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 30, 2010
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

- 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”, but better integrated into Haskell

- Implemented in the Utrecht Haskell Compiler

- We describe formally how it can be implemented in other compilers
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

```
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

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.
Example

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

type Rep^Exp_0 =
    D_1 $Exp ( C_1 $Const^{Exp}_{Exp} (Rec_0 Int)
    + C_1 $Plus^{Exp}_{Exp} (Rec_0 Exp \times Rec_0 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 Exp, not Rep_{Exp}^0.

Most of the representation is meta-information about:
Example

**data** \(\text{Exp} = \text{Const Int} \mid \text{Plus Exp Exp}\)

**type** \(\text{Rep}_{0}^{\text{Exp}} =\)

\[
D_1 \text{Exp} (\text{Const} \text{Exp} (\text{Int} + \text{Exp} \times \text{Exp}))
\]

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

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

\[
\textbf{type} \ \text{Rep}_{0}^{\text{Exp}} = \\
D_{1} \ \text{Exp} \ (\ C_{1} \ \text{Const}_{\text{Exp}} \ (\ \text{Int}) \ \\
+ \ C_{1} \ \text{Plus}_{\text{Exp}} \ (\ \text{Exp} \times \ \text{Exp}))
\]

Note that the representation is \textit{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,
Example

**data** \( \text{Exp} = \text{Const} \text{ Int} | \text{Plus} \text{ Exp Exp} \)

**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.
Lifting

Our approach can handle type classes with parameters of both

- kind \(*\) such as \texttt{Encode} and \texttt{Show};
- kind \(* \rightarrow \ast\) such as \texttt{Functor}.

We therefore represent all datatypes at kind \(* \rightarrow \ast\).

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

\begin{align*}
\text{data } & V_1 \quad \rho \\
\text{data } & U_1 \quad \rho = U_1 \\
\text{data } & (+) \phi \psi \rho = L_1 (\phi \rho) \mid R_1 (\psi \rho) \\
\text{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

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

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 $\text{Exp}$:

```haskell
data $\text{Exp}$
data $\text{Const}_{\text{Exp}}$
data $\text{Plus}_{\text{Exp}}$

instance Datatype $\text{Exp}$ where
    moduleName _ = "ModuleName"
    datatypeName _ = "Exp"

instance Constructor $\text{Const}_{\text{Exp}}$ where conName _ = "Const"
instance Constructor $\text{Plus}_{\text{Exp}}$ 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 Representable₀ α τ where
    from₀ :: α → τ χ
    to₀    :: τ χ → α
```

Instance for Exp (automatically generated by UHC):

```haskell
instance Representable₀ 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'
```
Conversion

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

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

Instance for Exp (automatically generated by UHC):

instance Representable₀ Exp Rep⁰ 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 `Representable0` class could use a functional dependency:

```haskell
class Representable0 α τ | α → τ where . . .
```

Alternatively, \( \tau \) could be encoded as an associated type:

```haskell
class Representable0 α where
    type Rep0 α :: ⋆ → ⋆
    from0 :: α → Rep0 α χ
    to0 :: Rep0 α χ → α
```

But we want to stay inside Haskell98 as much as possible. We only require support for multi-parameter type classes.
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 (specified by the library writer) and generates a default instance.
Outline

Overview

Viewpoints

End user

Compiler implementer

Library writer

Conclusion
Generic function definitions

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 \( \text{Encode}_1 \):

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

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

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

\textbf{instance} (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 \\

\textbf{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 + \text{encode}_1 b
Constants and base types

For constants, we rely on \texttt{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, \texttt{encode} will call \texttt{from} and then \texttt{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:

\[
\text{encode}_{\text{Default}} :: (\text{Representable}_0 \alpha \tau, \text{Encode}_1 \tau) \\
\Rightarrow \tau \chi \to \alpha \to [\text{Bit}]
\]

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

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

Every generic function needs a default case:

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

\[\text{encode}_{\text{Default}} \text{ rep } x = \text{encode}_1 (((\text{from}_0 x) \text{'} \text{asTypeOf} \text{'} \text{rep})\]

\{\neg \# \text{ DERIVABLE Encode encode encode}_{\text{Default}} \# - \}

We are done:

\textbf{data Exp} = \text{Const Int} \mid \text{Plus Exp Exp} \textbf{deriving Encode}

will cause the generation of

\textbf{instance Encode Exp where}

\[\text{encode} = \text{encode}_{\text{Default}} (\bot :: \text{Rep}_0^\text{Exp} \chi)\]
Back to the internals: kind $\star \to \star$ types

For type constructors (kind $\star \to \star$), 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

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

The compiler generates instance of `Representable0` for kind `⋆` functions:

