Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

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

- 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 $\star \to \star$ such as Functor.

Outline

Overview

Viewpoints

- End user
- Library writer
- Compiler implementer

Conclusion

Outline

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

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

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

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

Example

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

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

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

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

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

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,

Example

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

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

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

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

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

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,
- 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

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

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

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

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

```
data D  -- datatypes             type 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:

$$\textbf{data}\ \$\mathrm{Exp}$$
$$\textbf{data}\ \$\mathrm{Const}_{\mathrm{Exp}}$$
$$\textbf{data}\ \$\mathrm{Plus}_{\mathrm{Exp}}$$

```
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:

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

Conversion

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

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

Instance for Exp (automatically generated by UHC):

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

A note on extensions

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

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

Alternatively, τ could be encoded as an associated type:

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

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

Generic function definitions

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

$$
\begin{array}{l}
\textbf{class}\ \mathrm{Encode}\ \alpha\ \textbf{where} \\
\quad \mathrm{encode} :: \alpha \rightarrow [\mathrm{Bit}]
\end{array}
$$

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

$$
\begin{array}{l}
\textbf{class}\ \mathrm{Encode}_1\ \phi\ \textbf{where} \\
\quad \mathrm{encode}_1 :: \phi\ \chi \rightarrow [\mathrm{Bit}]
\end{array}
$$

Simple cases

The generic cases are defined as instances of Encode_1:

$$
\begin{array}{l}
\textbf{instance}\ \mathsf{Encode}_1\ \mathsf{V}_1\ \textbf{where} \\
\quad \mathsf{encode}_1\ \_ \qquad\quad = [\,] \\
\textbf{instance}\ \mathsf{Encode}_1\ \mathsf{U}_1\ \textbf{where} \\
\quad \mathsf{encode}_1\ \_ \qquad\quad = [\,] \\
\textbf{instance}\ (\mathsf{Encode}_1\ \phi) \Rightarrow \mathsf{Encode}_1\ (\mathsf{M}_1\ \iota\ \gamma\ \phi)\ \textbf{where} \\
\quad \mathsf{encode}_1\ (\mathsf{M}_1\ \mathsf{a}) = \mathsf{encode}_1\ \mathsf{a}
\end{array}
$$

Sums and products

$$
\begin{aligned}
&\textbf{instance}\ (\mathsf{Encode}_1\ \phi, \mathsf{Encode}_1\ \psi) \Rightarrow \mathsf{Encode}_1\ (\phi + \psi)\ \textbf{where}\\
&\quad \mathsf{encode}_1\ (\mathsf{L}_1\ \mathsf{a})\ = 0 : \mathsf{encode}_1\ \mathsf{a}\\
&\quad \mathsf{encode}_1\ (\mathsf{R}_1\ \mathsf{a})\ = 1 : \mathsf{encode}_1\ \mathsf{a}
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{instance}\ (\mathsf{Encode}_1\ \phi, \mathsf{Encode}_1\ \psi) \Rightarrow \mathsf{Encode}_1\ (\phi \times \psi)\ \textbf{where}\\
&\quad \mathsf{encode}_1\ (\mathsf{a} \times \mathsf{b}) = \mathsf{encode}_1\ \mathsf{a} +\!\!+\ \mathsf{encode}_1\ \mathsf{b}
\end{aligned}
$$

Constants and base types

For constants, we rely on Encode:

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

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:

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

Default generic instance

Every generic function needs a default case:

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

$$\{-\#\ \texttt{DERIVABLE}\ \mathsf{Encode\ encode\ encodeDefault}\ \#-\}$$

Default generic instance

Every generic function needs a default case:

