Generic Deriving in GHC 7.2

Sean Leather

Utrecht University

8 November 2011

deriving

You're familiar with the **deriving** mechanism in Haskell:

```
data Suit = Club | Diamond | Heart | Spade
  deriving (Eq, Ord, Enum, Bounded, Read {-, Show -} )
```

Not **deriving**

instance Show Suit **where**
    show Club   = "♣"
    show Diamond = "♢"
    show Heart   = "♡"
    show Spade   = "♠"

Problems with **deriving**

- Specification is largely informal
  - Difficult to validate an implementation against the specification
- Restricted to Eq, Ord, Enum, Bounded, Read, and Show
  - Cannot reuse the **deriving** mechanism for other purposes
- Mechanism is built into the compiler
  - More work to maintain compiler than library
  - Duplicated functionality for each derived class
  - Difficult to share implementations between compilers

Alternatives to **deriving**

Use generic programming libraries!
(of course)

Problems with GP

- Not built into compiler
  - Requires boilerplate or Template Haskell to use library
- Requires additional libraries
  - Nonstandard libraries are often not used
- Some libraries are difficult to understand/use
  - Complicated types and functions

Hello, Generic Deriving (GD)

- A library: `generic-deriving` on Hackage
- An implementation of **deriving** Eq, Ord, etc.
- Language extensions in GHC 7.2
  - Default implementation for type class methods
  - **deriving** the instance for `generic-deriving`

How Does GD Work?

class Eq a **where**
  $(\equiv) :: a \rightarrow a \rightarrow \text{Bool}$

How Does GD Work?

```
{-# LANGUAGE DefaultSignatures #-}
```

```
class Eq a where
  (≡) :: a → a → Bool
  default (≡) :: (Generic a, GEq (Rep a)) ⇒ a → a → Bool
  x ≡ y = geq (from x) (from y)
```

How Does GD Work?

```
{-# LANGUAGE DeriveGeneric #-}
```

```
import Generics.Deriving
```

```
data Exp = Var String
         | Lam String Exp
         | App Exp Exp
         deriving Generic
```

```
instance Eq Exp
```

```
test = Lam "x" (Var "x") ≡ Lam "y" (Var "y")
```

Behind the Scenes

- What are `Generic`, `Rep`, and `GEq`?
- How does **deriving** `Generic` allow us to use the `Eq` class?

Generic

```
class Generic a where
  type Rep a :: * → *
  from       :: a → Rep a x
  to         :: Rep a x → a
```

Isomorphisms

Haskell Type	$\cong$	**Representation**
()		()
(a, b)		(a, b)
Either a b		Either a b
Maybe a		Either () a
[a]		Either () (a, [a])

```
data Exp = Var String
         | Lam String Exp
         | App Exp    Exp
```

$\cong$

```
 Either  String
(Either (String, Exp)
        (Exp   , Exp))
```

Representation Types

Unit: `()`

$$\textbf{data}\ U_1\ p = U_1$$

Representation Types

Binary product (i.e. pair): (a, b)

data $(\mathrm{f} :\!\times\!: \mathrm{g})\ \mathrm{p} = \mathrm{f}\ \mathrm{p} :\!\times\!: \mathrm{g}\ \mathrm{p}$

Representation Types

Binary sum (i.e. coproduct, alternatives): Either a b

$\mathbf{data}\ (\mathsf{f} :+: \mathsf{g})\ \mathsf{p} = \mathsf{L}_1\ \{\mathsf{unL}_1 :: \mathsf{f}\ \mathsf{p}\} \mid \mathsf{R}_1\ \{\mathsf{unR}_1 :: \mathsf{g}\ \mathsf{p}\}$

Representation Types

Constant types: primitives, other types without representation

newtype K_1 i a p $= \mathsf{K}_1$ $\{$unK_1 :: a$\}$

Representation Types

Metadata: constructor and datatype names, associativity, fixity

newtype $\mathsf{M}_1\ \mathsf{i}\ \mathsf{c}\ \mathsf{f}\ \mathsf{p} = \mathsf{M}_1\ \{\mathsf{unM}_1 :: \mathsf{f}\ \mathsf{p}\}$

Representation Type Synonyms

Parameter type

type $\mathsf{Par}_0 = \mathsf{K}_1\ \mathsf{P}$
data P

Recursive type

type $\mathsf{Rec}_0 = \mathsf{K}_1\ \mathsf{R}$
data R

Note that these aren't that useful for most generic functions.

