A Generic Deriving Mechanism for Haskell

José Pedro Magalhães
Atze Dijkstra, Johan Jeuring, Andres Löh

Summer School on Metaprogramming
Cambridge, UK

August 8, 2016

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
- 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

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

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

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

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

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

Example

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

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

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:

Example

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

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

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

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

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

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

We therefore represent *all* datatypes at kind $\star \to \star$.
Types of kind $\star$ get a dummy parameter in their representation.

Representation types

$$\begin{array}{l}
\textbf{data}\ \mathsf{V_1}\qquad\quad \rho \\
\textbf{data}\ \mathsf{U_1}\qquad\quad \rho = \mathsf{U_1} \\
\textbf{data}\ (+)\ \phi\ \psi\ \rho = \mathsf{L_1}\ (\phi\ \rho) \mid \mathsf{R_1}\ (\psi\ \rho) \\
\textbf{data}\ (\times)\ \phi\ \psi\ \rho = \phi\ \rho \times \psi\ \rho
\end{array}$$

The void type $\mathsf{V_1}$ is for types without constructors.
The unit type $\mathsf{U_1}$ is for constructors without fields.
Sums represent choice between constructors.
Products represent sequencing of fields.

Meta-information

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

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

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:

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

Conversion

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

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

Instance for Exp (automatically generated by GHC):

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

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 simple 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

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 $\star$):

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

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

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

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

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

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

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:

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

Default generic instance

The generic case is provided by generic defaults:

$$
\begin{array}{l}
\textbf{class}\ \mathsf{Encode}\ \alpha\ \textbf{where} \\
\quad \mathsf{encode} \qquad\qquad\ :: \ \alpha \rightarrow [\mathsf{Bit}] \\
\quad \textbf{default}\ \mathsf{encode} :: (\mathsf{Generic}\ \alpha, \mathsf{Encode}_1\ (\mathsf{Rep}\ \alpha)) \\
\qquad\qquad\qquad\qquad\ \Rightarrow \alpha \rightarrow [\mathsf{Bit}] \\
\quad \mathsf{encode}\ \mathsf{x} = \mathsf{encode}_1\ (\mathsf{from}\ \mathsf{x})
\end{array}
$$

