A Generic Deriving Mechanism for Haskell

José Pedro Magalhães
Atze Dijkstra, Johan Jeuring, Andres Löh

Summer School on Metaprogramming
Cambridge, UK

August 8, 2016
Outline

Overview

Viewpoints
  End user
  Compiler implementer
  Library writer

Conclusion
Overview

- Haskell has a number of (built-in) type classes that can automatically be derived: **Bounded, Enum, Eq, Ord, Read, and Show**
- This talk is about a mechanism that lets you define these classes and your own *in* Haskell such that they can be derived automatically
- Implemented in the Glasgow Haskell Compiler
Features

We can:

- Handle meta-information such as constructor names and field labels
- Derive all the Haskell 98 classes
- Derive most of the classes that GHC can derive, including `Typeable` and classes of kind $\star \rightarrow \star$ such as `Functor`
Outline

Overview

Viewpoints

End user
Compiler implementer
Library writer

Conclusion
Using generic functions

If a class is generic, it can be used in a **deriving** construct. Assuming a type class

```haskell
data Bit = 0 | 1

class Encode α where
  encode :: α → [Bit]
```

The end user can write

```haskell
data Exp = Const Int | Plus Exp Exp

deriving (Show, Encode)
```

and then use

```haskell
test :: [Bit]
test = encode (Plus (Const 1) (Const 2))
```
Outline

Overview

Viewpoints
  End user
  Compiler implementer
  Library writer

Conclusion
Basic idea

- For each datatype, there is an equivalent internal representation.
- All the concepts contained in the data construct (application, abstraction, choice, sequence, recursion) are captured by a limited set of representation types.
- The compiler generates an internal representation for every datatype, together with conversion functions and derived instances.
The type representation is available in a module (Generics.Deriving.Base).

The representation types need to be bundled with the compiler (much like Data.Data for syb on GHC), but the library itself (generic-deriving on Hackage) is portable.

The library contains a set of datatypes as well as a class that allows conversion between a datatype and its representation.
Example

\[
\text{data } \text{Exp} = \text{Const Int} \mid \text{Plus Exp Exp}
\]

\[
\text{type } \text{Rep}_0^{\text{Exp}} = \\
D_1 \; \text{Exp} \; ( \; C_1 \; \text{Const}_\text{Exp} \; (\text{Rec}_0 \; \text{Int}) \\
\quad + \; C_1 \; \text{Plus}_\text{Exp} \; (\text{Rec}_0 \; \text{Exp} \times \text{Rec}_0 \; \text{Exp}))
\]
Example

```haskell
data Exp = Const Int | Plus Exp Exp

type Rep^Exp_0 =
    ( ( Int )
    + ( Exp × Exp ))
```

Note that the representation is *shallow* – recursive calls are to \( \text{Exp} \), not \( \text{Rep}^\text{Exp}_0 \).

Most of the representation is meta-information about:
Example

```haskell
data Exp = Const Int | Plus Exp Exp

type Rep^{Exp}_{0} =
  D_{1} \$Exp ( ( Int )
  + ( Exp \times Exp ))
```

Note that the representation is shallow – recursive calls are to `Exp`, not `Rep^{Exp}_{0}`. Most of the representation is meta-information about:

- the datatype itself,
Example

```haskell
data Exp = Const Int | Plus Exp Exp

type Rep^Exp_0 =
  D_1 $Exp ( C_1 $Const^Exp_0 ( Int)
  + C_1 $Plus^Exp_0 ( Exp × Exp))
```

Note that the representation is *shallow* – recursive calls are to \( \text{Exp} \), not \( \text{Rep}^\text{Exp}_0 \).

Most of the representation is meta-information about:

- the datatype itself,
- the constructors,
Example

```haskell
data Exp = Const Int | Plus Exp Exp

type Rep^{Exp}_0 =
    D_1 $Exp \times C_1 $Const_{Exp} (Rec_0 \text{ Int})
    + C_1 $Plus_{Exp} (Rec_0 \text{ Exp} \times Rec_0 \text{ Exp})
```

Note that the representation is shallow – recursive calls are to \text{Exp}, not \text{Rep}^{Exp}_0.

Most of the representation is meta-information about:

- the datatype itself,
- the constructors,
- where recursive calls take place.
Our approach can handle type classes with parameters of both
- kind ⋆ such as Encode and Show;
- kind ⋆ → ⋆ such as Functor.