Datatype tag

type $\mathsf{D}_1 = \mathsf{M}_1\ \mathsf{D}$
data D

Constructor tag

type $\mathsf{C}_1 = \mathsf{M}_1\ \mathsf{C}$
data C

Selector (label) tag

type $\mathsf{S}_1 = \mathsf{M}_1\ \mathsf{S}$
data S

Simplified Representation of Lists

class Generic a **where**
  type Rep a :: $* \to *$
  from :: a $\to$ Rep a x
  to :: Rep a x $\to$ a

instance Generic [a] **where**
  type Rep [a] = U_1 :+: Par_0 a :×: Rec_0 [a]
  from [] = L_1 U_1
  from (x : xs) = R_1 (K_1 x :×: K_1 xs)
  to (L_1 U_1) = []
  to (R_1 (K_1 x :×: K_1 xs)) = x : xs

"Full" Representation of Lists

$$
\begin{array}{l}
\textbf{instance}\ \mathsf{Generic}\ [\mathsf{a}]\ \textbf{where} \\
\quad \textbf{type}\ \mathsf{Rep}\ [\mathsf{a}] = \mathsf{D_1}\ \mathsf{DL}\ (\mathsf{C_1}\ \mathsf{CN}\ (\mathsf{S_1}\ \mathsf{SU}\ (\mathsf{U_1}))\ {:+:} \\
\qquad\qquad\qquad\qquad\qquad \mathsf{C_1}\ \mathsf{CC}\ (\mathsf{S_1}\ \mathsf{SP}\ (\mathsf{Par_0}\ \mathsf{a})\ {:\times:}\ \mathsf{S_1}\ \mathsf{SR}\ (\mathsf{Rec_0}\ [\mathsf{a}])))) \\
\quad \mathsf{from}\ [\,] \qquad\ \ = \mathsf{M_1}\ (\mathsf{L_1}\ (\mathsf{M_1}\ (\mathsf{M_1}\ \mathsf{U_1}))) \\
\quad \mathsf{from}\ (\mathsf{x} : \mathsf{xs}) = \mathsf{M_1}\ (\mathsf{R_1}\ (\mathsf{M_1}\ (\mathsf{M_1}\ (\mathsf{K_1}\ \mathsf{x})\ {:\times:}\ \mathsf{M_1}\ (\mathsf{K_1}\ \mathsf{xs})))) \\
\quad \mathsf{to}\ (\mathsf{M_1}\ (\mathsf{L_1}\ (\mathsf{M_1}\ (\mathsf{M_1}\ \mathsf{U_1})))) \qquad\qquad\qquad\qquad\quad = [\,] \\
\quad \mathsf{to}\ (\mathsf{M_1}\ (\mathsf{R_1}\ (\mathsf{M_1}\ (\mathsf{M_1}\ (\mathsf{K_1}\ \mathsf{x})\ {:\times:}\ \mathsf{M_1}\ (\mathsf{K_1}\ \mathsf{xs}))))) = \mathsf{x} : \mathsf{xs}
\end{array}
$$

data DL

data CN
data CC

data SU
data SP
data SR

Writing a Generic Function

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

Writing a Generic Function

instance GEq U_1 **where**
  geq _ _ = True

Writing a Generic Function

$$
\begin{array}{l}
\textbf{instance}\ (\mathsf{Eq}\ \mathsf{a}) \Rightarrow \mathsf{GEq}\ (\mathsf{K}_1\ \mathsf{i}\ \mathsf{a})\ \textbf{where} \\
\quad \mathsf{geq}\ (\mathsf{K}_1\ \mathsf{a})\ (\mathsf{K}_1\ \mathsf{b}) = \mathsf{a} \equiv \mathsf{b}
\end{array}
$$

