A Generic Deriving Mechanism for Haskell

José Pedro Magalhães

Atze Dijkstra, Johan Jeuring, Andres Löh

Haskell SG Meetup
Singapore

June 7, 2018

Outline

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 $\star \to \star$ such as Functor

Outline

Using generic functions

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

```
data Bit = 0 | 1
class Encode α where
  encode :: α → [Bit]
```

The end user can write

```
data Exp = Const Int | Plus Exp Exp
  deriving (Show, Encode)
```

and then use

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

Outline

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

Type representation

- 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

data Exp $=$ Const Int | Plus Exp Exp

type $\mathrm{Rep}_0^{\mathsf{Exp}} =$
  D_1 \$Exp (C_1 \$Const$_{\mathsf{Exp}}$ (Rec_0 Int)
         $+$ C_1 \$Plus$_{\mathsf{Exp}}$ (Rec_0 Exp $\times$ Rec_0 Exp))

Example

data Exp = Const Int | Plus Exp Exp

type Rep_0^{Exp} =
$$(\qquad\qquad (\qquad Int)$$
$$+ \qquad\qquad (\qquad Exp \times \qquad Exp))$$

Note that the representation is *shallow* – recursive calls are to Exp, not Rep_0^{Exp}.

Most of the representation is meta-information about:

Example

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

type Rep₀^Exp =
  D₁ $Exp  (                    (        Int)
              +                 (        Exp ×        Exp))
```

Note that the representation is *shallow* – recursive calls are to Exp, not Rep_0^{Exp}.

Most of the representation is meta-information about:

- the datatype itself,

Example

data Exp = Const Int | Plus Exp Exp

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

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

Most of the representation is meta-information about:

- the datatype itself,
- the constructors,

Example

data Exp = Const Int | Plus Exp Exp

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

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

Most of the representation is meta-information about:

- the datatype itself,
- the constructors,
- where recursive calls take place.

Lifting

Our approach can handle type classes with parameters of both

- kind $\star$ such as Encode and Show;
- kind $\star \rightarrow \star$ such as Functor.

We therefore represent *all* datatypes at kind $\star \rightarrow \star$.

Types of kind $\star$ get a dummy parameter in their representation.

Representation types

data V_1 ρ

data U_1 $\rho = U_1$

data $(+)\ \phi\ \psi\ \rho = L_1\ (\phi\ \rho)\ |\ R_1\ (\psi\ \rho)$

data $(\times)\ \phi\ \psi\ \rho = \phi\ \rho \times \psi\ \rho$

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

data $K_1 \iota \gamma \quad \rho = K_1 \gamma$
data $M_1 \iota \mu \phi \rho = M_1 (\phi \rho)$

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

data D	-- datatypes	**type** D_1	$= M_1 D$
data C	-- constructors	**type** C_1	$= M_1 C$
data S	-- record selectors	**type** S_1	$= M_1 S$
data R	-- recursive calls	**type** Rec_0	$= K_1 R$
data P	-- parameters	**type** Par_0	$= K_1 P$

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:

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:

class Generic α **where**
    type Rep $\alpha :: \star \to \star$
    from $:: \alpha \to$ Rep $\alpha \; \chi$
    to $::$ Rep $\alpha \; \chi \to \alpha$

Conversion

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

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

Instance for Exp (automatically generated by GHC):

```
instance Generic Exp where
    type Rep Exp = Rep₀^Exp
    from (Const n)  = M₁ (L₁ (M₁ (K₁ n)))
    from (Plus e e') = M₁ (R₁ (M₁ (K₁ e × K₁ e')))
    to (M₁ (L₁ (M₁ (K₁ n))))            = Const n
    to (M₁ (R₁ (M₁ (K₁ e × K₁ 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 simply 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

Generic function definitions

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

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

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

```
class Encode₁ φ where
  encode₁ :: φ χ → [Bit]
```

Simple cases

The generic cases are defined as instances of $Encode_1$:

```
instance Encode₁ V₁ where
  encode₁ _      = [ ]
instance Encode₁ U₁ where
  encode₁ _      = [ ]
instance (Encode₁ φ) ⇒ Encode₁ (M₁ ι γ φ) where
  encode₁ (M₁ a) = encode₁ a
```

Sums and products

instance $(\mathsf{Encode}_1 \; \phi, \mathsf{Encode}_1 \; \psi) \Rightarrow \mathsf{Encode}_1 \; (\phi + \psi)$ **where**
$\quad$ $\mathsf{encode}_1 \; (\mathsf{L}_1 \; \mathsf{a}) \;\; = 0 : \mathsf{encode}_1 \; \mathsf{a}$
$\quad$ $\mathsf{encode}_1 \; (\mathsf{R}_1 \; \mathsf{a}) \;\; = 1 : \mathsf{encode}_1 \; \mathsf{a}$

instance $(\mathsf{Encode}_1 \; \phi, \mathsf{Encode}_1 \; \psi) \Rightarrow \mathsf{Encode}_1 \; (\phi \times \psi)$ **where**
$\quad$ $\mathsf{encode}_1 \; (\mathsf{a} \times \mathsf{b}) = \mathsf{encode}_1 \; \mathsf{a} \;+\!\!+\; \mathsf{encode}_1 \; \mathsf{b}$

Constants and base types

For constants, we rely on Encode:

```
instance (Encode α) ⇒ Encode₁ (K₁ ι α) where
    encode₁ (K₁ a) = encode a
```

In this way we close the recursive loop: if α is a representable type, encode will call from and then encode_1 again.

