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 $\star \to \star$ 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

$$\textbf{data}\ \mathsf{Bit} = \mathsf{0} \mid \mathsf{1}$$

$$\textbf{class}\ \mathsf{Encode}\ \alpha\ \textbf{where}$$
$$\quad \mathsf{encode} :: \alpha \to [\mathsf{Bit}]$$

The end user can write

$$\textbf{data}\ \mathsf{Exp} = \mathsf{Const}\ \mathsf{Int} \mid \mathsf{Plus}\ \mathsf{Exp}\ \mathsf{Exp}$$
$$\quad \textbf{deriving}\ (\mathsf{Show}, \mathsf{Encode})$$

and then use

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

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

Example

$$\textbf{data}\ \mathsf{Exp} = \mathsf{Const}\ \mathsf{Int} \mid \mathsf{Plus}\ \mathsf{Exp}\ \mathsf{Exp}$$

$$\begin{aligned}&\textbf{type}\ \mathsf{Rep}_0^{\mathsf{Exp}} = \\ &\quad \mathsf{D}_1\ \$\mathsf{Exp}\ \ (\ \mathsf{C}_1\ \$\mathsf{Const}_{\mathsf{Exp}}\ (\mathsf{Rec}_0\ \mathsf{Int}) \\ &\qquad\qquad\quad + \mathsf{C}_1\ \$\mathsf{Plus}_{\mathsf{Exp}}\ \ (\mathsf{Rec}_0\ \mathsf{Exp} \times \mathsf{Rec}_0\ \mathsf{Exp}))\end{aligned}$$

Example

$$\textbf{data}\ \mathsf{Exp} = \mathsf{Const}\ \mathsf{Int} \mid \mathsf{Plus}\ \mathsf{Exp}\ \mathsf{Exp}$$

$$\begin{aligned}\textbf{type}\ \mathsf{Rep}_0^{\mathsf{Exp}} = {} & \\ & (\qquad\qquad (\qquad \mathsf{Int}) \\ & + \qquad\qquad (\qquad \mathsf{Exp} \times \qquad \mathsf{Exp}))\end{aligned}$$

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

Most of the representation is meta-information about:

Example

$$\textbf{data}\ \mathsf{Exp} = \mathsf{Const}\ \mathsf{Int} \mid \mathsf{Plus}\ \mathsf{Exp}\ \mathsf{Exp}$$

$$\begin{array}{l}\textbf{type}\ \mathsf{Rep}_0^{\mathsf{Exp}} = \\ \quad \mathsf{D}_1\ \$\mathsf{Exp}\ \ (\qquad (\quad \mathsf{Int}) \\ \qquad\qquad + \qquad (\quad \mathsf{Exp} \times \quad \mathsf{Exp}))\end{array}$$

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

Most of the representation is meta-information about:

- the datatype itself,

Example

$$\mathbf{data}\ \mathsf{Exp} = \mathsf{Const}\ \mathsf{Int} \mid \mathsf{Plus}\ \mathsf{Exp}\ \mathsf{Exp}$$

$$\begin{aligned}&\mathbf{type}\ \mathsf{Rep}_0^{\mathsf{Exp}} = \\ &\quad \mathsf{D}_1\ \$\mathsf{Exp}\ \ (\ \mathsf{C}_1\ \$\mathsf{Const}_{\mathsf{Exp}}\ (\qquad \mathsf{Int}) \\ &\qquad\qquad\qquad + \mathsf{C}_1\ \$\mathsf{Plus}_{\mathsf{Exp}}\ \ (\qquad \mathsf{Exp} \times \qquad \mathsf{Exp}))\end{aligned}$$

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

Most of the representation is meta-information about:

- the datatype itself,
- the constructors,

Example

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

$$\begin{aligned}&\textbf{type}\ \text{Rep}_0^{\text{Exp}} = \\ &\quad \text{D}_1\ \$\text{Exp}\ \ (\ \text{C}_1\ \$\text{Const}_{\text{Exp}}\ (\text{Rec}_0\ \text{Int}) \\ &\qquad\qquad + \text{C}_1\ \$\text{Plus}_{\text{Exp}}\ \ (\text{Rec}_0\ \text{Exp} \times \text{Rec}_0\ \text{Exp}))\end{aligned}$$