Writing a Generic Function

```
instance (GEq a) ⇒ GEq (M₁ i c a) where
  geq (M₁ a) (M₁ b) = geq a b
```

Writing a Generic Function

```
instance (GEq a, GEq b) ⇒ GEq (a :×: b) where
  geq (a1 :×: b1) (a2 :×: b2) = geq a1 a2 ∧ geq b1 b2
```

Writing a Generic Function

$$
\begin{array}{l}
\textbf{instance}\ (\mathsf{GEq\ a}, \mathsf{GEq\ b}) \Rightarrow \mathsf{GEq}\ (\mathsf{a} :+: \mathsf{b})\ \textbf{where} \\
\quad \mathsf{geq}\ (\mathsf{L_1\ a})\ (\mathsf{L_1\ b}) \ = \mathsf{geq\ a\ b} \\
\quad \mathsf{geq}\ (\mathsf{R_1\ a})\ (\mathsf{R_1\ b}) = \mathsf{geq\ a\ b} \\
\quad \mathsf{geq}\ \_\ \_ \qquad\qquad\quad\ = \mathsf{False}
\end{array}
$$

Design of Generic Deriving

Goals:

1. Support **deriving** Functor
2. Support **deriving** for Eq, Ord, Enum, Bounded, Show, and Read
3. Simplicity
   - Reduced number of representation types
   - Invisible to user of **deriving**

deriving Functor

class Functor f **where**
  $\text{fmap} :: (a \rightarrow b) \rightarrow f\ a \rightarrow f\ b$

- Need structure representation
- Need type of parameter
- Need location of elements with parameter type

Structure Types

We use the same structural elements we saw previously:

$$
\begin{array}{lll}
\textbf{data} & \mathrm{U_1} & \mathrm{p} = \mathrm{U_1} \\
\textbf{data} & (\mathrm{f} :\times: \mathrm{g}) & \mathrm{p} = \mathrm{f\ p} :\times: \mathrm{g\ p} \\
\textbf{data} & (\mathrm{f} :+: \mathrm{g}) & \mathrm{p} = \mathrm{L_1}\ \{\mathrm{unL_1} :: \mathrm{f\ p}\} \mid \mathrm{R_1}\ \{\mathrm{unR_1} :: \mathrm{g\ p}\} \\
\textbf{newtype} & \mathrm{K_1\ i\ a} & \mathrm{p} = \mathrm{K_1}\ \{\mathrm{unK_1} :: \mathrm{a}\} \\
\textbf{newtype} & \mathrm{M_1\ i\ c\ f} & \mathrm{p} = \mathrm{M_1}\ \{\mathrm{unM_1} :: \mathrm{f\ p}\}
\end{array}
$$

Plus, we add a few more...

Structure Types

Parameter types:

newtype $\mathsf{Par}_1\ \mathsf{p} = \mathsf{Par}_1\ \{\mathsf{unPar}_1 :: \mathsf{p}\}$

- Now, we see that `p` is used in the parameter position in Par_1.

Structure Types

Recursive types:

newtype $\mathsf{Rec}_1\ \mathsf{f}\ \mathsf{p} = \mathsf{Rec}_1\ \{\mathsf{unRec}_1 :: \mathsf{f}\ \mathsf{p}\}$

- Note that Rec_1 is actually more general than recursion, since f can be any functorial type.

