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.

```haskell
data Tree α = Leaf α | Fork (Tree α) (Tree α)
deriving (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 `[α] → Tree α`).

```haskell
Main> tree [0..3]
Fork (Fork (Leaf 0) (Leaf 1)) (Fork (Leaf 2) (Leaf 3))
```
Motivation

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

```
Main) tree [0..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.
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.
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.

```haskell
type Bin = [Bit]
data Bit = 0 | 1 deriving (Show)
bites :: (Enum α) ⇒ Int → α → Bin
```

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

We seek to generalise `bits` to a function `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).

```haskell
Main⟩ showBin (3 :: Int)
[1,1,0,0,0,0,0,0,0,0,0,0,0,0,0]
Main⟩ showBin ([3, 5] :: [Int])
[1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
Main⟩ showBin "Lisa"
[1,0,0,1,1,0,0,1,1,1,0,0,1,0,1,1,1,1,0,0,1,0,1,1,1,1,0,0,0,0,0,1,1,0]
```

You get the idea . . .
Implementing \textit{showBin} so that it works for arbitrary data types seems like a hard nut to crack.

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

\begin{verbatim}
data Unit = Unit
data Plus α β = Inl α | Inr β
data Pair α β = Pair{ outl :: α, outr :: β}
\end{verbatim}

Why these 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 \( \sigma \) if we can handle some representation type \( \tau \) that is isomorphic to \( \sigma \). 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.

```haskell
data Iso \( \alpha \ \beta \) = Iso\{ fromData :: \( \beta \to \alpha \), toData :: \( \alpha \to \beta \) \}
```
Turning to the implementation of \textit{showBin}, we first have to provide the signature of the generic function.

\begin{verbatim}
newtype ShowBin α = ShowBin{ appShowBin :: α → Bin }
\end{verbatim}

This is not a newtype declaration ( ).

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

\begin{verbatim}
showBin :: (Rep α) ⇒ α → Bin
showBin = appShowBin rep
\end{verbatim}

Loosely speaking, we apply the generic function to the type representation \textit{rep}.
The generic function performs a case analysis on types.

```haskell
instance Generic ShowBin where
    unit = ShowBin (\x -> [])
    plus = ShowBin (\x -> case x of Inl l -> 0 \showBin l
                      Inr r -> 1 \showBin r)
    pair = ShowBin (\x -> showBin (outl x) ++ showBin (outr x))
    datatype descr iso
        = ShowBin (\x -> showBin (fromData iso x))
    char = ShowBin (\x -> bits 7 x)
    int  = ShowBin (\x -> bits 16 x)
```

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.

```haskell
instance (Rep α) ⇒ Rep (Tree α) where
  rep = datatype ("Leaf" ./ 1 .| "Fork" ./ 2) -- syntax
  (Iso fromTree toTree) -- semantics
```
Defining a new type: specifying the syntax

The expression "Leaf" ./ 1 .| "Fork" ./ 2 is of type \texttt{DataDescr} and specifies the syntax of a \texttt{data} declaration.

\begin{verbatim}
type Name = String
type Arity = Int
data DataDescr = NoData |
    ConDescr {name :: Name, arity :: Arity}
    Alt {getl :: DataDescr, getr :: DataDescr}

infix 2 ./
infixr 1 .|

f ./ n = ConDescr {name = f, arity = n}
d_1 .| d_2 = Alt {getl = d_1, getr = d_2}
\end{verbatim}
The semantics of a `data` declaration is given by an isomorphic type, the `structure` type, which must be representable.

```
type Tree' α = Plus (Constr α) (Constr (Pair (Tree α) (Tree α)))
fromTree :: Tree α → Tree' α
fromTree (Leaf x) = Inl (Constr x)
fromTree (Fork l r) = Inr (Constr (Pair l r))
toTree :: Tree' α → Tree α
toTree (Inl (Constr x)) = Leaf x
toTree (Inr (Constr (Pair l r))) = Fork l r
```

The type `Constr` marks the occurrences of constructors.

```
newtype Constr α = Constr{ arg :: α }
```
Defining a new type

