Generics for the masses

RALF HINZE

Institut für Informatik III, Universität Bonn
Römerstraße 164, 53117 Bonn, Germany
Email: `ralf@informatik.uni-bonn.de`
Homepage: `http://www.informatik.uni-bonn.de/~ralf`

September, 2004

(Pick the slides at . . . /`~ralf/talks.html#T37`.)

Motivation

In Haskell, showing values of a data type is easy.

$$\textbf{data}\ \mathit{Tree}\ \alpha = \mathit{Leaf}\ \alpha \mid \mathit{Fork}\ (\mathit{Tree}\ \alpha)\ (\mathit{Tree}\ \alpha)$$
$$\qquad\qquad \textbf{deriving}\ (\mathit{Show})$$

Simply attach a deriving clause to the data type declaration; a suitable *show* function is then automatically generated by the compiler.

This *show* function is, for instance, implicitly called on the command line (the function *tree* has type $[\alpha] \rightarrow \mathit{Tree}\ \alpha$).

$$\mathit{Main}\rangle\quad \mathit{tree}\ [0\mathinner{\ldotp\ldotp}3]$$
$$\mathit{Fork}\ (\mathit{Fork}\ (\mathit{Leaf}\ 0)\ (\mathit{Leaf}\ 1))\ (\mathit{Fork}\ (\mathit{Leaf}\ 2)\ (\mathit{Leaf}\ 3))$$

Motivation

However, the display of larger data structures is not especially pretty, due to lack of indentation.

$Main\rangle \quad tree\ [0\mathinner{..}9]$
$Fork\ (Fork\ (Fork\ (Leaf\ 0)\ (Leaf\ 1))\ (Fork\ (Leaf\ 2)\ (Fork\ (Leaf\ 3)\ (Leaf\ 4))))\ (Fork\ (Fork\ (Leaf\ 5)\ (Leaf\ 6))\ (Fork\ (Leaf\ 7)\ (Fork\ (Leaf\ 8)\ (Leaf\ 9))))$

Urks.

Motivation

☞ We need a prettier printer.

This talk shows how to define a generic prettier printer and other generic functions.

A generic function is a function that can be instantiated on many data types to obtain data type specific functionality. Examples of generic functions are the functions that can be derived in Haskell, such as $show$, $read$, and '$==$'.

Motivation

Salient features of the approach.

- It's Haskell 98! No extensions, no fancy type systems, no preprocessors required.
- ... so you can play with it, modify it, extend it, adapt it to your needs.

In a nutshell:

- The definition of generic functions works 'as before'.
- A little bit of extra work is required for each newly defined data type.

Warmup: data compression

Let us start with a simpler, albeit related function: a generic data compressor.

For simplicity, we represent a binary string by a list of bits.

$$\begin{array}{lll}
\mathbf{type}\ \mathit{Bin} & = & [\mathit{Bit}] \\
\mathbf{data}\ \mathit{Bit} & = & 0 \mid 1\ \mathbf{deriving}\ (\mathit{Show}) \\
\mathit{bits} & :: & (\mathit{Enum}\ \alpha) \Rightarrow \mathit{Int} \to \alpha \to \mathit{Bin}
\end{array}$$

The function bits encodes an element of an enumeration type using the specified number of bits.

We seek to generalise bits to a function $\mathit{showBin}$ that works for arbitrary types.

Warmup: data compression

Here is an interactive session that illustrates the use of $showBin$ (characters consume 7 bits and integers 16 bits).

$Main\rangle \quad showBin\ (3 :: Int)$
$[1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]$
$Main\rangle \quad showBin\ ([3, 5] :: [Int])$
$[1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]$
$Main\rangle \quad showBin$ `"Lisa"`
$[1, 0, 0, 1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 0, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 0]$

You get the idea . . .

Defining generic functions: elementary types

Implementing $showBin$ so that it works for arbitrary data types seems like a hard nut to crack.