We therefore represent all datatypes at kind ⋆ → ⋆. Types of kind ⋆ get a dummy parameter in their representation.
The void type $V_1$ is for types without constructors. The unit type $U_1$ is for constructors without fields. Sums represent choice between constructors. Products represent sequencing of fields.
Meta-information

\[
\text{data } K_1 \ i \ \gamma \ \rho = K_1 \ \gamma \\
\text{data } M_1 \ i \ \mu \ \phi \ \rho = M_1 \ (\phi \ \rho)
\]

These types record additional information, such as names and fixity, for instance. They are instantiated as follows:

\[
\begin{align*}
\text{data } D & \quad -- \text{ datatypes} & \text{type } D_1 & = M_1 \ D \\
\text{data } C & \quad -- \text{ constructors} & \text{type } C_1 & = M_1 \ C \\
\text{data } S & \quad -- \text{ record selectors} & \text{type } S_1 & = M_1 \ S \\
\text{data } R & \quad -- \text{ recursive calls} & \text{type } \text{Rec}_0 & = K_1 \ R \\
\text{data } P & \quad -- \text{ parameters} & \text{type } \text{Par}_0 & = K_1 \ P
\end{align*}
\]

We group five combinators into two because we often do not care about all the different types of meta-information.
Example: meta-information for expressions

GHC automatically generates the following for Exp:

```haskell
data $Exp
data $Const_{Exp}
data $Plus_{Exp}

instance Datatype $Exp where
    moduleName _ = "ModuleName"
datatypeName _ = "Exp"

instance Constructor $Const_{Exp} where conName _ = "Const"
instance Constructor $Plus_{Exp} where conName _ = "Plus"
```

The classes **Datatype** and **Constructor** can hold more information if desired.
Conversion

We use a type class to mediate between values and representations:

```haskell
class Generic α where
    type Rep α :: ⋆ → ⋆
    from :: α → Rep α χ
    to   :: Rep α χ → α
```
Conversion

We use a type class to mediate between values and representations:

```haskell
class Generic α where
    type Rep α :: ⋆ → ⋆
    from :: α → Rep α χ
    to :: Rep α χ → α
```

Instance for Exp (automatically generated by GHC):

```haskell
instance Generic Exp where
    type Rep Exp = Rep^0_{Exp}
    from (Const n) = M_1 (L_1 (M_1 (K_1 n)))
    from (Plus e e') = M_1 (R_1 (M_1 (K_1 e × K_1 e')))
    to (M_1 (L_1 (M_1 (K_1 n)))) = Const n
    to (M_1 (R_1 (M_1 (K_1 e × K_1 e')))) = Plus e e'
```
Compiler support

For each datatype, the compiler generates the following:

- Meta-information, i.e. datatypes and class instances.
- Representation type synonym(s).
- `Generic` and/or `Generic_1` instance.

Each `deriving` construct simple gives rise to an empty instance (more on that later).
Design choices

There is a certain amount of flexibility in how the compiler generates the representation. For example, sums and products are currently balanced. It is not clear how much of these details should be part of the specification.
Outline

Overview

Viewpoints
- End user
- Compiler implementer
- Library writer

Conclusion
The library writer defines generic (derivable) functions. We use two classes: one for the base types (kind $\star$):

```haskell
class Encode $\alpha$ where
    encode :: $\alpha$ $\rightarrow$ [Bit]
```

and one for the representation types (kind $\star$ $\rightarrow$ $\star$):

```haskell
class Encode$_1$ $\phi$ where
    encode$_1$ :: $\phi$ $\chi$ $\rightarrow$ [Bit]
```
Simple cases

The generic cases are defined as instances of $\text{Encode}_1$:

\begin{verbatim}
instance $\text{Encode}_1 V_1$ where 
  $\text{encode}_1 _V = []$

instance $\text{Encode}_1 U_1$ where 
  $\text{encode}_1 _U = []$

instance $(\text{Encode}_1 \phi) \Rightarrow \text{Encode}_1 (M_1 \ i_\gamma \phi)$ where 
  $\text{encode}_1 (M_1 a) = \text{encode}_1 a$
\end{verbatim}
Sums and products

\[
\text{instance} \ (\text{Encode}_1 \ \phi, \text{Encode}_1 \ \psi) \Rightarrow \text{Encode}_1 \ (\phi + \psi) \ \text{where}
\]
\[
\text{encode}_1 \ (L_1 \ a) = 0 : \text{encode}_1 \ a
\]
\[
\text{encode}_1 \ (R_1 \ a) = 1 : \text{encode}_1 \ a
\]

\[
\text{instance} \ (\text{Encode}_1 \ \phi, \text{Encode}_1 \ \psi) \Rightarrow \text{Encode}_1 \ (\phi \times \psi) \ \text{where}
\]
\[
\text{encode}_1 \ (a \times b) = \text{encode}_1 \ a \uplus \text{encode}_1 \ b
\]
Constants and base types

