrm{eqa}\ (\mathrm{Node}\ x_1\ y_1) & (\mathrm{Node}\ x_2\ y_2) & = & \mathrm{eqTree}\ \mathrm{eqa}\ x_1\ x_2 \wedge \\
 & & & \mathrm{eqTree}\ \mathrm{eqa}\ y_1\ y_2 \\
\mathrm{eqTree}\ \mathrm{eqa}\ _ & _ & = & \mathrm{False}
\end{array}$$

Type classes

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

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

Type classes

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

```
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**:

$\textbf{data}\ \mathrm{Rose}\ a = \mathrm{Fork}\ a\ [\mathrm{Rose}\ a]$

Let's assume we already have:

$\mathrm{eqList} :: (a \to a \to \mathrm{Bool}) \to [a] \to [a] \to \mathrm{Bool}$

How to define eqRose ?

Yet another equality function

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

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

Let's assume we already have:

$$\text{eqList} :: (a \to a \to \text{Bool}) \to [a] \to [a] \to \text{Bool}$$

How to define `eqRose` ?

$$\begin{aligned}&\text{eqRose} :: (a \to a \to \text{Bool}) \to \text{Rose } a \to \text{Rose } a \to \text{Bool}\\&\text{eqRose eqa (Fork } x_1\ xs_1)\ (\text{Fork } x_2\ xs_2) =\\&\qquad\qquad \text{eqa } x_1\ x_2 \wedge \text{eqList (eqRose eqa) } xs_1\ xs_2\end{aligned}$$

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.

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 $(\wedge)$,
- 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 $(\wedge)$,
- 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

$$
\begin{aligned}
f &:: A \to B \\
g &:: B \to A
\end{aligned}
$$

that are mutual **inverses**, i.e., if

$$
\begin{aligned}
f \circ g &\equiv id \\
g \circ f &\equiv id
\end{aligned}
$$

Example

Lists and Snoc-lists are isomorphic

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

Example

Lists and Snoc-lists are isomorphic

```
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

If we can represent a type as an isomorphic type that is composed out of a limited number of type constructors, then we can define a function on each of the type constructors and gain a function that works on the original type – and in fact on any representable type.

The idea of datatype-generic programming

If we can represent a type as an isomorphic type that is composed out of a limited number of type constructors, then we can define a function on each of the type constructors and gain a function that works on the original type – and in fact on any representable type.

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

For a type A, we need a type B and $\mathsf{from} :: \mathsf{A} \to \mathsf{B}$ and $\mathsf{to} :: \mathsf{B} \to \mathsf{A}$ such that

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

We call such a combination an **embedding-projection pair**.

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 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.

data Either a b = Left a | Right a

Choice between three things:

type Either_3 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.

$$\textbf{data}\ (a, b) = (a, b)$$

Combining constructor fields

Which type best encodes combining fields?

Again, let's just consider two of them.

$$\textbf{data}\ (\mathrm{a}, \mathrm{b}) = (\mathrm{a}, \mathrm{b})$$

Combining three fields:

$$\textbf{type}\ \mathrm{Triple\ a\ b\ c} = (\mathrm{a}, (\mathrm{b}, \mathrm{c}))$$

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

```
class Representable a where
  type Rep a
  from :: a → Rep a
  to   :: Rep a → a
```

A class for representable types

```
class Representable 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).

Representable Booleans

```
instance Representable 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 Representable 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 Representable [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
```

Well-Typed

Representable lists

```
instance Representable [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

```
instance Representable [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 `Representable a`.

Representable trees

```
instance Representable (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

```
instance Representable (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.

```
instance Representable Int where
  type Rep Int = Int
  from = id
  to   = id
```

Back to equality

Intermediate summary

- We have defined class `Representable` 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 a1) (R a2) = eqb a1 a2
eqSum eqa eqb _      _      = False
```

Equality on products

$$\begin{aligned}
\mathsf{eqProd} :: (\; & a \qquad\quad \to a \qquad\quad \to \mathsf{Bool}) \to \\
(\; & \qquad\; b \to \qquad\quad b \to \mathsf{Bool}) \to \\
& a :\!*\!: b \to a :\!*\!: b \to \mathsf{Bool}
\end{aligned}$$

$$\begin{aligned}
& \mathsf{eqProd}\ \mathsf{eqa}\ \mathsf{eqb}\ (a_1 :\!*\!: b_1)\ (a_2 :\!*\!: b_2) = \\
& \quad \mathsf{eqa}\ a_1\ a_2 \wedge \mathsf{eqb}\ b_1\ b_2
\end{aligned}$$

Equality on units

```
eqUnit :: U → U → Bool
eqUnit U U = True
```

What now?

