Generic function extension in Haskell

Ralf Lämmel, Vrije University
Simon Peyton Jones, Microsoft Research

What’s function extension?

- You have
  - A generic query
    `gen :: Data a => a -> R`
  - A type-specific query
    `spec :: T -> R`
- You want a generic function which behaves like **spec** on values of type T, and like **gen** on all other values
  `` gen `extQ` spec :: Data a => a -> R ``

Example

```
gshow, gshow’ :: Data a => a -> String
gshow = gshow’ `extQ` showString

showString :: String -> String
showString s= “\”” ++ s ++ “\””

gshow’ x = “(“ ++ show (toConstr x)
               ++ concat (gmapQ gshow x)
               ++ “)”
-- gmapQ :: Data a
--       => (forall b. Data b => b->r)
--       -> a -> [r]
```

Defining extQ

```
extQ :: (Typeable a, Typeable b)
      => (a->r) -> (b->r) -> a -> r
extQ gen spec x
  = case (cast x) of
       Just x’ -> spec x’
       Nothing -> gen x


cast :: (Typeable a, Typeable b)
      => a -> Maybe b
```

Type safe cast

```
cast :: (Typeable a, Typeable b)
      => a -> Maybe b

ghci> (cast 'a') :: Maybe Char
Just 'a'
ghci> (cast 'a') :: Maybe Bool
Nothing
ghci> (cast True) :: Maybe Bool
Just True
```

Implementing cast

```
data TypeRep
instance Eq TypeRep
mkRep :: String -> [TypeRep] -> TypeRep

class Typeable a where
  typeOf :: a -> TypeRep

instance Typeable Int where
  typeOf i = mkRep "Int" []
```

An Int, perhaps

Guaranteed not to evaluate its argument

Implementing cast

```
class Typeable a where
  typeOf :: a -> TypeRep


instance (Typeable a, Typeable b)
      => Typeable (a,b) where
  typeOf p = mkTyConApp
                  (mkTyCon "(,)")
                  [ta,tb]
     where
          ta = typeOf (fst p)
          tb = typeOf (snd p)
```

Implementing cast

```
cast :: (Typeable a, Typeable b)
      => a -> Maybe b
cast x = r
  where
    r = if typeOf x == typeOf (get r)
          then Just (unsafeCoerce x)
          else Nothing

    get :: Maybe a -> a
    get x = undefined
```

Implementing cast

- In GHC:
  - Typeable instances are generated automatically by the compiler for any data type
  - The definition of cast is in a library
- Then cast is sound
- Bottom line: cast is best thought of as a language extension, but it is an easy one to implement. All the hard work is done by type classes

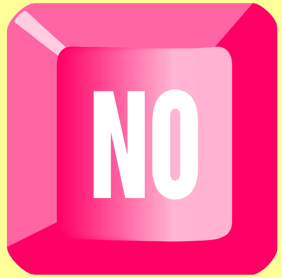
Generalisation 1
Leibnitz equality

Back to function extension

- You have
  - A generic **transform**
    `gen :: Data a => a -> a`
  - A type-specific transform
    `spec :: T -> T`
- You want a generic function which behaves like **spec** on values of type T, and like **gen** on all other values
  `` gen `extT` spec :: Data a => a -> a ``

Defining extT

```
extT :: (Typeable a, Typeable b)
      => (a->a) -> (b->b) -> a -> a
extT gen spec x
  = case (cast x) of
      Just x' -> spec x
      Nothing -> gen x
```



Wrong result type!

Defining extT

Instead, cast spec

```
extT :: (Typeable a, Typeable b)
     => (a->a) -> (b->b) -> a -> a
extT gen spec x
  = case (cast spec) of
      Just spec’ -> spec’ x
      Nothing    -> gen x
```

Compares type rep for (a->a) with that ofr (b->b), when all that is necessary is to compare a and b.

What about extM?

```
extM :: (Typeable a, Typeable b)
     => (a->m a)-> (b->m b)-> a -> m a
```

Need to compare type rep for (a->m a) with that for (b->m b), so we actually need.

```
extM :: (Typeable a, Typeable b,
         Typeable (m a), Typeable (m b))
     => (a->m a)-> (b->m b)-> a -> m a
```

Sigh: complex, and adds bogus constraints

Use Leibnitz equality!

```
gcast :: (Typeable a, Typeable b)
      => c a -> Maybe (c b)
```

- Abstracts over arbitrary type constructor c
- Now extM is simple...

Defining extM

```
newtype M m a = M (a -> m a)

extM :: (Typeable a, Typeable b)
      => (a->m a)-> (b->m b)-> a -> m a
extM gen spec x
  = case gcast (M spec) of
       Just (M spec’) -> spec’ x
       Nothing        -> gen x
```

- No problem with the ‘m’ part
- Compares type reps only for a, b.

Generalisation 2 polymorphic function extension

Polymorphic function extension

- So far we have only monomorphic function extension. Given

  show :: Data a => a -> String

  we can extend it with a type-specific function

  showInt :: Int -> String

  or showFoogle :: Tree [Int] -> String

- But we **can't** extend it with a polymorphic function:

  showList :: Data a => [a] -> String

Wanted

```
extQ1 :: (Data a, Typeable1 t)
  => (a -> r)
  -> (forall b. Data b => t b -> r)
  -> (a -> r)
```

compare previous:

```
extQ    :: (Typeable a, Typeable b)
        => (a -> r)
        -> (b -> r)
        -> (a -> r)
```

Then we could do:

```
gshow, gshow’ :: Data a => a -> String
gshow = gshow’ `extQ` showString
               `extQ1` showList
showString :: String -> String
showString s= “\”” ++ s ++ “\””

showList :: Data a => [a] -> String
showList xs = “[“
       ++ intersperse “,” (map gshow xs)
       ++ “]”
gshow’ x = ...unchanged...
```

Digression 1

What is Typeable1?