Type Representation

The type representation has type-level and term-level components.

$$
\begin{array}{l}
\textbf{class}\ \mathsf{Generic}_1\ \mathsf{f}\ \textbf{where} \\
\quad \textbf{type}\ \mathsf{Rep}_1\ \mathsf{f} :: * \to * \\
\quad \mathsf{from}_1 \qquad :: \mathsf{f}\ \mathsf{p} \to \mathsf{Rep}_1\ \mathsf{f}\ \mathsf{p} \\
\quad \mathsf{to}_1 \qquad\ \ :: \mathsf{Rep}_1\ \mathsf{f}\ \mathsf{p} \to \mathsf{f}\ \mathsf{p}
\end{array}
$$

- Unlike with `Generic`, the `p` is used in the type.

Simplified Representation of Lists

```
instance Generic₁ [] where
  type Rep₁ [] = U₁ :+: Par₁ :×: Rec₁ []
  from₁ []       = L₁ U₁
  from₁ (x : xs) = R₁ (Par₁ x :×: Rec₁ xs)
  to₁ (L₁ U₁)                     = []
  to₁ (R₁ (Par₁ x :×: Rec₁ xs)) = x : xs
```

Writing the Generic Functor

```
class Functor f where
  fmap :: (a → b) → f a → f b
  default fmap :: (Generic₁ f, Functor (Rep₁ f)) ⇒ (a → b) → f a → f b
  fmap = fmap_default
```

Writing the Generic Functor

instance Functor U_1 **where**
  fmap _ $U_1 = U_1$

instance Functor (K_1 i c) **where**
  fmap _ (K_1 a) = K_1 a

instance (Functor f) $\Rightarrow$ Functor (M_1 i c f) **where**
  fmap f (M_1 a) = M_1 (fmap f a)

Writing the Generic Functor

```
instance (Functor f, Functor g) ⇒ Functor (f :×: g) where
  fmap f (a :×: b) = fmap f a :×: fmap f b
```

Writing the Generic Functor

```
instance (Functor f, Functor g) ⇒ Functor (f :+: g) where
  fmap f (L1 a) = L1 (fmap f a)
  fmap f (R1 a) = R1 (fmap f a)
```

Writing the Generic Functor

```
instance Functor Par1 where
  fmap f (Par1 a) = Par1 (f a)
```

Writing the Generic Functor

instance (Functor f) $\Rightarrow$ Functor (Rec_1 f) **where**
  fmap f (Rec_1 a) $=$ Rec_1 (fmap f a)

Writing the Generic Functor

$\text{fmap\_default} :: (\text{Generic}_1\ f, \text{Functor}\ (\text{Rep}_1\ f)) \Rightarrow (a \to b) \to f\ a \to f\ b$
$\text{fmap\_default}\ f = \text{to}_1 \circ \text{fmap}\ f \circ \text{from}_1$

instance Functor [ ] **where**

Yet Another Generic Function

Lest you think GD is not that useful, let's look at a very practical example: the `binary` package.

- "Efficient, pure binary serialisation using lazy ByteStrings"
- "Serialisation speeds of over 1 G/sec have been observed, so this library should be suitable for high performance scenarios."

Using binary

- Monads
  - `Get` - a `State` monad carrying around the input `ByteString`
  - `Put` - a `Writer` monad over the efficient `Builder` monoid
- Primitives
  - putWord8 :: Word8 → Put
  - getWord8 :: Get Word8
- Serialization
  - encode :: (Binary a) ⇒ a → ByteString
  - decode :: (Binary a) ⇒ ByteString → a

Using `binary`

```
class Binary t where
  put :: t → Put
  get :: Get t
```

Writing a Generic Binary

```
class Binary t where
  default put :: (Generic t, GBinary (Rep t)) ⇒ t → Put
  put :: t → Put
  put = put_default
  default get :: (Generic t, GBinary (Rep t)) ⇒ Get t
  get :: Get t
  get = get_default
```

Writing a Generic Binary

```
class GBinary f where
  gput :: f a → Put
  gget :: Get (f a)
```