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,
- 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

$$
\begin{aligned}
&\textbf{data}\ V_1 && \rho \\
&\textbf{data}\ U_1 && \rho = U_1 \\
&\textbf{data}\ (+)\ \phi\ \psi\ \rho && = L_1\ (\phi\ \rho) \mid R_1\ (\psi\ \rho) \\
&\textbf{data}\ (\times)\ \phi\ \psi\ \rho && = \phi\ \rho \times \psi\ \rho
\end{aligned}
$$

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

$$\textbf{data}\ \mathsf{K}_1\ \iota\ \gamma\quad \rho = \mathsf{K}_1\ \gamma$$
$$\textbf{data}\ \mathsf{M}_1\ \iota\ \mu\ \phi\ \rho = \mathsf{M}_1\ (\phi\ \rho)$$

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

```
data D  -- datatypes              type D1   = M1 D
data C  -- constructors           type C1   = M1 C
data S  -- record selectors       type S1   = M1 S

data R  -- recursive calls        type Rec0 = K1 R
data P  -- parameters             type Par0 = K1 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:

```
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:

$$
\begin{array}{l}
\textbf{class}\ \mathsf{Representable_0}\ \alpha\ \tau\ \textbf{where} \\
\quad \mathsf{from_0} :: \alpha \to \tau\ \chi \\
\quad \mathsf{to_0}\ \ \ \ :: \tau\ \chi \to \alpha
\end{array}
$$

Conversion

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

$$
\begin{array}{l}
\textbf{class}\ \mathsf{Representable}_0\ \alpha\ \tau\ \textbf{where} \\
\quad \mathsf{from}_0 :: \alpha \to \tau\ \chi \\
\quad \mathsf{to}_0 \quad :: \tau\ \chi \to \alpha
\end{array}
$$

Instance for Exp (automatically generated by UHC):

$$
\begin{array}{l}
\textbf{instance}\ \mathsf{Representable}_0\ \mathsf{Exp}\ \mathsf{Rep}_0^{\mathsf{Exp}}\ \textbf{where} \\
\quad \mathsf{from}_0\ (\mathsf{Const}\ \mathsf{n}) \quad = \mathsf{M}_1\ (\mathsf{L}_1\ (\mathsf{M}_1\ (\mathsf{K}_1\ \mathsf{n}))) \\
\quad \mathsf{from}_0\ (\mathsf{Plus}\ \mathsf{e}\ \mathsf{e}') = \mathsf{M}_1\ (\mathsf{R}_1\ (\mathsf{M}_1\ (\mathsf{K}_1\ \mathsf{e} \times \mathsf{K}_1\ \mathsf{e}'))) \\
\quad \mathsf{to}_0\ (\mathsf{M}_1\ (\mathsf{L}_1\ (\mathsf{M}_1\ (\mathsf{K}_1\ \mathsf{n})))) \qquad\qquad = \mathsf{Const}\ \mathsf{n} \\
\quad \mathsf{to}_0\ (\mathsf{M}_1\ (\mathsf{R}_1\ (\mathsf{M}_1\ (\mathsf{K}_1\ \mathsf{e} \times \mathsf{K}_1\ \mathsf{e}')))) = \mathsf{Plus}\ \mathsf{e}\ \mathsf{e}'
\end{array}
$$

A note on extensions

The $\mathsf{Representable}_0$ class could use a functional dependency:

$$\mathbf{class}\ \mathsf{Representable}_0\ \alpha\ \tau \mid \alpha \to \tau\ \mathbf{where} \ldots$$

Alternatively, τ could be encoded as an associated type:

$$\begin{array}{l} \mathbf{class}\ \mathsf{Representable}_0\ \alpha\ \mathbf{where} \\ \quad \mathbf{type}\ \mathsf{Rep}_0\ \alpha :: \star \to \star \\ \quad \mathsf{from}_0 :: \alpha \to \mathsf{Rep}_0\ \alpha\ \chi \\ \quad \mathsf{to}_0 \quad :: \mathsf{Rep}_0\ \alpha\ \chi \to \alpha \end{array}$$

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 $\star$):

$$\textbf{class}\ \text{Encode}\ \alpha\ \textbf{where}$$
$$\quad \text{encode} :: \alpha \rightarrow [\text{Bit}]$$