Fortunately, it suffices to define $showBin$ for primitive types and for three elementary types: the one-element type, the binary sum, and the binary product.

$$\begin{array}{l}
\mathbf{data}\ \mathit{Unit} \quad\quad = \mathit{Unit} \\
\mathbf{data}\ \mathit{Plus}\ \alpha\ \beta = \mathit{Inl}\ \alpha \mid \mathit{Inr}\ \beta \\
\mathbf{data}\ \mathit{Pair}\ \alpha\ \beta = \mathit{Pair}\{\mathit{outl} :: \alpha, \mathit{outr} :: \beta\}
\end{array}$$

Why these types?

Defining generic functions: representation types

Because a **data** declaration introduces a type that is isomorphic to a sum of products.

If we know how to compress sums and products, we can compress elements of an arbitrary data type.

In general, we can handle a type σ if we can handle some representation type τ that is isomorphic to σ. ☞ The details of the representation type are largely irrelevant. When programming a generic function it suffices to know the two mappings that witness the isomorphism.

$$\mathbf{data}\ \mathit{Iso}\ \alpha\ \beta = \mathit{Iso}\{\mathit{fromData} :: \beta \to \alpha, \mathit{toData} :: \alpha \to \beta\}$$

Defining generic functions: the signature

Turning to the implementation of $showBin$, we first have to provide the signature of the generic function.

$$\textbf{newtype}\ ShowBin\ \alpha = ShowBin\{\, appShowBin :: \alpha \to Bin\,\}$$

This is not a newtype declaration ().

Data compression does not work for arbitrary types, but only for types that are representable.

$$\begin{aligned} &showBin :: (Rep\ \alpha) \Rightarrow \alpha \to Bin \\ &showBin = appShowBin\ rep \end{aligned}$$

Loosely speaking, we apply the generic function to the type representation rep.

Defining generic functions: the definition itself

The generic function performs a case analysis on types.

$$
\begin{array}{l}
\mathbf{instance}\ \mathit{Generic}\ \mathit{ShowBin}\ \mathbf{where} \\
\quad \begin{array}{lll}
\mathit{unit} & = & \mathit{ShowBin}\ (\lambda x \to [\,]) \\
\mathit{plus} & = & \mathit{ShowBin}\ (\lambda x \to \mathbf{case}\ x\ \mathbf{of}\ \begin{array}[t]{lll} \mathit{Inl}\ l & \to & \mathsf{0} : \mathit{showBin}\ l \\ \mathit{Inr}\ r & \to & \mathsf{1} : \mathit{showBin}\ r) \end{array} \\
\mathit{pair} & = & \mathit{ShowBin}\ (\lambda x \to \mathit{showBin}\ (\mathit{outl}\ x) +\!\!+ \mathit{showBin}\ (\mathit{outr}\ x)) \\
\multicolumn{3}{l}{\mathit{datatype}\ \mathit{descr}\ \mathit{iso}} \\
 & = & \mathit{ShowBin}\ (\lambda x \to \mathit{showBin}\ (\mathit{fromData}\ \mathit{iso}\ x)) \\
\mathit{char} & = & \mathit{ShowBin}\ (\lambda x \to \mathit{bits}\ 7\ \ x) \\
\mathit{int} & = & \mathit{ShowBin}\ (\lambda x \to \mathit{bits}\ 16\ x)
\end{array}
\end{array}
$$

☞ This is not an instance declaration ().

Defining a new type

Recall: a generic function such as $showBin$ can only be instantiated to a representable type.

By default, only the elementary types, $Unit$, $Plus$, and $Pair$, and the primitive types $Char$ and Int are representable.

The declaration below makes the type $Tree$ representable.

$$
\begin{array}{ll}
\textbf{instance}\ (Rep\ \alpha) \Rightarrow Rep\ (Tree\ \alpha)\ \textbf{where} & \\
\quad rep = datatype\ (\texttt{"Leaf"}\ ./\ 1\ .|\ \texttt{"Fork"}\ ./\ 2) & \text{-- syntax} \\
\qquad\qquad\qquad\qquad (Iso\ fromTree\ toTree) & \text{-- semantics}
\end{array}
$$

Defining a new type: specifying the syntax

The expression `"Leaf"` $./\, 1 \,.|$ `"Fork"` $./\, 2$ is of type $DataDescr$ and specifies the syntax of a **data** declaration.

