A GENERIC PROGRAMMING EXTENSION FOR HASKELL

RALF HINZE

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

October, 1999

Examples, examples, examples . . .

Generic programming (aka structural polymorphism, type parametric programming, shape polymorphism, polytypic programming) is about making programming easier and more economical.

We will introduce the generic programming extension through a series of examples (common theme: encoding values as bit strings).

☞ For the nitty-gritty look at the proceedings.

☞ For the theoretical background look at "A new approach to generic functional programming", to appear in POPL'00; a preliminary version of this paper is published as a technical report (IAI-TR-99-9).

Binary encoding (the non-generic way)

Binary data (`Bin` was primitive in Haskell 1.2).

```
type Bin =  [Bit]
data Bit =  O | I
```

Encoding strings (we assume that `encChar :: Char -> Bin` is given).

```
encString          :: String -> Bin
encString []       =  O : []
encString (c : s)  =  I : encChar c ++ encString s
```


Binary encoding (the non-generic way)

Leaf trees.

```
data Tree a =  Leaf a | Fork (Tree a) (Tree a)
```

Encoding binary leaf trees.

```
encTree             :: Tree Bin -> Bin
encTree (Leaf bs)   =  O : bs
encTree (Fork l r)  =  I : encTree l ++ encTree r
```

NB. Alternatively, one could define an encoder of type

```
encTree :: (a -> Bin) -> (Tree a -> Bin)
```

Example encodings

```
encString "hello"
==> 1000101101101001101001101101001101101111101100

encTree (fmap encString (
    Fork (Leaf "")
         (Fork (Leaf "") (Leaf "")) ))
==> 10010000
```

NB. `Tree` must be an instance of `Functor`.

NB. Characters are encoded using 8 bits (LSB first).

Programming contest ;-)

Write encoders for

`Rose [] String`, `Fix (Base Char)` and `Sequ (Sequ String)`

where `Rose`, `Fix`, `Base` and `Sequ` are given by

```
data Rose f a =  Branch a (f (Rose f a))

data Fix f    =  In (f (Fix f))
data Base a b =  Nil | Cons a b

data Sequ a   =  Empty | Zero (Sequ (a, a))
                       | One a (Sequ (a, a))
```


Résumé

Writing encoders is mostly straightforward except, perhaps, for the types on the previous slide.

But, it is painful to do so.

Especially, since it is crystal clear what to do:

- emit bits for the constructors and
- catenate encodings of the components of a constructor.

Generic definitions

Never fear. Generic functions save the day.

Using a generic definition we can program `encode` once and for all times.

Main difference:

- ad-hoc encoders are defined by induction on the structure of *values*,
- a generic function is defined by induction on the structure of *types*.

Modelling datatypes

A Haskell `data` declaration defines a sum of products.

```
data [a]     =  [] | a : [a]
data Tree a =  Leaf a | Fork (Tree a) (Tree a)
```

Making sums and products explicit and emitting constructor names we obtain

```
[a]     =  1 + (a * [a])
Tree a =  a + (Tree a * Tree a)
```


Modelling datatypes

The types `1`, '`+`', and '`*`' serve as abbreviations for the predefined types `()`, `Either`, and `(,)` given by

```
data ()          =  ()
data Either a b =  Left a | Right b
data (a, b)     =  (a, b)
```


A generic encoding function

```
encode(t)                     :: t -> Bin
encode(1)      ()             =  []
encode(Char)   x              =  encChar x
encode(a + b)  (Left  x)      =  O : encode(a) x
encode(a + b)  (Right y)      =  I : encode(b) y
encode(a * b)  (x, y)         =  encode(a) x
                              ++ encode(b) y
```

NB. Type signatures are mandatory for generic values.

Example encodings

```
encode(String) "hello"
==> 10001011011010011010011011010011011011111101100

encode(Tree String) (
    Fork (Leaf "")
         (Fork (Leaf "") (Leaf "")) )
==> 10010000

encode([Tree Bool]) (
    [Leaf True, Fork (Leaf False) (Leaf True)] )
==> 1011100010
```

NB. The type parameter of `encode` must not be omitted.

Improving the efficiency

```
encodes(t)                   :: t -> (Bin -> Bin)
encodes(1)      ()           =  id
encodes(Char)   x            =  (encChar x ++)
encodes(a + b)  (Left  x)    =  (O :) . encodes(a) x
encodes(a + b)  (Right y)    =  (I :) . encodes(b) y
encodes(a * b)  (x, y)       =  encodes(a) x
                             .  encodes(b) y
```


A generic decoding function

```
decode(t)                :: Bin -> (t, Bin)
decode(1)      bs        =  ((), bs)
decode(Char)   bs        =  decChar bs
decode(a + b)  []        =  error "decode"
decode(a + b)  (O : bs)  =  let (x, cs) = decode(a) bs
                            in (Left x, cs)
decode(a + b)  (I : bs)  =  let (y, cs) = decode(b) bs
                            in (Right y, cs)
decode(a * b)  bs        =  let (x, cs) = decode(a) bs
                                (y, ds) = decode(b) cs
                            in  ((x, y), ds)
```


Dumping values to files

A generic value is either defined inductively or in terms of other generic values.

```
dump(t) :: FilePath -> t -> IO ()
dump(t) fpath x
  =  writeFile fpath
  $  map toChar       -- convert 16 bit word into char
  $  rsplit 16        -- split list into 16 bit words
  $  encode(t) x
```


Ad-hoc definitions

Ad-hoc definitions specify exceptions to the ‘general rule’.