For constants, we rely on Encode:

\[
\text{instance } (\text{Encode } \alpha) \Rightarrow \text{Encode}_1 (K_1 \iota \alpha) \text{ where } \\
\quad \text{encode}_1 (K_1 \alpha) = \text{encode } a
\]

In this way we close the recursive loop: if \( \alpha \) is a representable type, 
\text{encode} will call from and then \text{encode}_1 again.

For base types, we need to provide ad-hoc instances:

\[
\begin{align*}
\text{instance } &\text{Encode } \text{Int} & \text{ where } \text{encode} = \ldots \\
\text{instance } &\text{Encode } \text{Char} & \text{ where } \text{encode} = \ldots 
\end{align*}
\]
The generic case is provided by generic defaults:

```haskell
class Encode α where
  encode :: α → [Bit]
  default encode :: (Generic α, Encode₁ (Rep α))
  ⇒ α → [Bit]
  encode x = encode₁ (from x)
```

These are just like regular default methods, only with a different type signature.
Using the generic instance

We are done:

```haskell
data Exp = Const Int | Plus Exp Exp deriving Encode
```

will cause the generation of

```haskell
instance Encode Exp where
    encode x = encode₁ (from x)
```
Back to the internals: kind $\star \rightarrow \star$ types

For type constructors (kind $\star \rightarrow \star$) we use a few more representation types:

- newtype $\text{Par}_1$ \hspace{1cm} $\rho = \text{Par}_1 \rho$
- newtype $\text{Rec}_1 \phi$ \hspace{1cm} $\rho = \text{Rec}_1 (\phi \rho)$
- newtype $(\circ)$ $\phi \psi \rho = \text{Comp}_1 (\phi (\psi \rho))$

We use $\text{Par}_1$ to store the parameter, $\text{Rec}_1$ to encode recursive occurrences of type constructors, and $\circ$ for type composition (eg. lists of trees).
Example: representing lists I

```
\textbf{data} \text{List } \rho = \text{Nil} \mid \text{Cons } \rho \ (\text{List } \rho) \\
\textbf{deriving} \ (\text{Show, Encode, Functor})
```

The compiler generates instance of \textbf{Generic} for kind \(\star\) functions:

```
\textbf{type} \ \text{Rep}_{0}^{\text{List}}(\rho) = \\
D_{1} \ \text{List} \ (C_{1} \ \text{Nil}_{\text{List}} \ U_{1} \\
+ C_{1} \ \text{Cons}_{\text{List}} \ (\text{Par}_{0} \ \rho \times \ \text{Rec}_{0} \ (\text{List } \rho)))
```

```
\textbf{instance} \ \text{Generic} \ (\text{List } \rho) \ \textbf{where} \\
\textbf{type} \ \text{Rep} (\text{List } \rho) = \text{Rep}_{0}^{\text{List}}(\rho) \\
\text{from} \ \text{Nil} = M_{1} \ (L_{1} \ (M_{1} \ U_{1})) \\
\text{from} \ (\text{Cons } h \ t) = M_{1} \ (R_{1} \ (M_{1} \ (K_{1} \ h \times K_{1} \ t)))) \\
\text{to} \ (M_{1} \ (L_{1} \ (M_{1} \ U_{1})))) = \text{Nil} \\
\text{to} \ (M_{1} \ (R_{1} \ (M_{1} \ (K_{1} \ h \times K_{1} \ t))))) = \text{Cons } h \ t
```
Example: representing lists II

type $\text{Rep}_0^\text{List} \quad \rho =$

\[
D_1 \; \text{List} \; ( \; C_1 \; \text{Nil}_{\text{List}} \; U_1 \\
+ \; C_1 \; \text{Cons}_{\text{List}} \; (\text{Par}_0 \; \rho \; \times \; \text{Rec}_0 \; (\text{List} \; \rho)))
\]