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

```
{-# 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**
$$\mathsf{encode} = \mathsf{encodeDefault}\ (\bot :: \mathsf{Rep}_0^{\mathsf{Exp}}\ \chi)$$

Representing kind $\star \to \star$ types

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

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

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

Example: representing lists I

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

We need an instance of $\mathsf{Representable}_0$ for kind $\star$ functions:

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

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

Example: representing lists II

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

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

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

Generic map I

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:

$$
\begin{array}{l}
\textbf{class}\ \mathsf{Functor}\ \ \phi\ \textbf{where} \\
\quad \mathsf{fmap}\ \ :: (\rho \to \alpha) \to \phi\ \rho \to \phi\ \alpha \\
\textbf{class}\ \mathsf{Functor}_1\ \phi\ \textbf{where} \\
\quad \mathsf{fmap}_1 :: (\rho \to \alpha) \to \phi\ \rho \to \phi\ \alpha
\end{array}
$$

Generic map I

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:

$$
\begin{aligned}
&\textbf{class}\ \mathsf{Functor}\ \ \phi\ \textbf{where}\\
&\quad \mathsf{fmap}\ \ :: (\rho \to \alpha) \to \phi\ \rho \to \phi\ \alpha\\
&\textbf{class}\ \mathsf{Functor_1}\ \phi\ \textbf{where}\\
&\quad \mathsf{fmap_1} :: (\rho \to \alpha) \to \phi\ \rho \to \phi\ \alpha
\end{aligned}
$$

Recursion and composition rely on Functor:

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

Generic map II

Most cases are trivial:

$$
\begin{array}{l}
\textbf{instance}\ \mathsf{Functor_1}\ \mathsf{U_1}\ \textbf{where} \\
\quad \mathsf{fmap_1}\ \mathsf{f}\ \mathsf{U_1} \qquad\ = \mathsf{U_1} \\
\textbf{instance}\ \mathsf{Functor_1}\ (\mathsf{K_1}\ \iota\ \gamma)\ \textbf{where} \\
\quad \mathsf{fmap_1}\ \mathsf{f}\ (\mathsf{K_1}\ \ \mathsf{a}) = \mathsf{K_1}\ \ \mathsf{a} \\
\textbf{instance}\ (\mathsf{Functor_1}\ \phi) \Rightarrow \mathsf{Functor_1}\ (\mathsf{M_1}\ \iota\ \gamma\ \phi)\ \textbf{where} \\
\quad \mathsf{fmap_1}\ \mathsf{f}\ (\mathsf{M_1}\ \mathsf{a}) = \mathsf{M_1}\ (\mathsf{fmap_1}\ \mathsf{f}\ \mathsf{a}) \\
\textbf{instance}\ (\mathsf{Functor_1}\ \phi, \mathsf{Functor_1}\ \psi) \Rightarrow \mathsf{Functor_1}\ (\phi + \psi)\ \textbf{where} \\
\quad \mathsf{fmap_1}\ \mathsf{f}\ (\mathsf{L_1}\ \ \mathsf{a}) = \mathsf{L_1}\ \ (\mathsf{fmap_1}\ \mathsf{f}\ \mathsf{a}) \\
\quad \mathsf{fmap_1}\ \mathsf{f}\ (\mathsf{R_1}\ \ \mathsf{a}) = \mathsf{R_1}\ \ (\mathsf{fmap_1}\ \mathsf{f}\ \mathsf{a}) \\
\textbf{instance}\ (\mathsf{Functor_1}\ \phi, \mathsf{Functor_1}\ \psi) \Rightarrow \mathsf{Functor_1}\ (\phi \times \psi)\ \textbf{where} \\
\quad \mathsf{fmap_1}\ \mathsf{f}\ (\mathsf{a} \times \mathsf{b}) = \mathsf{fmap_1}\ \mathsf{f}\ \mathsf{a} \times \mathsf{fmap_1}\ \mathsf{f}\ \mathsf{b}
\end{array}
$$

Generic map III

The most interesting instance is the one for parameters:

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

The default case applies the conversion functions:

$$\{\text{-\# } \texttt{DERIVABLE}\ \mathsf{Functor}\ \mathsf{fmap}\ \mathsf{fmap_{Default}}\ \text{\#-}\}$$
$$\mathsf{fmap_{Default}} :: (\mathsf{Representable_1}\ \phi\ \tau, \mathsf{Functor_1}\ \tau)$$
$$\Rightarrow \tau\ \rho \rightarrow (\rho \rightarrow \alpha) \rightarrow \phi\ \rho \rightarrow \phi\ \alpha$$
$$\mathsf{fmap_{Default}}\ \mathsf{rep}\ \mathsf{f}\ \mathsf{x} = \mathsf{to_1}\ (\mathsf{fmap_1}\ \mathsf{f}\ (\mathsf{from_1}\ \mathsf{x}\ \text{`asTypeOf`}\ \mathsf{rep}))$$

And we are now ready to derive Functor for List.

Outline

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

Conclusion

- The deriving mechanism does not have to be "magic": it can be explained in Haskell.
- Derivable functions become acessible 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.