Typeable

a :: *

```
class Typeable a where
   typeOf :: a -> TypeRep


instance Typeable a => Typeable [a] where
  typeOf _ = mkTyConApp
                 (mkTyCon "[]")
                 (typeOf (undefined::a))
```

Proxy type argument

Typeable1

t::*->*

```
class Typeable1 t where
  typeOf :: t a -> TypeRep

instance Typeable1 [] where
  typeOf _ = mkTyConApp (mkTyCon "[]") []
```

But what about Typeable [a]?

Typeable1

t::*->*

```
class Typeable1 t where
  typeOf1 :: t a -> TypeRep

instance Typeable1 [] where
  typeOf1 _ = mkTyConApp (mkTyCon “[]”) []

instance (Typeable1 t, Typeable a)
             => Typeable (t a) where
  typeOf _ = mkAppTy
               (typeOf1 (undefined::t a))
               (typeOf  (undefined::a))
```

Sigh

- Typeable2 (t :: *->*->*)
- Typeable3 (t :: *->*->*->*)
- ...how far?
- and what about
  Typeable1_2 (t :: (*->*) -> *)
- etc

Wanted: kind polymorphism

```
class Typeable (a::k) where
  typeOf :: a -> TypeRep
```

No! Only works if k = *

Wanted: kind polymorphism

```
class Typeable (a::k) where
  typeOf :: Proxy a -> TypeRep

-- typeOf :: forall k. forall a:k.
--                Proxy a -> TypeRep

data Proxy (a::k)
-- Proxy :: forall k. k -> *
```

Wanted: kind polymorphism

```
class Typeable (a::k) where
  typeOf :: Proxy a -> TypeRep

instance Typeable [] where
  typeOf _ = mkTyConApp (mkTyCon “[]”) []

instance (Typeable t, Typeable a)
             => Typeable (t a) where
  typeOf _ = mkAppTy
               (typeOf (undefined::Proxy t))
               (typeOf (undefined::Proxy a))
```

End of digression 1

Back to extQ1

```
extQ1 :: (Data a, Typeable1 t)
  => (a -> r)
  -> (forall b. Data b => t b -> r)
  -> (a -> r)
```

compare previous:

```
extQ    :: (Typeable a, Typeable b)
        => (a -> r)
        -> (b -> r)
        -> (a -> r)
```

Recall extQ implementation

```
extQ :: (Typeable a, Typeable b)
      => (a->r) -> (b->r) -> a -> r
extQ gen spec x
  = case gcast (Q spec) of
       Just (Q spec') -> spec' x'
       Nothing        -> gen x

newtype Q r a = Q (a->r)

gcast :: (Typeable a, Typeable b)
       => c a -> Maybe (c b)
```

Implementation of extQ1

```
extQ1 :: (Data a, Typeable1 t)
  => (a->r)-> (forall b.Data b => t b -> r)
  -> (a->r)
extQ1 gen spec x
  = case dataCast1 (Q spec) of
         Just (Q spec’) -> spec’ x
         Nothing        -> gen x
newtype Q r a = Q (a->r)


dataCast1 :: (Data a, Typeable1 t)
  => (forall b. Data b => c (t b))
  -> Maybe (c a)
```

Implemen

The $64M question: How does dataCast1 get the (Data b) dictionary to pass to spec?

```
extQ1 :: (Data a, T
  => (a->r)-> (fora
  -> (a->r)
extQ1 gen spec x
  = case dataCast1 (Q spec) of
         Just (Q spec') -> spec' x
         Nothing        -> gen x
newtype Q r a = Q (a->r)

dataCast1 :: (Data a, Typeable1 t)
  => (forall b. Data b => c (t b))
  -> Maybe (c a)
```

The stunning blow!

- Answer: dataCast1 is a method of Data

```
class Typeable a => Data a where
  ...
  dataCast1 :: Typeable1 t
        => forall b. Data b => c (t b))
        -> Maybe (c a)
```

Instances are trivial

```
instance Data a => Data [a] where
  ...
  dataCast1 f = gcast1 f

gcast1 :: (Typeable1 t, Typeable1 s)
       => c (t a)
       -> Maybe (c (s a))
```

Instances are trivial

```
instance (da:Data a) => Data [a] where
  ...
  dataCast1 (tt:Typeable1 t) f
        = gcast1 (tt,tList) (f da)

tList :: Typeable1 List

gcast1 :: (Typeable1 t, Typeable1 s)
       => c (t a)
       -> Maybe (c (s a))
```

Gcast1 is just like cast

```
gcast1 :: (Typeable1 t, Typeable1 s)
       => c (t a)
       -> Maybe (c (s a))

gcast1 (x::c (t a)) :: Maybe (c (s a))
  = if typeOf1 (undefined::t a)
          == typeOf1 (undefined::s a)
    then Just (unsafeCoerce x)
         else Nothing
```

Conclusions

- Polymorphic type extension is (surprisingly) possible
- We want polymorphism at the kind level. Are there other applications for this? How much more complicated does the type system become?

Papers: http://research.microsoft.com/~simonpj