$$
\begin{array}{l}
\mathbf{type}\ Name \qquad\quad = String \\
\mathbf{type}\ Arity \qquad\quad = Int \\
\mathbf{data}\ DataDescr = NoData \\
\qquad\qquad\qquad\quad \mid ConDescr\{name :: Name, \quad arity :: Arity\} \\
\qquad\qquad\qquad\quad \mid Alt \qquad\quad \{getl \ \ :: DataDescr, getr \ :: DataDescr\} \\
\mathbf{infix}\ \ 2\ ./ \\
\mathbf{infixr}\ 1\ .| \\
f\ ./\ n \ = ConDescr\{name = f,\ arity = n\} \\
d_1\ .|\ d_2 = Alt \qquad\quad \{getl \ = d_1, getr \ = d_2\}
\end{array}
$$

Defining a new type: specifying the semantics

The semantics of a **data** declaration is given by an isomorphic type, the structure type, which must be representable.

$$\begin{array}{ll}
\textbf{type}\ \mathit{Tree}'\ \alpha = \mathit{Plus}\ (\mathit{Constr}\ \alpha)\ (\mathit{Constr}\ (\mathit{Pair}\ (\mathit{Tree}\ \alpha)\ (\mathit{Tree}\ \alpha))) & \\
\mathit{fromTree} & :: \mathit{Tree}\ \alpha \to \mathit{Tree}'\ \alpha \\
\mathit{fromTree}\ (\mathit{Leaf}\ x) & = \mathit{Inl}\ (\mathit{Constr}\ x) \\
\mathit{fromTree}\ (\mathit{Fork}\ l\ r) & = \mathit{Inr}\ (\mathit{Constr}\ (\mathit{Pair}\ l\ r)) \\
\mathit{toTree} & :: \mathit{Tree}'\ \alpha \to \mathit{Tree}\ \alpha \\
\mathit{toTree}\ (\mathit{Inl}\ (\mathit{Constr}\ x)) & = \mathit{Leaf}\ x \\
\mathit{toTree}\ (\mathit{Inr}\ (\mathit{Constr}\ (\mathit{Pair}\ l\ r))) & = \mathit{Fork}\ l\ r
\end{array}$$

The type Constr marks the occurrences of constructors.

$$\textbf{newtype}\ \mathit{Constr}\ \alpha = \mathit{Constr}\{\mathit{arg} :: \alpha\}$$

Defining a new type

Haskell's list data type can be treated in a similar manner.

$$
\begin{array}{ll}
\mathit{fromList} & :: [\alpha] \to \mathit{Plus}\ \mathit{Unit}\ (\mathit{Pair}\ \alpha\ [\alpha]) \\
\mathit{fromList}\ [\,] & = \mathit{Inl}\ \mathit{Unit} \\
\mathit{fromList}\ (x : xs) & = \mathit{Inr}\ (\mathit{Pair}\ x\ xs) \\
\mathit{toList} & :: \mathit{Plus}\ \mathit{Unit}\ (\mathit{Pair}\ \alpha\ [\alpha]) \to [\alpha] \\
\mathit{toList}\ (\mathit{Inl}\ \mathit{Unit}) & = [\,] \\
\mathit{toList}\ (\mathit{Inr}\ (\mathit{Pair}\ x\ xs)) & = x : xs
\end{array}
$$

Implementation: type case

Th class *Generic* accommodates the different instances of a generic function.

