A Generic Deriving Mechanism for Haskell

José Pedro Magalhães
Atze Dijkstra, Johan Jeuring, Andres Löh

Dept. of Information and Computing Sciences, Utrecht University
P.O. Box 80.089, 3508 TB Utrecht, The Netherlands
Web pages: http://www.cs.uu.nl/wiki/Center

September 16, 2010
Outline

Overview

Viewpoints

End user

Library writer

Compiler implementer

Conclusion
Outline

Overview

Viewpoints
  End user
  Library writer
  Compiler implementer

Conclusion
Overview

- Haskell has a number of (built-in) type classes that can automatically be derived: Bounded, Enum, Eq, Ord, Read, and Show.
- We present a mechanism that lets you define these classes and your own in Haskell such that they can be derived automatically.
- Similar to “Derivable Type Classes” implemented in GHC, but better integrated into Haskell, more lightweight (no new syntax), more complete, and more flexible.
- The mechanism is implemented in the Utrecht Haskell Compiler, and we describe formally how it can be implemented in other compilers.
Features

- We can handle meta-information such as constructors or field labels.
- We can derive all the Haskell 98 classes.
- We can 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

Library writer

Compiler implementer

Conclusion
Outline

Overview

Viewpoints

End user
Library writer
Compiler implementer

Conclusion
Using generic functions

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

\[
\text{data Bit} = 0 \mid 1
\]
\[
\text{class Encode } \alpha \text{ where}
\]
\[
\text{encode :: } \alpha \to [\text{Bit}]
\]

The end user can write

\[
\text{data Exp} = \text{Const Int} \mid \text{Plus Exp Exp}
\]
\[
\text{deriving (Show, Encode)}
\]

and then use

\[
\text{test :: } [\text{Bit}]
\]
\[
\text{test} = \text{encode (Plus (Const 1) (Const 2))}
\]
Outline

Overview

Viewpoints
  End user
  Library writer
  Compiler implementer

Conclusion
Basic idea

- For each datatype, there is an equivalent internal representation.
- All the concepts contained in the data construct (parameterization, choice, sequence, recursion, composition) are captured by a limited set of representation types.
- The library writer has to implement generic methods on this limited set of representation types only, the rest then comes for free.
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 Rep₀^Exp =

D₁ $Exp ( C₁ $Const_{Exp} (Rec₀ Int) + C₁ $Plus_{Exp} (Rec₀ Exp × Rec₀ Exp))
Example

\begin{verbatim}
  data Exp = Const Int | Plus Exp Exp

  type Rep^Exp_0 =
    (               (     Int)
      +               (     Exp ×     Exp))
\end{verbatim}

Note that the representation is shallow – recursive calls are to Exp, not Rep^Exp_0.

Most of the representation is meta-information about:
Example

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

\[
\textbf{type} \quad \text{Rep}_0^{\text{Exp}} =
\]
\[
D_1 \ \text{$\&\text{Exp}$ (}
\]
\[
+ \quad ( \text{ Int})
\]
\[
( \text{ Exp} \times \text{ Exp})
\]

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

Most of the representation is meta-information about:

- the datatype itself,
Example

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

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

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

Most of the representation is meta-information about:

- the datatype itself,
- the constructors,
Example

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

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

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

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 \( \star \) such as \texttt{Encode} and \texttt{Show};
- kind \( \star \to \star \) such as \texttt{Functor}.

We therefore represent all datatypes at kind \( \star \to \star \).

Types of kind \( \star \) get a dummy parameter in their representation.
Representation types

\[
\begin{align*}
data \ V_1 & \quad \rho \\
data \ U_1 & \quad \rho = U_1 \\
data \ (\ + \ ) \ \phi \ \psi \ \rho & = L_1 \ (\phi \ \rho) \ | \ R_1 \ (\psi \ \rho) \\
data \ (\times \ ) \ \phi \ \psi \ \rho & = \phi \ \rho \ \times \ \psi \ \rho
\end{align*}
\]

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

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

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

\[ \text{data } D \quad \text{-- datatypes} \]
\[ \text{data } C \quad \text{-- constructors} \]
\[ \text{data } S \quad \text{-- record selectors} \]
\[ \text{data } R \quad \text{-- recursive calls} \]
\[ \text{data } P \quad \text{-- parameters} \]

\[ \text{type } D_1 = M_1 D \]
\[ \text{type } C_1 = M_1 C \]
\[ \text{type } S_1 = M_1 S \]
\[ \text{type } \text{Rec}_0 = K_1 R \]
\[ \text{type } \text{Par}_0 = K_1 P \]

