Derivable Type Classes

RALF HINZE
Universität Bonn

SIMON PEYTON JONES
Microsoft Research Ltd

September, 2000

(Pick the slides at
`http://www.cs.uni-bonn.de/~ralf/talks.html#T15`.)

Extending Haskell with generic programming

A *generic function* is one that the programmer writes once, but which works over many different data types.

The standard examples are parsing and printing, taking equality, ordering, and so on. In fact, all of Haskell's *derivable methods* are generic functions.

In this talk we show how to extend Haskell with the power of generic programming.

The *key idea* is to present generic programming as a richer language in which to write default method definitions in a class declaration.

A recap of Haskell's type classes

Haskell supports *overloading*, based on type classes.

$$
\begin{array}{l}
\mathbf{class}\ Eq\ t\ \mathbf{where} \\
\quad (==), (\neq) :: t \rightarrow t \rightarrow Bool
\end{array}
$$

Before you can use $(==)$ on values of, say Int, you must explain how to take equality over Int values:

$$
\begin{array}{l}
\mathbf{instance}\ Eq\ Int\ \mathbf{where} \\
\quad (==) = eqInt \\
\quad (\neq) = neInt.
\end{array}
$$

Default methods

The code for the $(\neq)$ method is usually the negation of the code for the $(==)$ method, and vice versa.

$$\begin{array}{l} \textbf{class } Eq\ t \textbf{ where} \\ \quad (==), (\neq) :: t \to t \to Bool \\ \quad x_1 == x_2 \quad = not\ (x_1 \neq x_2) \\ \quad x_1 \neq x_2 \quad = not\ (x_1 == x_2). \end{array}$$

In an instance declaration, you can now *either* give a definition for $(==)$, *or* a definition for $(\neq)$, or both. If you specify neither, then you will get an infinite loop!

Overloading is not generic programming

$$\mathbf{data}\ \mathit{Wibble} = \mathit{Wag}\ \mathit{Int} \mid \mathit{Wug}\ \mathit{Bool}.$$

It is "obvious" how to take equality over Wibble, and support for generic programming would allow us to specify this "obvious" precisely.

In Haskell, however, you have to give an explicit instance declaration, containing the code for equality:

$$\begin{array}{l}
\mathbf{instance}\ \mathit{Eq}\ \mathit{Wibble}\ \mathbf{where} \\
\quad \begin{array}{ll}
(\mathit{Wag}\ i_1) \mathrel{==} (\mathit{Wag}\ i_2) & = i_1 \mathrel{==} i_2 \\
(\mathit{Wug}\ b_1) \mathrel{==} (\mathit{Wug}\ b_2) & = b_1 \mathrel{==} b_2 \\
w_1 \mathrel{==} w_2 & = \mathit{False}.
\end{array}
\end{array}$$

deriving

However, for a handful of built-in classes Haskell provides special support — the so-called "**deriving**" mechanism.

$$\boxed{\textbf{data}\ \mathit{Wibble} = \mathit{Wag}\ \mathit{Int} \mid \mathit{Wug}\ \mathit{Bool}\ \textbf{deriving}\ (\mathit{Eq})}$$

The "**deriving** (Eq)" part tells Haskell to generate the "obvious" code for equality.

Stocktaking

Haskell's "**deriving**" mechanism is rather ad-hoc.

- ☐ The code generated for derived methods is specified only informally in an Appendix of the language definition.
- ☐ You cannot define your own derivable methods.
- ☐ In particular, you cannot modify existing derivable classes to adapt them to your needs (for instance, if you require a slight variant of the $Show$ class).
- ☐ The "**deriving**" mechanism does not work for higher-order kinded data types. In fact, it is not even possible to provide explicit instance declarations for these data types.

A generic equality class

To extend Haskell with the power of generic programming we provide a richer language for default method declarations.

$$
\begin{array}{ll}
\multicolumn{2}{l}{\textbf{class}\ Eq\ t\ \textbf{where}} \\
\quad (==), (\neq) & ::\ t \to t \to Bool \\
\qquad\quad \text{-- generic default method} & \\
\quad (==)\{1\} Unit\ Unit & = True \\
\quad (==)\{a+b\} (Inl\ x_1)\ (Inl\ x_2) & = x_1 == x_2 \\
\quad (==)\{a+b\} (Inr\ y_1)\ (Inr\ y_2) & = y_1 == y_2 \\
\quad (==)\{a+b\} \_\ \_ & = False \\
\quad (==)\{a*b\} (x_1 :\!*\!:\ y_1)\ (x_2 :\!*\!:\ y_2) & = x_1 == x_2 \wedge y_1 == y_2 \\
\qquad\quad \text{-- vanilla, non-generic default method} & \\
\quad (\neq)\ x_1\ x_2 & = not\ (x_1 == x_2)
\end{array}
$$

You can now give an instance declaration like this:

$$\textbf{instance}\ Eq\ Wibble$$

without giving code for either method.