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

```
instance Encode Int    where encode = ...
instance Encode Char where encode = ...
```

Default generic instance

The generic case is provided by generic defaults:

```
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:

data Exp = Const Int | Plus Exp Exp **deriving** Encode

will cause the generation of

instance Encode Exp **where**
   encode x = encode$_1$ (from x)

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

newtype $\mathsf{Par_1}$ $\quad\quad \rho = \mathsf{Par_1} \quad \rho$

newtype $\mathsf{Rec_1}\ \phi$ $\quad \rho = \mathsf{Rec_1} \quad (\phi\ \rho)$

newtype $(\circ)$ $\quad \phi\ \psi\ \rho = \mathsf{Comp_1}\ (\phi\ (\psi\ \rho))$

We use $\mathsf{Par_1}$ to store the parameter, $\mathsf{Rec_1}$ to encode recursive occurrences of type constructors, and $\circ$ for type composition (eg. lists of trees).

Example: representing lists I

data List ρ = Nil | Cons ρ (List ρ)
    deriving (Show, Encode, Functor)

The compiler generates instance of Generic for kind $\star$ functions:

type $\text{Rep}_0^{\text{List}}$ ρ =
  D_1 \$List (C_1 \$$\text{Nil}_{\text{List}}$ U_1
                 + C_1 \$$\text{Cons}_{\text{List}}$ (Par_0 ρ $\times$ Rec_0 (List ρ)))

instance Generic (List ρ) **where**
  type Rep (List ρ) = $\text{Rep}_0^{\text{List}}$ ρ
  from Nil   = M_1 (L_1 (M_1 U_1))
  from (Cons h t) = M_1 (R_1 (M_1 (K_1 h $\times$ K_1 t)))

  to (M_1 (L_1 (M_1 U_1)))   = Nil
  to (M_1 (R_1 (M_1 (K_1 h $\times$ K_1 t)))) = Cons h t

Example: representing lists II

type $\text{Rep}_0^{\text{List}}\ \rho =$
    $\text{D}_1\ \$\text{List}\quad (\ \text{C}_1\ \$\text{Nil}_{\text{List}}\quad \text{U}_1$
                  $+\ \text{C}_1\ \$\text{Cons}_{\text{List}}\ (\text{Par}_0\ \rho \times \text{Rec}_0\ (\text{List}\ \rho)))$

And an instance of Generic_1 for kind $\star \rightarrow \star$ functions:

type $\text{Rep}_1^{\text{List}} = \text{D}_1\ \$\text{List}\quad (\ \text{C}_1\ \$\text{Nil}_{\text{List}}\quad \text{U}_1$
                        $+\ \text{C}_1\ \$\text{Cons}_{\text{List}}\ (\text{Par}_1 \times \text{Rec}_1\ \text{List}))$

instance Generic_1 List **where**
    type Rep_1 List $= \text{Rep}_1^{\text{List}}$
    from_1 Nil$\quad\quad = \text{M}_1\ (\text{L}_1\ (\text{M}_1\ \text{U}_1))$
    from_1 (Cons h t) $= \text{M}_1\ (\text{R}_1\ (\text{M}_1\ (\text{Par}_1\ \text{h} \times \text{Rec}_1\ \text{t})))$
    $\text{to}_1\ (\text{M}_1\ (\text{L}_1\ (\text{M}_1\ \text{U}_1)))\quad\quad\quad\quad = \text{Nil}$
    $\text{to}_1\ (\text{M}_1\ (\text{R}_1\ (\text{M}_1\ (\text{Par}_1\ \text{h} \times \text{Rec}_1\ \text{t})))) = \text{Cons h t}$

We show how to define Functor generically as an example of a kind $\star \to \star$ function. For consistency, we again use two type classes:

class Functor ϕ **where**
   fmap   :: $(\rho \to \alpha) \to \phi\ \rho \to \phi\ \alpha$
   default fmap :: $(\text{Generic}_1\ \phi, \text{Functor}_1\ (\text{Rep}_1\ \phi))$
                $\Rightarrow (\rho \to \alpha) \to \phi\ \rho \to \phi\ \alpha$
   fmap f x = to_1 (fmap_1 f (from_1 x))

class $\text{Functor}_1\ \phi$ **where**
   $\text{fmap}_1 :: (\rho \to \alpha) \to \phi\ \rho \to \phi\ \alpha$

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
```

Generic map II

The most interesting instance is the one for parameters:

instance Functor_1 Par_1 **where**
    fmap_1 f $(\text{Par}_1$ a$)$ = Par_1 (f a)

Recursion and composition rely on Functor:

instance $(\text{Functor } \phi) \Rightarrow \text{Functor}_1 \ (\text{Rec}_1 \ \phi)$ **where**
    fmap_1 f $(\text{Rec}_1$ a$)$ = Rec_1 (fmap f a)
instance $(\text{Functor } \phi, \text{Functor}_1 \ \psi) \Rightarrow \text{Functor}_1 \ (\phi \circ \psi)$ **where**
    fmap_1 f $(\text{Comp}_1$ x$)$ = Comp_1 (fmap $(\text{fmap}_1$ f) x)

Generic map III

Now the compiler can derive Functor for List:

instance Functor List **where**
    fmap f x = to_1 ($fmap_1$ f ($from_1$ x))

Outline

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.