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

UHC automatically generates the following for Exp:

```hs
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 Representable\(_0\) \(\alpha\) \(\tau\) where
  from0 :: \(\alpha \to \tau\ \chi\)
  to0   :: \(\tau\ \chi \to \alpha\)
```
Conversion

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

```haskell
class Representable₀ α τ where
  from₀ :: α → τ χ
  to₀    :: τ χ → α
```

Instance for `Exp` (automatically generated by UHC):

```haskell
instance Representable₀ Exp Repₑ₀ Exp where
  from₀ (Const n) = M₁ (L₁ (M₁ (K₁ n)))
  from₀ (Plus e e') = M₁ (R₁ (M₁ (K₁ e × K₁ e')))
  to₀ (M₁ (L₁ (M₁ (K₁ n)))) = Const n
  to₀ (M₁ (R₁ (M₁ (K₁ e × K₁ e')))) = Plus e e'
```
A note on extensions

The `Representable_0` class could use a functional dependency:

```
class Representable_0 α τ | α → τ where ...
```

Alternatively, `τ` could be encoded as an associated type:

```
class Representable_0 α where
  type Rep_0 α :: ⊥ → ⊥
  from_0 :: α → Rep_0 α χ
  to_0 :: Rep_0 α χ → α
```

But we want to stay inside Haskell98 as much as possible. We only require support for multi-parameter type classes.
Generic function definitions

We use two classes: one for the base types (kind $\star$):

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

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

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

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

\[
\text{instance } \text{Encode}_1 \ V_1 \ \text{where} \\
\quad \text{encode}_1 \ _ \quad = \ [\ ]
\]

\[
\text{instance } \text{Encode}_1 \ U_1 \ \text{where} \\
\quad \text{encode}_1 \ _ \quad = \ [\ ]
\]

\[
\text{instance } (\text{Encode}_1 \ \phi) \ \Rightarrow \ \text{Encode}_1 \ (M_1 \ \iota \ \gamma \ \phi) \ \text{where} \\
\quad \text{encode}_1 \ (M_1 \ a) = \text{encode}_1 \ a
\]
Sums and products

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

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

For constants, we rely on Encode:

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

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

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

\[
\text{instance } \text{Encode } \text{Int} \text{ where } \text{encode} = \ldots
\]
\[
\text{instance } \text{Encode } \text{Char} \text{ where } \text{encode} = \ldots
\]
Default generic instance

Every generic function needs a default case:

\[
\text{encodeDefault} :: (\text{Representable}_0 \alpha \tau, \text{Encode}_1 \tau) \\
\Rightarrow \tau \chi \rightarrow \alpha \rightarrow [\text{Bit}]
\]

\[
\text{encodeDefault rep x} = \text{encode}_1 ((\text{from}_0 x) \text{‘asTypeOf’ rep})
\]

\{-\# \text{DERIVABLE Encode encode encodeDefault \#-}\}
Default generic instance

Every generic function needs a default case:

```
encodeDefault :: (Representable₀ α τ, Encode₁ τ) ⇒ τ → α → [Bit]
encodeDefault rep x = encode₁ ((from₀ x) `asTypeOf` rep)
```

\{-# DERIVABLE Encode encode encodeDefault #-\}

We are done:

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

will cause the generation of

```
instance Encode Exp where
  encode = encodeDefault (⊥ :: Rep₀² Exp α)
```

Representing kind $\star \rightarrow \star$ types

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

- **newtype** $\text{Par}_1 \rho = \text{Par}_1 \rho$
- **newtype** $\text{Rec}_1 \phi \rho = \text{Rec}_1 (\phi \rho)$
- **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

\[
\textbf{data} \text{ List } \rho = \text{ Nil } | \text{ Cons } \rho (\text{ List } \rho)
\]

\textbf{deriving} (Show, Encode, Functor)

We need an instance of \texttt{Representable}_0 for kind \( \star \) functions:

\[
\textbf{type} \ \texttt{Rep}_0^{\text{List}} \rho =
\begin{align*}
D_1 & \text{ List } (C_1 \ \texttt{Nil}_\text{List} \ U_1 \\
+ C_1 & \texttt{Cons}_\text{List} (\text{ Par}_0 \ \rho \times \text{ Rec}_0 (\text{ List } \rho)))
\end{align*}
\]