$$
\begin{array}{lll}
\multicolumn{3}{l}{\textbf{class}\ \mathit{Generic}\ g\ \textbf{where}} \\
\quad \mathit{unit} & :: & g\ \mathit{Unit} \\
\quad \mathit{plus} & :: (\mathit{Rep}\ \alpha, \mathit{Rep}\ \beta) \Rightarrow & g\ (\mathit{Plus}\ \alpha\ \beta) \\
\quad \mathit{pair} & :: (\mathit{Rep}\ \alpha, \mathit{Rep}\ \beta) \Rightarrow & g\ (\mathit{Pair}\ \alpha\ \beta) \\
\quad \mathit{datatype} & :: (\mathit{Rep}\ \alpha) \Rightarrow \mathit{DataDescr} \to \mathit{Iso}\ \alpha\ \beta \to & g\ \beta \\
\quad \mathit{char} & :: & g\ \mathit{Char} \\
\quad \mathit{int} & :: & g\ \mathit{Int} \\
\quad \mathit{list} & :: (\mathit{Rep}\ \alpha) \Rightarrow & g\ [\alpha] \\
\quad \mathit{constr} & :: (\mathit{Rep}\ \alpha) \Rightarrow & g\ (\mathit{Constr}\ \alpha) \\
\quad \mathit{list} & = \mathit{datatype}\ (\texttt{"[]"}\ ./\ 0\ .|\ \texttt{":"}\ ./\ 2)\ (\mathit{Iso}\ \mathit{fromList}\ \mathit{toList}) & \\
\quad \mathit{constr} & = \mathit{datatype}\ (\texttt{"Constr"}\ ./\ 1)\ \quad (\mathit{Iso}\ \mathit{arg}\ \mathit{Constr}) &
\end{array}
$$

The class abstracts over the type constructor g, the type of a generic function.

Implementation: type representation

What does it mean for a type to be representable?

For our purposes, this simply means that we can instantiate a generic function to that type.

So an intriguing choice is to identify type representations with generic functions.

$$
\begin{array}{l}
\mathbf{class}\ \mathit{Rep}\ \alpha\ \mathbf{where} \\
\quad \mathit{rep} :: (\mathit{Generic}\ g) \Rightarrow g\ \alpha
\end{array}
$$

☞ The type variable g is universally quantified: the type representation must work for all instances of g.

Implementation: type representation

A type is representable if we can instantiate a generic function to that type.

$$
\begin{array}{llll}
\textbf{instance} & & \mathit{Rep}\ \mathit{Unit} & \textbf{where} \\
\quad \mathit{rep} = \mathit{unit} & & & \\
\textbf{instance} & (\mathit{Rep}\ \alpha, \mathit{Rep}\ \beta) \Rightarrow & \mathit{Rep}\ (\mathit{Plus}\ \alpha\ \beta) & \textbf{where} \\
\quad \mathit{rep} = \mathit{plus} & & & \\
\textbf{instance} & (\mathit{Rep}\ \alpha, \mathit{Rep}\ \beta) \Rightarrow & \mathit{Rep}\ (\mathit{Pair}\ \alpha\ \beta) & \textbf{where} \\
\quad \mathit{rep} = \mathit{pair} & & & \\
\textbf{instance} & & \mathit{Rep}\ \mathit{Char} & \textbf{where} \\
\quad \mathit{rep} = \mathit{char} & & & \\
\textbf{instance} & & \mathit{Rep}\ \mathit{Int} & \textbf{where} \\
\quad \mathit{rep} = \mathit{int} & & & \\
\textbf{instance} & (\mathit{Rep}\ \alpha) \Rightarrow & \mathit{Rep}\ [\alpha] & \textbf{where} \\
\quad \mathit{rep} = \mathit{list} & & & \\
\textbf{instance} & (\mathit{Rep}\ \alpha) \Rightarrow & \mathit{Rep}\ (\mathit{Constr}\ \alpha) & \textbf{where} \\
\quad \mathit{rep} = \mathit{constr} & & &
\end{array}
$$

Implementation: type representation

The type of rep is quite remarkable:

$$rep :: (Rep\ \alpha, Generic\ g) \Rightarrow g\ \alpha$$

In a sense, rep can be seen as the mother of all generic functions.

A generic prettier printer

$$pretty :: (Rep\ \alpha) \Rightarrow \alpha \to Doc$$
$$pretty = pretty'\ NoData$$

The helper function $pretty'$ is defined generically:

$$\mathbf{newtype}\ Pretty'\ \alpha = Pretty'\{\, applyPretty' :: DataDescr \to \alpha \to Doc\}$$
$$pretty' :: (Rep\ \alpha) \Rightarrow DataDescr \to \alpha \to Doc$$
$$pretty' = applyPretty'\ rep$$