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 = encode₁ (from x)
```

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

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

$$
\begin{array}{lllll}
\textbf{newtype}\ \mathsf{Par}_1 & & \rho = \mathsf{Par}_1 & & \rho \\
\textbf{newtype}\ \mathsf{Rec}_1\ \phi & & \rho = \mathsf{Rec}_1 & & (\phi\ \rho) \\
\textbf{newtype}\ (\circ) & \phi\ \psi & \rho = \mathsf{Comp}_1 & & (\phi\ (\psi\ \rho))
\end{array}
$$

We use Par_1 to store the parameter, 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 Generic for kind $\star$ functions:

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

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

Example: representing lists II

$$
\begin{aligned}
&\textbf{type}\ \mathsf{Rep}_0^{\mathsf{List}}\ \rho = \\
&\quad \mathsf{D_1}\ \$\mathsf{List}\ \ (\ \mathsf{C_1}\ \$\mathsf{Nil}_{\mathsf{List}}\quad \mathsf{U_1} \\
&\qquad\qquad\quad + \mathsf{C_1}\ \$\mathsf{Cons}_{\mathsf{List}}\ (\mathsf{Par}_0\ \rho \times \mathsf{Rec}_0\ (\mathsf{List}\ \rho)))
\end{aligned}
$$

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

$$
\begin{aligned}
\textbf{type}\ \mathsf{Rep}_1^{\mathsf{List}} = \mathsf{D_1}\ \$\mathsf{List}\ \ &(\ \mathsf{C_1}\ \$\mathsf{Nil}_{\mathsf{List}}\quad \mathsf{U_1} \\
&+ \mathsf{C_1}\ \$\mathsf{Cons}_{\mathsf{List}}\ (\mathsf{Par_1} \times \mathsf{Rec_1}\ \mathsf{List}))
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{instance}\ \mathsf{Generic_1}\ \mathsf{List}\ \textbf{where} \\
&\quad \textbf{type}\ \mathsf{Rep_1}\ \mathsf{List} = \mathsf{Rep}_1^{\mathsf{List}} \\
&\quad \mathsf{from_1}\ \mathsf{Nil} \qquad\qquad\ \ = \mathsf{M_1}\ (\mathsf{L_1}\ (\mathsf{M_1}\ \mathsf{U_1})) \\
&\quad \mathsf{from_1}\ (\mathsf{Cons}\ \mathsf{h}\ \mathsf{t}) = \mathsf{M_1}\ (\mathsf{R_1}\ (\mathsf{M_1}\ (\mathsf{Par_1}\ \mathsf{h} \times \mathsf{Rec_1}\ \mathsf{t}))) \\
&\quad \mathsf{to_1}\ (\mathsf{M_1}\ (\mathsf{L_1}\ (\mathsf{M_1}\ \mathsf{U_1}))) \qquad\qquad\qquad\quad\ \ = \mathsf{Nil} \\
&\quad \mathsf{to_1}\ (\mathsf{M_1}\ (\mathsf{R_1}\ (\mathsf{M_1}\ (\mathsf{Par_1}\ \mathsf{h} \times \mathsf{Rec_1}\ \mathsf{t})))) = \mathsf{Cons}\ \mathsf{h}\ \mathsf{t}
\end{aligned}
$$

Back to the library writer: generic map I

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

$$
\begin{array}{l}
\textbf{class}\ \mathsf{Functor}\ \phi\ \textbf{where} \\
\quad \mathsf{fmap} \qquad\qquad\quad :: (\rho \to \alpha) \to \phi\ \rho \to \phi\ \alpha \\
\quad \textbf{default}\ \mathsf{fmap} :: (\mathsf{Generic}_1\ \phi, \mathsf{Functor}_1\ (\mathsf{Rep}_1\ \phi)) \\
\qquad\qquad\qquad\qquad \Rightarrow (\rho \to \alpha) \to \phi\ \rho \to \phi\ \alpha \\
\quad \mathsf{fmap}\ \mathsf{f}\ \mathsf{x} = \mathsf{to}_1\ (\mathsf{fmap}_1\ \mathsf{f}\ (\mathsf{from}_1\ \mathsf{x}))
\end{array}
$$

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

Generic map II

Most cases are trivial:

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

Generic map II

The most interesting instance is the one for parameters:

$$\textbf{instance}\ \mathsf{Functor}_1\ \mathsf{Par}_1\ \textbf{where}$$
$$\quad \mathsf{fmap}_1\ \mathsf{f}\ (\mathsf{Par}_1\ \mathsf{a}) = \mathsf{Par}_1\ (\mathsf{f}\ \mathsf{a})$$

Recursion and composition rely on Functor:

$$\textbf{instance}\ (\mathsf{Functor}\ \phi) \Rightarrow \mathsf{Functor}_1\ (\mathsf{Rec}_1\ \phi)\ \textbf{where}$$
$$\quad \mathsf{fmap}_1\ \mathsf{f}\ (\mathsf{Rec}_1\ \mathsf{a}) = \mathsf{Rec}_1\ (\mathsf{fmap}\ \mathsf{f}\ \mathsf{a})$$
$$\textbf{instance}\ (\mathsf{Functor}\ \phi, \mathsf{Functor}_1\ \psi) \Rightarrow \mathsf{Functor}_1\ (\phi \circ \psi)\ \textbf{where}$$
$$\quad \mathsf{fmap}_1\ \mathsf{f}\ (\mathsf{Comp}_1\ \mathsf{x}) = \mathsf{Comp}_1\ (\mathsf{fmap}\ (\mathsf{fmap}_1\ \mathsf{f})\ \mathsf{x})$$

Generic map III

Now the compiler can derive Functor for List:

```
instance Functor List where
  fmap f x = to₁ (fmap₁ f (from₁ x))
```

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 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.