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

$$\textbf{class}\ \text{Encode}_1\ \phi\ \textbf{where}$$
$$\quad \text{encode}_1 :: \phi\ \chi \rightarrow [\text{Bit}]$$

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 (Encode₁ φ, Encode₁ ψ) ⇒ Encode₁ (φ + ψ) where
  encode₁ (L₁ a)  = 0 : encode₁ a
  encode₁ (R₁ a)  = 1 : encode₁ a

instance (Encode₁ φ, Encode₁ ψ) ⇒ Encode₁ (φ × ψ) where
  encode₁ (a × b) = encode₁ a ++ encode₁ b
```

Constants and base types

For constants, we rely on Encode:

$$\textbf{instance}\ (\text{Encode}\ \alpha) \Rightarrow \text{Encode}_1\ (\text{K}_1\ \iota\ \alpha)\ \textbf{where}$$
$$\quad \text{encode}_1\ (\text{K}_1\ \text{a}) = \text{encode a}$$

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:

$$\textbf{instance}\ \text{Encode Int}\quad \textbf{where}\ \text{encode} = \ldots$$
$$\textbf{instance}\ \text{Encode Char}\ \textbf{where}\ \text{encode} = \ldots$$

Default generic instance

Every generic function needs a default case:

$$
\begin{aligned}
\mathsf{encode}_{\mathsf{Default}} &:: (\mathsf{Representable}_0\ \alpha\ \tau, \mathsf{Encode}_1\ \tau) \\
&\Rightarrow \tau\ \chi \rightarrow \alpha \rightarrow [\mathsf{Bit}] \\
\mathsf{encode}_{\mathsf{Default}}\ &\mathsf{rep}\ \mathsf{x} = \mathsf{encode}_1\ ((\mathsf{from}_0\ \mathsf{x})\ \text{`}\mathsf{asTypeOf}\text{`}\ \mathsf{rep})
\end{aligned}
$$

$\{-\#$ DERIVABLE Encode encode $\mathsf{encode}_{\mathsf{Default}}$ $\#-\}$

Default generic instance

Every generic function needs a default case:

$$\begin{aligned}
&\mathsf{encode}_{\mathsf{Default}} :: (\mathsf{Representable}_0\ \alpha\ \tau, \mathsf{Encode}_1\ \tau) \\
&\qquad \Rightarrow \tau\ \chi \to \alpha \to [\mathsf{Bit}] \\
&\mathsf{encode}_{\mathsf{Default}}\ \mathsf{rep}\ \mathsf{x} = \mathsf{encode}_1\ ((\mathsf{from}_0\ \mathsf{x})\ \text{`asTypeOf`}\ \mathsf{rep})
\end{aligned}$$

{-# DERIVABLE Encode encode $\mathsf{encode}_{\mathsf{Default}}$ #-}

We are done:

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

will cause the generation of

instance Encode Exp **where**
  $\mathsf{encode} = \mathsf{encode}_{\mathsf{Default}}\ (\bot :: \mathsf{Rep}_0^{\mathsf{Exp}}\ \chi)$

Back to the internals: kind $\star \to \star$ types

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

$$
\begin{aligned}
&\textbf{newtype}\ \mathsf{Par_1}\ \ \ \ \ \ \ \ \rho = \mathsf{Par_1}\ \ \ \rho \\
&\textbf{newtype}\ \mathsf{Rec_1}\ \phi\ \ \ \ \rho = \mathsf{Rec_1}\ \ \ (\phi\ \rho) \\
&\textbf{newtype}\ (\circ)\ \ \ \phi\ \psi\ \rho = \mathsf{Comp_1}\ (\phi\ (\psi\ \rho))
\end{aligned}
$$

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

$$\textbf{data}\ \mathsf{List}\ \rho = \mathsf{Nil} \mid \mathsf{Cons}\ \rho\ (\mathsf{List}\ \rho)$$