Here is a more space-efficient scheme for encoding lists over some base type a (we assume that `encInt :: Int -> Bin` is given).

```
encode([a]) xs =  encInt (length xs)
               ++ concatMap (encode(a)) xs
```

Blending generic definitions with type classes

Generic definitions can be used to give default definitions for class methods.

```
class Binary a where
    showBin :: a -> Bin -> Bin
    readBin :: Bin -> (a, Bin)

    showBin =  encodes(a)
    readBin =  decode(a)
```

NB. `Binary` was part of Haskell 1.2.

Deriving instances of `Binary`

We may now write

```
data Tree a =  Leaf a | Fork (Tree a) (Tree a)
               deriving (Binary)
```

which generates a `(Binary a) => Binary (Tree a)` instance.

NB. In Haskell 98 `deriving` is restricted to a few predefined classes. Furthermore, `deriving` does not work for all user-defined types.

Digression: accessing constructor names

For defining generic pretty printers and parsers (eg `show` and `read`) we need to know the constructor names.

```
Tree a =  Leaf (a) + Fork (Tree a * Tree a)
```

Generic definitions may use an additional type pattern: `c a` where `c` is a *value* variable of type `Con` and `a` is a *type* variable.

```
data Con -- abstract
showCon :: Con -> String    -- primitive
```


Digression: pretty printing

Example. Display a value in a tree-like fashion.

```
layn(Tree Int) 0 (
    Fork (Leaf 1)
         (Fork (Leaf 2) (Leaf 3)) )
==> Fork
      Leaf
        1
      Fork
        Leaf
          2
        Leaf
          3
```

Digression: pretty printing

```
layn(t)                    :: Int -> t -> String
layn(1)      i x           =  ""
layn(Int)    i x           =  line i (showInt x)
layn(a + b) i (Left x)    =  layn(a) i x
layn(a + b) i (Right y)   =  layn(b) i y
layn(c a)    i x           =  line i (showCon c)
                           ++ layn(a) (i + 2) x
layn(a * b) i (x, y)      =  layn(a) i x ++ "\n"
                           ++ layn(b) i y

line      :: Int -> String -> String
line i s =  replicate i ' ' ++ s ++ "\n"
```


Binary encoding: separating shape and contents

Observation: for values of type `f a` it is sometimes advantageous to encode the shape, `f ()`, and the contents, `[a]`, separately.

```
separate(f)    :: f a -> (f (), [a])
separate(f) x =  ( map(f) (const ()) x
                 , flatten(f) x)
```

☞ `separate` is indexed by a *type constructor* (of kind `* -> *`) rather than a type.

Flattening

```
flatten(t)                            :: t u -> [u]
flatten(\a.a)          x            =  [x]
flatten(\a.1)          x            =  []
flatten(\a.Char)       x            =  []
flatten(\a.f a + g a) (Left x)     =  flatten(f) x
flatten(\a.f a + g a) (Right y)    =  flatten(g) y
flatten(\a.f a * g a) (x, y)       =  flatten(f) x
                                     ++ flatten(g) y
```

☞ The type argument of `flatten` ranges over type constructors of kind `* -> *`. Therefore, the type patterns denote functions.

A generic mapping function

```
map(t)        :: (u -> v) -> (t u -> t v)
map(Id)   m x =  m x
map(K s) m x =  x
map(\a.f a + g a) m (Left x)
              =  Left (map(f) m x)
map(\a.f a + g a) m (Right y)
              =  Right (map(g) m y)
map(\a.f a * g a) m (x, y)
              =  (map(f) m x, map(g) m y)
```

NB. `Id` abbreviates `\a.a` and `K s` abbreviates `\a.s`.

Mutually recursive definitions

We can even extend map to include function spaces.

```
map(t)        :: (u -> v) -> (t u -> t v)
...
map(\a.f a -> g a) m phi
              =  map(g) m . phi . comap(f) m
```

where comap(f) is given by ...

Mutually recursive definitions

```
comap(t)        :: (u -> v) -> (t v -> t u)
comap(K s) m x =  x
comap(\a.f a + g a) m (Left x)
               =  Left (comap(f) m x)
comap(\a.f a + g a) m (Right y)
               =  Right (comap(g) m y)
comap(\a.f a * g a) m (x, y)
               =  (comap(f) m x, comap(g) m y)
comap(\a.f a -> g a) m phi
               =  comap(g) m . phi . map(f) m
```

☞ comap(Id) is undefined, ie map is only defined for covariant functors.

Blending generic definitions with type classes

```
class Functor t where
  map   :: (u -> v) -> (t u -> t v)
  comap :: (u -> v) -> (t v -> t u)

  map(Id)  m x =  m x
  map(K s) m x =  x
  ...
```


Deriving instances of Functor

We may now write

```
data Tree a   =  Leaf a | Fork (Tree a) (Tree a)
                 deriving (Functor)
```

which generates a Functor Tree instance.

Concluding remarks

- ⊕ Generic definitions have been used to specify Haskell's derivable methods: `(==)`, `compare`, `show`, `read`, etc.
- ⊕ Generic values can be specialized to arbitrary Haskell datatypes.
- ⊖ There is a considerable overlap with type classes.
- ⊖ The generic programming extension has not been implemented yet.