And an instance of $\text{Generic}_1$ for kind $\star \rightarrow \star$ functions:

type $\text{Rep}_1^\text{List} = D_1 \; \text{List} \; ( \; C_1 \; \text{Nil}_{\text{List}} \; U_1 \\
+ \; C_1 \; \text{Cons}_{\text{List}} \; (\text{Par}_1 \; \times \; \text{Rec}_1 \; \text{List}))$

instance $\text{Generic}_1 \; \text{List}$ where

\[
\text{type} \quad \text{Rep}_1 \; \text{List} = \text{Rep}_1^\text{List} \\
\text{from}_1 \; \text{Nil} = M_1 \; (L_1 \; (M_1 \; U_1)) \\
\text{from}_1 \; (\text{Cons} \; h \; t) = M_1 \; (R_1 \; (M_1 \; (\text{Par}_1 \; h \; \times \; \text{Rec}_1 \; t)))) \\
\text{to}_1 \; (M_1 \; (L_1 \; (M_1 \; U_1))) = \text{Nil} \\
\text{to}_1 \; (M_1 \; (R_1 \; (M_1 \; (\text{Par}_1 \; h \; \times \; \text{Rec}_1 \; t)))) = \text{Cons} \; h \; t
\]
We show how to define **Functor** generically as an example of a kind \( \star \rightarrow \star \) function. For consistency, we again use two type classes:

```haskell
class Functor \( \phi \) where
  fmap :: (\( \rho \rightarrow \alpha \)) \rightarrow \phi \rho \rightarrow \phi \alpha

default fmap :: (Generic_1 \( \phi \), Functor_1 (Rep_1 \( \phi \)))
  \Rightarrow (\( \rho \rightarrow \alpha \)) \rightarrow \phi \rho \rightarrow \phi \alpha

fmap f x = to_1 (fmap_1 f (from_1 x))
```

```haskell
class Functor_1 \( \phi \) where
  fmap_1 :: (\( \rho \rightarrow \alpha \)) \rightarrow \phi \rho \rightarrow \phi \alpha
```
Generic map II

Most cases are trivial:

**instance** Functor₁ U₁ **where**
  fmap₁ f U₁ = U₁

**instance** Functor₁ (K₁ ι γ) **where**
  fmap₁ f (K₁ a) = K₁ a

**instance** (Functor₁ φ) ⇒ Functor₁ (M₁ ι γ φ) **where**
  fmap₁ f (M₁ a) = M₁ (fmap₁ f a)

**instance** (Functor₁ φ, Functor₁ ψ) ⇒ Functor₁ (φ + ψ) **where**
  fmap₁ f (L₁ a) = L₁ (fmap₁ f a)
  fmap₁ f (R₁ a) = R₁ (fmap₁ f a)

**instance** (Functor₁ φ, Functor₁ ψ) ⇒ Functor₁ (φ × ψ) **where**
  fmap₁ f (a × b) = fmap₁ f a × fmap₁ f b
The most interesting instance is the one for parameters:

\[
\text{instance } \text{Functor}_1 \text{ Par}_1 \text{ where }
\]
\[
fmap_1 f (\text{Par}_1 a) = \text{Par}_1 (f a)
\]

Recursion and composition rely on \text{Functor}:

\[
\text{instance } (\text{Functor } \phi) \Rightarrow \text{Functor}_1 (\text{Rec}_1 \phi) \text{ where }
\]
\[
fmap_1 f (\text{Rec}_1 a) = \text{Rec}_1 (fmap f a)
\]

\[
\text{instance } (\text{Functor } \phi, \text{Functor}_1 \psi) \Rightarrow \text{Functor}_1 (\phi \circ \psi) \text{ where }
\]
\[
fmap_1 f (\text{Comp}_1 x) = \text{Comp}_1 (fmap (fmap_1 f) x)
\]
Now the compiler can derive Functor for List:

```haskell
instance Functor List where
  fmap f x = to1 (fmap1 f (from1 x))
```
Outline

Overview

Viewpoints
  End user
  Compiler implementer
  Library writer

Conclusion
The deriving mechanism does not have to be “magic”: it can be explained in Haskell.

Derivable functions become accessible and portable.

We provide an implementation in GHC and detailed information on how to implement it for other compilers.

We hope that the behaviour of derived instances can be redefined in Haskell Prime, perhaps along the lines of our work.