This works equally well for parameterized types like

$$\textbf{data}\ List\ a = Cons\ a\ (List\ a) \mid Nil.$$

In this case the instance declaration takes the following form:

$$\textbf{instance}\ (Eq\ a) \Rightarrow Eq\ (List\ a).$$

Another example

The class $ShowBin$ has a method that converts a value to a list of bits:

$$
\begin{array}{ll}
\mathbf{data}\ Bit & = \mathsf{0} \mid \mathsf{1} \\
\mathbf{type}\ Bin & = [Bit] \\
\mathbf{class}\ ShowBin\ t\ \mathbf{where} & \\
\quad showBin & :: t \to Bin \\
\quad showBin\{1\}\,Unit & = [\,] \\
\quad showBin\{a + b\}(Inl\ x) & = \mathsf{0} : showBin\ x \\
\quad showBin\{a + b\}(Inr\ y) & = \mathsf{1} : showBin\ y \\
\quad showBin\{a * b\}(x :*: y) & = showBin\ x +\!\!+\ showBin\ y\,.
\end{array}
$$

NB. In the paper we also define $showBin$'s inverse.

Higher-order kinded types

Functional programmers love abstraction. In Haskell you can, for instance, abstract over the $List$ data type in

$$\textbf{data}\ Rose\ a = Branch\ a\ (List\ (Rose\ a))$$

to obtain the more general type:

$$\textbf{data}\ GRose\ f\ a = GBranch\ a\ (f\ (GRose\ f\ a)).$$

Here, the type variable "f" ranges over type constructors, rather than types.

What's the problem?

Consider defining an instance for $ShowBin\ (Rose\ a)$ by hand:

$$\textbf{instance}\ (ShowBin\ a) \Rightarrow ShowBin\ (Rose\ a)\ \textbf{where}$$
$$\quad showBin\ (Branch\ x\ ts) = showBin\ x +\!\!+\ showBin\ ts.$$

The first call to $showBin$ requires a $ShowBin\ a$ instance, the second a $List\ (Rose\ a)$ instance.

The latter requirement can be satisfied if we have an instance elsewhere of the form

$$\textbf{instance}\ (ShowBin\ t) \Rightarrow ShowBin\ (List\ t).$$

For $GRose$ matters are not so simple

$$\begin{array}{l}\textbf{instance}\ (???) \Rightarrow \mathit{ShowBin}\ (\mathit{GRose}\ f\ \ a)\ \textbf{where} \\ \quad \mathit{showBin}\ (\mathit{GBranch}\ x\ \ ts) = \mathit{showBin}\ x + \mathit{showBin}\ ts.\end{array}$$

The first call requires $\mathit{ShowBin}\ a$, the second $\mathit{ShowBin}\ (f\ \ (\mathit{GRose}\ f\ \ a))$.

Alas, we cannot write

$$\begin{array}{l}\textbf{instance}\ (\mathit{ShowBin}\ a, \mathit{ShowBin}\ (f\ \ (\mathit{GRose}\ f\ \ a))) \\ \qquad\qquad \Rightarrow \mathit{ShowBin}\ (\mathit{GRose}\ f\ \ a)\ \textbf{where} \\ \quad \mathit{showBin}\ (\mathit{GBranch}\ x\ \ ts) = \mathit{showBin}\ x + \mathit{showBin}\ ts.\end{array}$$

This is not legal Haskell and, even if it were, the typechecker would diverge. Indeed, no ordinary Haskell context will do.

A solution

The data type $GRose$ is obtained from $Rose$ by abstracting over the $List$ data type.

Likewise, we obtain a $GRose$ instance from the $Rose$ instance by abstracting over the $List$ instance.

$$\begin{array}{l}
\mathbf{instance}\ (ShowBin\ a, \forall b\,.\,(ShowBin\ b) \Rightarrow ShowBin\ (f\ b)) \\
\qquad\qquad\quad \Rightarrow ShowBin\ (GRose\ f\ a)\ \mathbf{where} \\
\quad showBin\ (GBranch\ x\ ts) = showBin\ x + showBin\ ts.
\end{array}$$

The second context is called a *polymorphic predicate*.

Discussion

- It is not necessary for the instance declaration to appear in the same module as the data type declaration, or the class declaration.

- The compiler only fills in a method definition if the programmer omits it. In other words, *the programmer can override the generic definition on a type-by-type basis*.

  This ability is crucial to support *abstract data types*.

- You can use instance declarations to specify what a generic operation should do on primitive types, such as $Char$, Int, $Double$, and "$\rightarrow$".