A class for generic equality

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

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

$$\mathsf{eq} :: (\mathsf{Representable}\ a, \mathsf{GEq}\ (\mathsf{Rep}\ a)) \Rightarrow a \rightarrow a \rightarrow \mathsf{Bool}$$
$$\mathsf{eq}\ x\ y = \mathsf{geq}\ (\mathsf{from}\ x)\ (\mathsf{from}\ y)$$

Dispatching to the representation type

$$
\begin{array}{l}
\mathsf{eq} :: (\mathsf{Representable}\ \mathsf{a}, \mathsf{GEq}\ (\mathsf{Rep}\ \mathsf{a})) \Rightarrow \mathsf{a} \to \mathsf{a} \to \mathsf{Bool} \\
\mathsf{eq}\ \mathsf{x}\ \mathsf{y} = \mathsf{geq}\ (\mathsf{from}\ \mathsf{x})\ (\mathsf{from}\ \mathsf{y})
\end{array}
$$

Defining generic instances is now trivial:

$$
\begin{array}{llll}
\textbf{instance} & & \mathsf{GEq}\ \mathsf{Bool} & \textbf{where} \\
\quad \mathsf{geq} = \mathsf{eq} & & & \\
\textbf{instance} & \mathsf{GEq}\ \mathsf{a} \Rightarrow & \mathsf{GEq}\ [\mathsf{a}] & \textbf{where} \\
\quad \mathsf{geq} = \mathsf{eq} & & & \\
\textbf{instance} & \mathsf{GEq}\ \mathsf{a} \Rightarrow & \mathsf{GEq}\ (\mathsf{Tree}\ \mathsf{a}) & \textbf{where} \\
\quad \mathsf{geq} = \mathsf{eq} & & & \\
\textbf{instance} & \mathsf{GEq}\ \mathsf{a} \Rightarrow & \mathsf{GEq}\ (\mathsf{Rose}\ \mathsf{a}) & \textbf{where} \\
\quad \mathsf{geq} = \mathsf{eq} & & &
\end{array}
$$

Have we won
or
have we lost?

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).

Other generic functions

Coding and decoding

We want to define

$$\textbf{data}\ \mathsf{Bit} = \mathsf{O} \mid \mathsf{I}$$

$$\mathsf{encode} :: (\mathsf{Representable}\ \mathsf{a}, \mathsf{GEncode}\ (\mathsf{Rep}\ \mathsf{a})) \Rightarrow \mathsf{a} \rightarrow [\mathsf{Bit}]$$
$$\mathsf{decode} :: (\mathsf{Representable}\ \mathsf{a}, \mathsf{GDecode}\ (\mathsf{Rep}\ \mathsf{a})) \Rightarrow \mathsf{BitParser}\ \mathsf{a}$$

$$\textbf{type}\ \mathsf{BitParser}\ \mathsf{a} = [\mathsf{Bit}] \rightarrow \mathsf{Maybe}\ (\mathsf{a}, [\mathsf{Bit}])$$

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:

$$\textbf{data}\ \mathrm{C}\ \mathrm{c}\ \mathrm{a} = \mathrm{C}\ \mathrm{a}$$

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:

```
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:

```
class Constructor c where
  conName :: t c a → String
```

Trees with constructors

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

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

Trees with constructors

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

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

```
instance Representable (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
```

Defining functions on constructors

```
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

```
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
```

```
instance (GEq a) ⇒ 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 available on Hackage as a library called **instant-generics**.

- `Representable` instances for most prelude types.
- Template Haskell generation of `Representable` instances.
- A number of example generic functions.
- Additional markers in the representation to distinguish positions of type variables from other fields.

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:

$$C\ x_1\ \dots\ x_n$$

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.

$$\mathsf{uniplate} :: \mathsf{Uniplate\ a} \Rightarrow \mathsf{a} \to ([\mathsf{a}], [\mathsf{a}] \to \mathsf{a})$$

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.

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

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.

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

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 = 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 = In (f (Fix f))
out (In 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).

GHC implementation

An approach that is quite similar to instant-generics has just been implemented directly in GHC, and will be available in the upcoming 7.2.1 release together with the Hackage library **generic-deriving**.

GHC implementation

An approach that is quite similar to instant-generics has just been implemented directly in GHC, and will be available in the upcoming 7.2.1 release together with the Hackage library **generic-deriving**.

With this approach, GHC can automatically (without using TH) generate the representations for you.

Outlook: dependent types

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.

Outlook: dependent types

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.

Outlook: dependent types

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.
- ...

Questions?