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

\[
\begin{align*}
\text{fromList} &: [\alpha] \rightarrow \text{Plus Unit (Pair } \alpha [\alpha]) \\
\text{fromList } [] &= \text{Inl Unit} \\
\text{fromList } (x : xs) &= \text{Inr (Pair } x xs) \\
\text{toList} &: \text{Plus Unit (Pair } \alpha [\alpha]) \rightarrow [\alpha] \\
\text{toList } (\text{Inl Unit}) &= [] \\
\text{toList } (\text{Inr (Pair } x xs)) &= x : xs
\end{align*}
\]
The class *Generic* accommodates the different instances of a generic function.

```plaintext
class Generic g where
  unit :: g Unit
  plus :: (Rep α, Rep β) ⇒ g (Plus α β)
  pair :: (Rep α, Rep β) ⇒ g (Pair α β)
  datatype :: (Rep α) ⇒ DataDescr → Iso α β → g β
  char :: g Char
  int :: g Int
  list :: (Rep α) ⇒ g [α]
  constr :: (Rep α) ⇒ g (Constr α)

list = datatype ("[]" ./ 0 ./ ":" ./ 2) (Iso fromList toList)
constr = datatype ("Constr" ./ 1) (Iso arg Constr)
```

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

```haskell
class Rep α where
  rep :: (Generic g) ⇒ g α
```

 반드시 The type variable \( g \) is universally quantified: the type representation must work for **all** instances of \( g \).
A type is representable if we can instantiate a generic function to that type.

<table>
<thead>
<tr>
<th>instance</th>
<th>Rep Unit</th>
<th>where</th>
</tr>
</thead>
<tbody>
<tr>
<td>rep = unit</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>instance</th>
<th>Rep (Plus α β)</th>
<th>where</th>
</tr>
</thead>
<tbody>
<tr>
<td>rep = plus</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>instance</th>
<th>Rep (Pair α β)</th>
<th>where</th>
</tr>
</thead>
<tbody>
<tr>
<td>rep = pair</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>instance</th>
<th>Rep Char</th>
<th>where</th>
</tr>
</thead>
<tbody>
<tr>
<td>rep = char</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>instance</th>
<th>Rep Int</th>
<th>where</th>
</tr>
</thead>
<tbody>
<tr>
<td>rep = int</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>instance</th>
<th>Rep [α]</th>
<th>where</th>
</tr>
</thead>
<tbody>
<tr>
<td>rep = list</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>instance</th>
<th>Rep (Constr α)</th>
<th>where</th>
</tr>
</thead>
<tbody>
<tr>
<td>rep = constr</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The type of \( rep \) is quite remarkable:

\[
rep :: (\text{Rep } \alpha, \text{Generic } g) \Rightarrow g \alpha
\]

In a sense, \( rep \) can be seen as the mother of all generic functions.
A generic prettier printer

pretty :: (Rep α) ⇒ α → Doc
pretty = pretty' NoData

The helper function \( \text{pretty}' \) is defined generically:

newtype Pretty' α = Pretty'\{ applyPretty' :: DataDescr → α → Doc \}
pretty' :: (Rep α) ⇒ DataDescr → α → Doc
pretty' = applyPretty' rep
instance Generic Pretty' where

unit  = Pretty' (λd x → empty)
plus  = Pretty' (λd x → case x of Inl l → pretty' (getl d) l
                      Inr r → pretty' (getr d) r)
pair  = Pretty' (λd x → pretty (outl x) ⟨⟩ line ⟨⟩ pretty (outr x))
char  = Pretty' (λd x → prettyChar x)
int   = Pretty' (λd x → prettyInt x)
list  = Pretty' (λd x → prettyl pretty x)
datatype descr iso
       = Pretty' (λd x → pretty' descr (fromData iso x))
constr = Pretty' (λd x → if arity d == 0 then
                      text (name d)
                else
                      group (nest 1 (text "(" ⟨⟩ text (name d) ⟨⟩ line
                      ⟨⟩ pretty (arg x) ⟨⟩ text ")")"))))
**A generic prettier printer**

For completeness:

\[
\text{prettyl} \quad :: \quad (\alpha \to \text{Doc}) \to ([\alpha] \to \text{Doc}) \\
\text{prettyl} \quad p \quad [] \quad = \quad \text{text} \quad "\[\]"
\]

\[
\text{prettyl} \quad p \quad (a : as) \quad = \quad \text{group} \quad (\text{nest} \quad 1 \quad (\text{text} \quad "[" \quad \langle\rangle \quad p \quad a \quad \langle\rangle \quad \text{rest} \quad as\rangle))
\]

**where**

\[
\begin{align*}
\text{rest} \quad [] & \quad = \quad \text{text} \quad ""]" \\
\text{rest} \quad (x : xs) & \quad = \quad \text{text} \quad "," \quad \langle\rangle \quad \text{line} \quad \langle\rangle \quad p \quad x \quad \langle\rangle \quad \text{rest} \quad xs
\end{align*}
\]
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 \((\text{size} :: (FRep \varphi) \Rightarrow \varphi \alpha \rightarrow \text{Int})\).
- Abstraction over two type parameters \((\text{map})\).
- 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.