$$\quad \textbf{deriving}\ (\mathsf{Show}, \mathsf{Encode}, \mathsf{Functor})$$

The compiler generates instance of $\mathsf{Representable}_0$ for kind $\star$ functions:

$$\begin{array}{l}
\textbf{type}\ \mathsf{Rep}_0^{\mathsf{List}}\ \rho = \\
\quad \mathsf{D}_1\ \$\mathsf{List}\ \ (\ \ \mathsf{C}_1\ \$\mathsf{Nil}_{\mathsf{List}}\quad \mathsf{U}_1 \\
\qquad\qquad\quad + \mathsf{C}_1\ \$\mathsf{Cons}_{\mathsf{List}}\ (\mathsf{Par}_0\ \rho \times \mathsf{Rec}_0\ (\mathsf{List}\ \rho)))
\end{array}$$

$$\begin{array}{l}
\textbf{instance}\ \mathsf{Representable}_0\ (\mathsf{List}\ \rho)\ (\mathsf{Rep}_0^{\mathsf{List}}\ \rho)\ \textbf{where} \\
\quad \mathsf{from}_0\ \mathsf{Nil} \qquad\qquad = \mathsf{M}_1\ (\mathsf{L}_1\ (\mathsf{M}_1\ \mathsf{U}_1)) \\
\quad \mathsf{from}_0\ (\mathsf{Cons}\ \mathsf{h}\ \mathsf{t}) = \mathsf{M}_1\ (\mathsf{R}_1\ (\mathsf{M}_1\ (\mathsf{K}_1\ \mathsf{h} \times \mathsf{K}_1\ \mathsf{t}))) \\
\quad \mathsf{to}_0\ (\mathsf{M}_1\ (\mathsf{L}_1\ (\mathsf{M}_1\ \mathsf{U}_1))) \qquad\qquad\qquad = \mathsf{Nil} \\
\quad \mathsf{to}_0\ (\mathsf{M}_1\ (\mathsf{R}_1\ (\mathsf{M}_1\ (\mathsf{K}_1\ \mathsf{h} \times \mathsf{K}_1\ \mathsf{t})))) = \mathsf{Cons}\ \mathsf{h}\ \mathsf{t}
\end{array}$$

Example: representing lists II

$$
\begin{aligned}
&\textbf{type}\ \mathsf{Rep}_0^{\mathsf{List}}\ \rho = \\
&\quad \mathsf{D}_1\ \$\mathsf{List}\ \ (\ \mathsf{C}_1\ \$\mathsf{Nil}_{\mathsf{List}}\ \ \ \mathsf{U}_1 \\
&\qquad\qquad\quad + \mathsf{C}_1\ \$\mathsf{Cons}_{\mathsf{List}}\ (\mathsf{Par}_0\ \rho \times \mathsf{Rec}_0\ (\mathsf{List}\ \rho)))
\end{aligned}
$$

And an instance of $\mathsf{Representable}_1$ for kind $\star \to \star$ functions:

$$
\begin{aligned}
\textbf{type}\ \mathsf{Rep}_1^{\mathsf{List}} = \mathsf{D}_1\ \$\mathsf{List}\ \ &(\ \mathsf{C}_1\ \$\mathsf{Nil}_{\mathsf{List}}\ \ \ \mathsf{U}_1 \\
&+ \mathsf{C}_1\ \$\mathsf{Cons}_{\mathsf{List}}\ (\mathsf{Par}_1 \times \mathsf{Rec}_1\ \mathsf{List}))
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{instance}\ \mathsf{Representable}_1\ \mathsf{List}\ \mathsf{Rep}_1^{\mathsf{List}}\ \textbf{where} \\
&\quad \mathsf{from}_1\ \mathsf{Nil} \qquad\qquad = \mathsf{M}_1\ (\mathsf{L}_1\ (\mathsf{M}_1\ \mathsf{U}_1)) \\
&\quad \mathsf{from}_1\ (\mathsf{Cons}\ \mathsf{h}\ \mathsf{t}) = \mathsf{M}_1\ (\mathsf{R}_1\ (\mathsf{M}_1\ (\mathsf{Par}_1\ \mathsf{h} \times \mathsf{Rec}_1\ \mathsf{t}))) \\
&\quad \mathsf{to}_1\ (\mathsf{M}_1\ (\mathsf{L}_1\ (\mathsf{M}_1\ \mathsf{U}_1))) \qquad\qquad\qquad\quad = \mathsf{Nil} \\
&\quad \mathsf{to}_1\ (\mathsf{M}_1\ (\mathsf{R}_1\ (\mathsf{M}_1\ (\mathsf{Par}_1\ \mathsf{h} \times \mathsf{Rec}_1\ \mathsf{t})))) = \mathsf{Cons}\ \mathsf{h}\ \mathsf{t}