Writing a Generic Binary

```
instance GBinary U₁ where
  gput U₁ = return ()
  gget    = return U₁
```

instance GBinary U_1 **where**
  gput U_1 = return ()
  gget = return U_1

instance (Binary a) ⇒ GBinary (K_1 i a) **where**
  gput (K_1 x) = put x
  gget = K_1 ⟨\$⟩ get

instance (GBinary a) ⇒ GBinary (M_1 i c a) **where**
  gput (M_1 x) = gput x
  gget = M_1 ⟨\$⟩ gget

Writing a Generic Binary

$$
\begin{array}{l}
\textbf{instance}\ (\mathsf{GBinary}\ \mathsf{f}, \mathsf{GBinary}\ \mathsf{g}) \Rightarrow \mathsf{GBinary}\ (\mathsf{f}\ {:}{\times}{:}\ \mathsf{g})\ \textbf{where} \\
\quad \mathsf{gput}\ (\mathsf{x}\ {:}{\times}{:}\ \mathsf{y}) = \mathsf{gput}\ \mathsf{x} \gg \mathsf{gput}\ \mathsf{y} \\
\quad \mathsf{gget} \qquad\qquad\ \ = ({:}{\times}{:})\ \langle\$\rangle\ \mathsf{gget}\ \langle\star\rangle\ \mathsf{gget}
\end{array}
$$

Writing a Generic Binary

$$
\begin{array}{l}
\textbf{instance}\ (\mathsf{GBinary}\ \mathsf{f}, \mathsf{GBinary}\ \mathsf{g}) \Rightarrow \mathsf{GBinary}\ (\mathsf{f} :\!+\!: \mathsf{g})\ \textbf{where} \\
\quad \mathsf{gput}\ (\mathsf{L}_1\ \mathsf{x}) = \mathsf{putWord8}\ 0 \gg \mathsf{gput}\ \mathsf{x} \\
\quad \mathsf{gput}\ (\mathsf{R}_1\ \mathsf{y}) = \mathsf{putWord8}\ 1 \gg \mathsf{gput}\ \mathsf{y} \\
\quad \mathsf{gget} = \textbf{do}\ \mathsf{w} \leftarrow \mathsf{getWord8} \\
\qquad\qquad\qquad \textbf{case}\ \mathsf{w}\ \textbf{of}\ 0 \rightarrow \mathsf{L}_1\ \langle\$\rangle\ \mathsf{gget} \\
\qquad\qquad\qquad\qquad\qquad\ \ \_ \rightarrow \mathsf{R}_1\ \langle\$\rangle\ \mathsf{gget}
\end{array}
$$

Writing a Generic Binary

$$\mathrm{put\_default} :: (\mathrm{Generic}\ t, \mathrm{GBinary}\ (\mathrm{Rep}\ t)) \Rightarrow t \rightarrow \mathrm{Put}$$
$$\mathrm{put\_default} = \mathrm{gput} \circ \mathrm{from}$$

$$\mathrm{get\_default} :: (\mathrm{Generic}\ t, \mathrm{GBinary}\ (\mathrm{Rep}\ t)) \Rightarrow \mathrm{Get}\ t$$
$$\mathrm{get\_default} = \mathrm{to}\ \langle\$\rangle\ \mathrm{gget}$$

How Does GD Compare?

- Compared to SYB:
  - More efficient
  - Easier to implement some kinds of functions
- Compared to other GP libraries:
  - Being included with GHC makes things easier
  - Can't write some generic functions: folds, zips

Resources

- José Pedro Magalhães, Atze Dijkstra, Johan Jeuring, and Andres Löh. A generic deriving mechanism for Haskell. Proceedings of Haskell 2010. pp. 3748.
- Section 7.17 Generic Programming. GHC User's Guide. Version 7.2.1.
- This talk on GitHub: `https://github.com/spl/dutchhug2011`