Towards the implementation: generic representation types

Recall the generic definition of $showBin$.

$$
\begin{aligned}
&showBin\{1\}\,Unit &&= [\,] \\
&showBin\{a+b\}(Inl\ x) &&= \mathsf{0} : showBin\ x \\
&showBin\{a+b\}(Inr\ y) &&= \mathsf{1} : showBin\ y \\
&showBin\{a*b\}(x :\!*\!: y) &&= showBin\ x +\!\!+\ showBin\ y.
\end{aligned}
$$

How can we use this generic definition to generate specific instances?

In Haskell a data declaration defines a sum of products. For instance,

$$\mathbf{data}\ List\ a = Cons\ a\ (List\ a) \mid Nil$$

is (more or less) isomorphic to

$$\mathbf{type}\ List^{\circ}\ a = (a * List\ a) + 1.$$

We call $List^{\circ}$ the *generic representation type* for $List$.

The unit, sum, and product types are defined like this:

$$\begin{aligned}
&\mathbf{data}\ 1 &&= Unit \\
&\mathbf{data}\ a + b &&= Inl\ a \mid Inr\ b \\
&\mathbf{data}\ a * b &&= a :\!*\!: b.
\end{aligned}$$

The following functions make the isomorphism explicit:

$$
\begin{aligned}
&\mathit{to\text{-}List} && :: \forall a\,.\,\mathit{List}^{\circ}\ a \to \mathit{List}\ a \\
&\mathit{to\text{-}List}\ (\mathit{Inl}\ (x :\!*\!:\ \mathit{xs})) && = \mathit{Cons}\ x\ \mathit{xs} \\
&\mathit{to\text{-}List}\ (\mathit{Inr}\ \mathit{Unit}) && = \mathit{Nil}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{from\text{-}List} && :: \forall a\,.\,\mathit{List}\ a \to \mathit{List}^{\circ}\ a \\
&\mathit{from\text{-}List}\ (\mathit{Cons}\ x\ \mathit{xs}) && = \mathit{Inl}\ (x :\!*\!:\ \mathit{xs}) \\
&\mathit{from\text{-}List}\ \mathit{Nil} && = \mathit{Inr}\ \mathit{Unit}.
\end{aligned}
$$

Generic instances

Now, the key idea is that *by regarding a* $List$ *as a* $List^{\circ}$*, the generic code explains what to do*.

To this end it is useful to imagine re-expressing the default methods as three ordinary instance declarations:

$$
\begin{array}{l}
\mathbf{instance}\ \mathit{ShowBin}\ 1\ \mathbf{where} \\
\quad \mathit{showBin}\ \mathit{Unit} \qquad = [\,] \\
\mathbf{instance}\ (\mathit{ShowBin}\ a, \mathit{ShowBin}\ b) \Rightarrow \mathit{ShowBin}\ (a + b)\ \mathbf{where} \\
\quad \mathit{showBin}\ (\mathit{Inl}\ x) \;\; = \mathtt{0} : \mathit{showBin}\ x \\
\quad \mathit{showBin}\ (\mathit{Inr}\ y) \;\; = \mathtt{1} : \mathit{showBin}\ y \\
\mathbf{instance}\ (\mathit{ShowBin}\ a, \mathit{ShowBin}\ b) \Rightarrow \mathit{ShowBin}\ (a * b)\ \mathbf{where} \\
\quad \mathit{showBin}\ (x :\!*\!: y) = \mathit{showBin}\ x +\!\!+ \mathit{showBin}\ y.
\end{array}
$$

Filling in the missing methods

Now, when the programmer writes

$$\textbf{instance}\ (\mathit{ShowBin}\ a) \Rightarrow \mathit{ShowBin}\ (\mathit{List}\ a)$$

the compiler will fill in the method declaration as follows:

$$\mathit{showBin}\ \mathit{xs} = \mathit{showBin}\ (\mathit{from\text{-}List}\ \mathit{xs}).$$

Compare this to the definition one would write by hand:

$$\begin{aligned} &\mathit{showBin}\ \mathit{Nil} &&= \mathsf{0} : [\,] \\ &\mathit{showBin}\ (\mathit{Cons}\ x\ \mathit{xs}) &&= \mathsf{1} : \mathit{showBin}\ x \mathbin{+\!\!+} \mathit{showBin}\ \mathit{xs}. \end{aligned}$$

Constructor names

To be able to give a generic definition for $showsPrec$, the *names* of the constructors, and their fixities, must be made accessible.

To this end we provide a new type $ConDescr$ that comprises all constructor names and the following set of primitives.

