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

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

Let’s try ourselves:
Equality on binary trees

\[ \text{data } T = L \mid N \ T \ T \]

Let's try ourselves:

\[ \text{eqT :: } T \to T \to \text{Bool} \]
\[ \text{eqT } L \quad L = \text{True} \]
\[ \text{eqT } (N \ x_1 \ y_1) \ (N \ x_2 \ y_2) = \text{eqT } x_1 \ x_2 \land \text{eqT } y_1 \ y_2 \]
\[ \text{eqT } \_ \ \_ = \text{False} \]
data \( T = L \mid N T T \)

Let’s try ourselves:

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

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

\[
eq_T (N x_1 y_1) (N x_2 y_2) = \ eq_T x_1 x_2 \land \ eq_T y_1 y_2
\]

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

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

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

Do you see a pattern?
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 _ _ = False
```
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 _ _ = 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

```
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
```
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:

```
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**: 

```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.
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 \(\text{eqRose}\)?

```haskell
eqRose :: (a → a → Bool) → Rose a → Rose a → Bool
eqRose eqa (Fork x_1 xs_1) (Fork x_2 xs_2) =
    eqa x_1 x_2 ∧ eqList (eqRose eqa) xs_1 xs_2
```

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 \( \text{False} \),
- for each of the other cases, compare the constructor fields pair-wise and combine them using \( \land \),
- 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

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

data SnocList a = Lin | SnocList a :> a
Example
Lists and Snoc-lists are isomorphic

\[
\text{data SnocList } a = \text{Lin} | \text{SnocList } a :> a
\]

\[
\text{listToSnocList} :: [a] \rightarrow \text{SnocList } a
\]

\[
\text{listToSnocList} [ ] = \text{Lin}
\]

\[
\text{listToSnocList} (x : xs) = \text{listToSnocList} xs :> x
\]

\[
\text{snocListToList} :: \text{SnocList } a \rightarrow [a]
\]

\[
\text{snocListToList} \text{Lin} = [ ]
\]

\[
\text{snocListToList} (xs :> x) = x : \text{snocListToList} xs
\]

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 from :: \( A \rightarrow B \) and to :: \( B \rightarrow A \) 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?

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)
```
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 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.

\[
\text{data } (a, b) = (a, b)
\]

Combining three fields:

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

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

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

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

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

We can now get started:

```haskell
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:

\[
\text{data } U = U \\
\text{data } a :+ : b = L a | R b \\
\text{data } a :* : b = a :* : b
\]

We can now get started:

\[
\text{data } \text{Bool} = \text{False} | \text{True}
\]

How do we represent \text{Bool}?

\[
\text{type } \text{RepBool} = U :+ : U
\]
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 \texttt{Rep} is an \textit{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

```haskell
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

```haskell
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 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

```haskell
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

\[
\text{instance } \text{Representable } (\text{Tree } a) \quad \text{where}
\]
\[
\text{type } \text{Rep } (\text{Tree } a) = a :+ : (\text{Tree } a :*: \text{Tree } a)
\]
\[
\text{from } (\text{Leaf } n) = L n
\]
\[
\text{from } (\text{Node } x y) = R (x :*: y)
\]
\[
\text{to } (L n) = \text{Leaf } n
\]
\[
\text{to } (R (x :*: y)) = \text{Node } x y
\]
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.

```hs
instance Representable Int where
  type Rep Int = Int
  from = id
  to  = id
```
Back to equality
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

\[
\text{eqSum} :: (a \to a \to \text{Bool}) \to \\
(\quad b \to b \to \text{Bool}) \to \\
a :+ b \to a :+ b \to \text{Bool}
\]

\[
\text{eqSum }\text{eqa eqb} (L \ a_1) (L \ a_2) = \text{eqa }a_1 \ a_2 \\
\text{eqSum }\text{eqa eqb} (R \ a_1) (R \ a_2) = \text{eqb }a_1 \ a_2 \\
\text{eqSum }\text{eqa eqb} \quad \quad \quad = \text{False}
\]
Equality on products

\[
\text{eqProd} :: (a \rightarrow a \rightarrow \text{Bool}) \rightarrow \\
\quad (b \rightarrow b \rightarrow \text{Bool}) \rightarrow \\
\quad a : \ast : b \rightarrow a : \ast : b \rightarrow \text{Bool}
\]

\[
\text{eqProd } \text{eqa } \text{eqb } (a_1 : \ast : b_1) (a_2 : \ast : b_2) = \\
\quad \text{eqa } a_1 \ a_2 \land \text{eqb } b_1 \ b_2
\]
Equality on units

\[ \text{eqUnit} :: U \rightarrow U \rightarrow \text{Bool} \]
\[ \text{eqUnit} U U = \text{True} \]
What now?
A class for generic equality

```haskell
class GEq a where
  geq :: a → a → Bool
```
A class for generic equality

```haskell
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 :: (Representable a, GEq (Rep a)) ⇒ a → a → Bool
eq x y = geq (from x) (from y)
Dispatching to the representation type

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

Defining generic instances is now trivial:

\[
\text{instance } \mathtt{GEq} \text{ Bool } \text{ where }
\]
\[
\quad \mathtt{geq} = \mathtt{eq}
\]
\[
\text{instance } \mathtt{GEq} a \Rightarrow \mathtt{GEq} \ [a] \text{ where }
\]
\[
\quad \mathtt{geq} = \mathtt{eq}
\]
\[
\text{instance } \mathtt{GEq} a \Rightarrow \mathtt{GEq} \ (\text{Tree } a) \text{ where }
\]
\[
\quad \mathtt{geq} = \mathtt{eq}
\]
\[
\text{instance } \mathtt{GEq} a \Rightarrow \mathtt{GEq} \ (\text{Rose } a) \text{ where }
\]
\[
\quad \mathtt{geq} = \mathtt{eq}
\]
Have we won
or
have we lost?
Haven’t we just replaced some tedious work (defining equality for a type) by some other tedious work (defining a representation for a type)?

- 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).
## 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

```haskell
data Bit = O | I
```

```haskell
encode :: (Representable a, GEncode (Rep a)) ⇒ a → [Bit]
decode :: (Representable a, GDecode (Rep a)) ⇒ BitParser a
type BitParser a = [Bit] → 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 → 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 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

```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
```
instance (GShow a, Constructor c) \Rightarrow GShow (C c a) \textbf{where}
\begin{align*}
gshow c@(C a) & \mid \text{null args} = \text{conName } c \\
& \mid \text{otherwise} = "(" ++ \text{conName } c ++ " " ++ \text{args} ++ ")" \\
\textbf{where} \ \text{args} = \text{gshow } a
\end{align*}

instance (GEq a) \Rightarrow GEq (C c a) \textbf{where}
\begin{align*}
geq (C x) (C y) &= \text{geq } x \ y
\end{align*}
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:

\[ C \ x_1 \ldots \ x_n \]

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:

\[ C \ x_1 \ ... \ x_n \]

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.

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

```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

```haskell
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).
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`.
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.
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,

\text{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).
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?