```haskell
type Rep₀List ρ =
  D₁ $List ( C₁ $NilList U₁ + C₁ $ConsList (Par₀ ρ × Rec₀ (List ρ))))
```

```haskell
instance Representable₀ (List ρ) (Rep₀List ρ) where
  from₀ Nil = M₁ (L₁ (M₁ U₁))
  from₀ (Cons h t) = M₁ (R₁ (M₁ (K₁ h × K₁ t))))
  to₀ (M₁ (L₁ (M₁ U₁))) = Nil
  to₀ (M₁ (R₁ (M₁ (K₁ h × K₁ t)))) = Cons h t
```
Example: representing lists II

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

And an instance of \text{Representable}_1 for kind \( \star \to \star \) functions:

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

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

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

\[
\begin{align*}
\text{to}_1 (M_1 (L_1 (M_1 U_1))) & = \text{Nil} \\
\text{to}_1 (M_1 (R_1 (M_1 (\text{Par}_1 h \times \text{Rec}_1 t)))) & = \text{Cons} \ h \ t
\end{align*}
\]
We show how to define `Functor` generically as an example of a kind $\star \to \star$ function. For consistency, we again use two type classes:

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

class Functor\(_1\) \( \phi \) where
    fmap\(_1\) :: (\( \rho \to \alpha \)) \to \( \phi \rho \to \phi \alpha \)
```

The most interesting instance is the one for parameters:

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

Recursion and composition rely on \text{Functor}:

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

\[
\text{instance } (\text{Functor } \phi, \text{Functor}_1 \ \psi) \Rightarrow \text{Functor}_1 \ (\phi \circ \psi) \text{ where }
\text{fmap}_1 \ f \ (\text{Comp}_1 \ x) = \text{Comp}_1 \ (\text{fmap} \ (\text{fmap}_1 \ f) \ x)
\]
Generic map III

The default case applies the conversion functions:

\[
\{-\# \text{ DERIVABLE Functor fmap fmap}_{\text{Default}} \#-\}\n\]
\[
fmap_{\text{Default}} :: (\text{Representable}_1 \phi \tau, \text{Functor}_1 \tau) \Rightarrow \tau \rho \rightarrow (\rho \rightarrow \alpha) \rightarrow \phi \rho \rightarrow \phi \alpha
\]
\[
fmap_{\text{Default}} \text{ rep } f x = \text{ to}_1 (\text{fmap}_1 f (\text{from}_1 x \text{ `asTypeOf` rep}))
\]
The default case applies the conversion functions:

\[
\{- \# \text{ DERIVABLE Functor fmap fmap}_{\text{Default}} \ # - \}
\]

\[
fmap_{\text{Default}} :: (\text{Representable}_1 \phi \tau, \text{Functor}_1 \tau) \\
\Rightarrow \tau \rho \to (\rho \to \alpha) \to \phi \rho \to \phi \alpha
\]

\[
fmap_{\text{Default}} \ rep \ f \ x = \text{to}_1 \ (fmap_1 \ f \ (\text{from}_1 \ x \ \text{｀asTypeOf´ rep}))
\]

Now the compiler can derive \text{Functor} for \text{List}:

\[
\text{instance Functor List where}
\]

\[
fmap = \text{fmap}_\text{List} (\bot :: \text{Rep}_1^{\text{List}} \rho) \ \text{where}
\]

\[
fmap_\text{List} :: \text{Rep}_1^{\text{List}} \rho \to (\rho \to \alpha) \to \text{List} \rho \to \text{List} \alpha
\]

\[
fmap_\text{List} = \text{fmap}_{\text{Default}}
\]
Outline

Overview

Viewpoints

End user

Compiler implementer

Library writer

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.