\textbf{instance} \texttt{Representable}_0 (\text{ List } \rho) (\texttt{Rep}_0^{\text{List}} \rho) \textbf{ where}

\[
\begin{align*}
\text{from}_0 \text{ Nil} & = M_1 (L_1 (M_1 U_1)) \\
\text{from}_0 (\text{ Cons } h t) & = M_1 (R_1 (M_1 (K_1 h \times K_1 t)))
\end{align*}
\]

\[
\begin{align*}
\text{to}_0 (M_1 (L_1 (M_1 U_1))) & = \text{ Nil} \\
\text{to}_0 (M_1 (R_1 (M_1 (K_1 h \times K_1 t)))) & = \text{ Cons } h t
\end{align*}
\]
Example: representing lists II

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

\[
\text{type } \text{Rep}_1^\text{List} = D_1 \, \text{List} \, (C_1 \, \text{Nil}_1 \, U_1 \\
+ C_1 \, \text{Cons}_1 \, (\text{Par}_1 \times \text{Rec}_1 \, \text{List}))
\]

\text{instance } \text{Representable}_1 \, \text{List} \, \text{Rep}_1^\text{List} \, \text{where}

\[
\begin{align*}
\text{from}_1 \, \text{Nil} &= M_1 \, (L_1 \, (M_1 \, U_1)) \\
\text{from}_1 \, (\text{Cons} \, h \, t) &= M_1 \, (R_1 \, (M_1 \, (\text{Par}_1 \, h \times \text{Rec}_1 \, t)))) \\
\text{to}_1 \, (M_1 \, (L_1 \, (M_1 \, U_1))) &= \text{Nil} \\
\text{to}_1 \, (M_1 \, (R_1 \, (M_1 \, (\text{Par}_1 \, h \times \text{Rec}_1 \, t))))) &= \text{Cons} \, h \, t
\end{align*}
\]
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

class Functor$_1$ $\phi$ where
  fmap$_1$ :: (\rho \rightarrow \alpha) \rightarrow \phi \rho \rightarrow \phi \alpha
```
Generic map I

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:

\begin{verbatim}
class Functor $\phi$ where 
fmap :: ($\rho \rightarrow \alpha$) $\rightarrow$ $\phi \rho \rightarrow \phi \alpha$

class Functor$_1$ $\phi$ where 
fmap$_1$ :: ($\rho \rightarrow \alpha$) $\rightarrow$ $\phi \rho \rightarrow \phi \alpha$
\end{verbatim}

Recursion and composition rely on Functor:

\begin{verbatim}
instance (Functor $\phi$) $\Rightarrow$ Functor$_1$ (Rec$_1$ $\phi$) where 
fmap$_1$ f (Rec$_1$ a) = Rec$_1$ (fmap f a)

instance (Functor $\phi$, Functor$_1$ $\psi$) $\Rightarrow$ Functor$_1$ ($\phi \circ \psi$) where 
fmap$_1$ f (Comp$_1$ x) = Comp$_1$ (fmap (fmap$_1$ f) x)
\end{verbatim}
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 III

The most interesting instance is the one for parameters:

```haskell
instance Functor₁ Par₁ where
  fmap₁ f (Par₁ a) = Par₁ (f a)
```

The default case applies the conversion functions:

```haskell
{-# DERIVABLE Functor fmap fmapDefault #-}

fmapDefault :: (Representable₁ φ τ, Functor₁ τ) ⇒ τ ρ → (ρ → α) → φ ρ → φ α

fmapDefault rep f x = to₁ (fmap₁ f (from₁ x `asTypeOf` rep))
```

And we are now ready to derive Functor for List.
Outline

Overview

Viewpoints

End user

Library writer

Compiler implementer

Conclusion
Compiler support

For each datatype, the compiler generates the following:

- Meta-information, i.e. datatypes and class instances.
- Representation type synonym(s).
- `Representable_0` and/or `Representable_1` instance.

For each `deriving` construct, the compiler looks for an appropriate `DERIVABLE` pragma and generates a default instance.
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 yet how much of these details should be part of the specification.
Outline

Overview

Viewpoints
- End user
- Library writer
- Compiler implementer

Conclusion
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 UHC and detailed information on how to implement it for other compilers.
- We hope that the behavior of derived instances can be redefined in Haskell Prime, perhaps along the lines of our work.