\end{aligned}
$$

Back to the library writer: 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:

$$
\begin{aligned}
&\textbf{class}\ \text{Functor}\ \ \phi\ \textbf{where}\\
&\quad \text{fmap}\ \ :: (\rho \rightarrow \alpha) \rightarrow \phi\ \rho \rightarrow \phi\ \alpha\\
&\textbf{class}\ \text{Functor}_1\ \phi\ \textbf{where}\\
&\quad \text{fmap}_1 :: (\rho \rightarrow \alpha) \rightarrow \phi\ \rho \rightarrow \phi\ \alpha
\end{aligned}
$$

Generic map II

The most interesting instance is the one for parameters:

$$\textbf{instance}\ \mathsf{Functor_1}\ \mathsf{Par_1}\ \textbf{where}$$
$$\quad \mathsf{fmap_1}\ f\ (\mathsf{Par_1}\ a) = \mathsf{Par_1}\ (f\ a)$$

Recursion and composition rely on Functor:

$$\textbf{instance}\ (\mathsf{Functor}\ \phi) \Rightarrow \mathsf{Functor_1}\ (\mathsf{Rec_1}\ \phi)\ \textbf{where}$$
$$\quad \mathsf{fmap_1}\ f\ (\mathsf{Rec_1}\ a) = \mathsf{Rec_1}\ (\mathsf{fmap}\ f\ a)$$
$$\textbf{instance}\ (\mathsf{Functor}\ \phi, \mathsf{Functor_1}\ \psi) \Rightarrow \mathsf{Functor_1}\ (\phi \circ \psi)\ \textbf{where}$$
$$\quad \mathsf{fmap_1}\ f\ (\mathsf{Comp_1}\ x) = \mathsf{Comp_1}\ (\mathsf{fmap}\ (\mathsf{fmap_1}\ f)\ x)$$

Generic map III

The default case applies the conversion functions:

$$\{\text{-\# } \texttt{DERIVABLE} \text{ Functor fmap fmap}_{\text{Default}} \text{ \#-}\}$$

$$\begin{aligned}\text{fmap}_{\text{Default}} &:: (\text{Representable}_1\ \phi\ \tau, \text{Functor}_1\ \tau) \\ &\Rightarrow \tau\ \rho \to (\rho \to \alpha) \to \phi\ \rho \to \phi\ \alpha\end{aligned}$$

$$\text{fmap}_{\text{Default}}\ \text{rep f x} = \text{to}_1\ (\text{fmap}_1\ \text{f}\ (\text{from}_1\ \text{x `asTypeOf` rep}))$$

Generic map III

The default case applies the conversion functions:

$$\{\text{-\# DERIVABLE Functor fmap fmap}_{\text{Default}} \text{ \#-}\}$$
$$\begin{aligned}
\text{fmap}_{\text{Default}} &:: (\text{Representable}_1\ \phi\ \tau, \text{Functor}_1\ \tau) \\
&\Rightarrow \tau\ \rho \to (\rho \to \alpha) \to \phi\ \rho \to \phi\ \alpha \\
\text{fmap}_{\text{Default}}\ \text{rep f x} &= \text{to}_1\ (\text{fmap}_1\ \text{f}\ (\text{from}_1\ \text{x `asTypeOf` rep}))
\end{aligned}$$

Now the compiler can derive Functor for List:

$$\begin{aligned}
&\textbf{instance}\ \text{Functor List}\ \textbf{where} \\
&\quad \text{fmap} = \text{fmap}_{\text{List}}\ (\bot :: \text{Rep}_1^{\text{List}}\ \rho)\ \textbf{where} \\
&\quad\quad \text{fmap}_{\text{List}} :: \text{Rep}_1^{\text{List}}\ \rho \to (\rho \to \alpha) \to \text{List}\ \rho \to \text{List}\ \alpha \\
&\quad\quad \text{fmap}_{\text{List}} = \text{fmap}_{\text{Default}}
\end{aligned}$$

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 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.