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathbf{data}\ ConDescr} \\
\multicolumn{3}{l}{\mathbf{data}\ Fixity = Nofix \mid Infix\ Int \mid Infixl\ Int \mid Infixr\ Int} \\
conName & :: & ConDescr \to String \\
conArity & :: & ConDescr \to Int \\
conFixity & :: & ConDescr \to Fixity \\
\multicolumn{3}{l}{\mathbf{instance}\ Show\ ConDescr\ \mathbf{where}} \\
\quad show & = & conName
\end{array}
$$

Furthermore, we provide a new type pattern c **of** a, where c is a *value* variable of type $ConDescr$ and a is a *type* variable.

The generic representation type for $List$ is now

$$\boxed{\mathbf{type}\ List^{\circ}\ a = con\text{-}Cons\ \mathbf{of}\ (a * List\ a) + con\text{-}Nil\ \mathbf{of}\ 1,}$$

where $con\text{-}Cons$ and $con\text{-}Nil$ are the elements of $ConDescr$ that describe $Cons$ and Nil, respectively.

A generic $Show$ class

Using these features we can implement a simple variant of Haskell's $Show$ class.

$$
\begin{array}{ll}
\textbf{class}\ Show\ t\ \textbf{where} & \\
\quad showPrec & :: Int \to t \to String \\
\quad showPrec\{\!|\ a + b\ |\!\}\, d\ (Inl\ x) & = showPrec\ d\ x \\
\quad showPrec\{\!|\ a + b\ |\!\}\, d\ (Inr\ y) & = showPrec\ d\ y \\
\quad showPrec\{\!|\ c\ \textbf{of}\ a\ |\!\}\, d\ (Con\ c\ x) & \\
\qquad \mid conArity\ c == 0 & = show\ c \\
\qquad \mid otherwise & = showParen\ (d \geqslant 10)\ str \\
\qquad \textbf{where}\ str & = show\ c +\!\!+ \texttt{" "} +\!\!+ showPrec\ 10\ x \\
\quad showPrec\{\!|\ a * b\ |\!\}\, d\ (x :\!*\!: y) & = showPrec\ d\ x +\!\!+ \texttt{" "} +\!\!+ showPrec\ d\ y
\end{array}
$$

A prettier $Show$ class

The $showPrec$ method generates one long string.

We can do better using pretty printing combinators.

$$
\begin{array}{ll}
nil & :: Doc \\
(\diamond) & :: Doc \to Doc \to Doc \\
text & :: String \to Doc \\
line & :: Doc \\
nest & :: Int \to Doc \to Doc \\
group & :: Doc \to Doc
\end{array}
$$

$$
\begin{array}{ll}
\multicolumn{2}{l}{\textbf{class } \mathit{Pretty}\ t \textbf{ where}} \\
\quad \mathit{ppPrec} & ::\ \mathit{Int} \to t \to \mathit{Doc} \\
\quad \mathit{ppPrec}\{\,a + b\,\}d\ (\mathit{Inl}\ x) & = \mathit{ppPrec}\ d\ x \\
\quad \mathit{ppPrec}\{\,a + b\,\}d\ (\mathit{Inr}\ y) & = \mathit{ppPrec}\ d\ y \\
\quad \mathit{ppPrec}\{\,c \textbf{ of } a\,\}d\ (\mathit{Con}\ c\ x) & \\
\qquad \mid \mathit{conArity}\ c == 0 & = \mathit{text}\ (\mathit{show}\ c) \\
\qquad \mid \mathit{otherwise} & = \mathit{group}\ (\mathit{nest}\ 2\ (\mathit{ppParen}\ (d \geqslant 10)\ \mathit{doc})) \\
\qquad \textbf{where } \mathit{doc} & = \mathit{text}\ (\mathit{show}\ c) \diamond \mathit{line} \diamond \mathit{ppPrec}\ 10\ x \\
\quad \mathit{ppPrec}\{\,a * b\,\}d\ (x :\!*\!:\ y) & = \mathit{ppPrec}\ d\ x \diamond \mathit{line} \diamond \mathit{ppPrec}\ d\ y
\end{array}
$$

Discussion

- We believe that our proposal fit nicely into the Haskell language. At first sight, generic programming and Haskell type classes are in competition, but we use generic programming to smoothly extend the power of type classes.
- Using generic definitions we are able to explain what "**deriving**" means in a systematic way. Furthermore, programmers can now add new derivable classes.
- Through polymorphic predicates programmers can define instances for higher-order kinded data types. (This extension is completely orthogonal to generic definitions).
- In the paper we describe the implementation in considerable detail. It is worth noting that there is no run-time passing or case-analysis of types, beyond Haskell's existing dictionary passing.

Future work

- We still have to complete the implementation. *Andrei Serjantov* is doing all the hard work.

- Currently, generic default declarations may be given only for type classes. We plan to lift this restriction so that we can also deal with *constructor classes*.

- It remains to investigate the interaction of generic definitions with

  - non-general instance heads,
  - multiple parameter type classes, and
  - implicit parameters.