A generic prettier printer

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Generic}\ \mathit{Pretty}'\ \textbf{where} \\
\quad \begin{array}{lll}
\mathit{unit} & = \mathit{Pretty}'\ (\lambda d\ x \rightarrow \mathit{empty}) \\
\mathit{plus} & = \mathit{Pretty}'\ (\lambda d\ x \rightarrow \textbf{case}\ x\ \textbf{of}\ \mathit{Inl}\ \ l\ \rightarrow \mathit{pretty}'\ (\mathit{getl}\ \ d)\ l \\
 & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{Inr}\ r \rightarrow \mathit{pretty}'\ (\mathit{getr}\ d)\ r) \\
\mathit{pair} & = \mathit{Pretty}'\ (\lambda d\ x \rightarrow \mathit{pretty}\ (\mathit{outl}\ x)\ \langle\rangle\ \mathit{line}\ \langle\rangle\ \mathit{pretty}\ (\mathit{outr}\ x)) \\
\mathit{char} & = \mathit{Pretty}'\ (\lambda d\ x \rightarrow \mathit{prettyChar}\ x) \\
\mathit{int} & = \mathit{Pretty}'\ (\lambda d\ x \rightarrow \mathit{prettyInt}\ \ \ x) \\
\mathit{list} & = \mathit{Pretty}'\ (\lambda d\ x \rightarrow \mathit{prettyl}\ \mathit{pretty}\ x) \\
\multicolumn{2}{l}{\mathit{datatype}\ \mathit{descr}\ \mathit{iso}} \\
 & = \mathit{Pretty}'\ (\lambda d\ x \rightarrow \mathit{pretty}'\ \mathit{descr}\ (\mathit{fromData}\ \mathit{iso}\ x)) \\
\mathit{constr} & = \mathit{Pretty}'\ (\lambda d\ x \rightarrow \textbf{if}\ \mathit{arity}\ d == 0\ \textbf{then} \\
 & \qquad\qquad\qquad\qquad\quad \mathit{text}\ (\mathit{name}\ d) \\
 & \qquad\qquad\qquad\qquad \textbf{else} \\
 & \qquad\qquad\qquad\qquad\quad \mathit{group}\ (\mathit{nest}\ 1\ (\\
 & \qquad\qquad\qquad\qquad\qquad \mathit{text}\ \texttt{"("}\ \langle\rangle\ \mathit{text}\ (\mathit{name}\ d)\ \langle\rangle\ \mathit{line} \\
 & \qquad\qquad\qquad\qquad\qquad \langle\rangle\ \mathit{pretty}\ (\mathit{arg}\ x)\ \langle\rangle\ \mathit{text}\ \texttt{")"}))) \\
\end{array}
\end{array}
$$

A generic prettier printer

For completeness:

$$
\begin{aligned}
&\mathit{prettyl} && :: (\alpha \rightarrow \mathit{Doc}) \rightarrow ([\alpha] \rightarrow \mathit{Doc}) \\
&\mathit{prettyl}\ p\ [\,] && = \mathit{text}\ \texttt{"[]"} \\
&\mathit{prettyl}\ p\ (a : \mathit{as}) && = \mathit{group}\ (\mathit{nest}\ 1\ (\mathit{text}\ \texttt{"["} \langle\rangle\ p\ a\ \langle\rangle\ \mathit{rest}\ \mathit{as})) \\
&\quad \textbf{where}\ \mathit{rest}\ [\,] && = \mathit{text}\ \texttt{"]"} \\
&\qquad\qquad\ \ \mathit{rest}\ (x : \mathit{xs}) && = \mathit{text}\ \texttt{","} \langle\rangle\ \mathit{line}\ \langle\rangle\ p\ x\ \langle\rangle\ \mathit{rest}\ \mathit{xs}
\end{aligned}
$$

Extensions and variations

- Additional type cases (extending the $Generic$ class).
- Default type cases (using default methods).
- Mutually recursive definitions (easy).
- Generic functions on type constructors ($size :: (FRep\ \varphi) \Rightarrow \varphi\ \alpha \rightarrow Int$).
- Abstraction over two type parameters (map).
- Multiple representation types.

Conclusion

- Pro: It's Haskell 98!
- Con: Generic data types are out of reach.
- Con: Not suitable for a general-purpose library.
- Without type classes: you need records with polymorphic components.


Ceci n'est pas une pipe.