Generic Programming

Pouya Larjani

FP2006

Generic Programming

- Want to write a *universal* algorithm for certain task that works on any data structure.
- Useful for traversals, searches, serialization (`show`), de-serialization (`re`
- Straight-forward approach contains a l of "setup" code for traversal, and a sm amount of code for real algorithm.

Generics

- Generics are polytypic constructs whe value can have different types.
- Already seen basic examples in λ2:

```
id :: forall a. a → a
mypoly a b = (id a, id b)
mypoly 5 'a'
```

- Other languages:
  - Templates in `C++`
  - Generics in `Java 5.0` and `C# 2.0`

Example

Standard representation of a campus:

```
data Campus = C Name [Building]
data Building = B Name Address [Floor]
data Floor = F Name [Room]
data Room = R Int Capacity
data Capacity = Cap Int
```

Need to reduce the capacity of each roc by 25%:

```
decCap :: Int → Capacity → Capacity
decCap p (Cap c) = Cap $ 100 * c `div` (100 - p)
```

`decCap` is the only "interesting" part of th algorithm, the rest of the code is trave routine:

```
decC :: Int → Campus → Campus
decC p (C n bs) = C n $ map (decB p) bs
decB :: Int → Building → Building
decB p (B n a fs) = B n a $ map (decF p) fs
decF :: Int → Floor → Floor
decF p (F n rs) = F n $ map (decR p) rs
decR :: Int → Room → Room
decR p (R n c) = R n $ decCap p c
```

Using Generics

- Have a polymorphic traversal function that goes through every node in the da structure and applies a certain functio
- Another polymorphic function that performs the algorithm we want on the certain type of nodes, and leaves everything else untouched.

Problems

- Most of the code was for traversal, no real work we wanted to do.
- A different data structure is going to n another bunch of traversal code with exact same idea.
- A different task on the same data structure (e.g. total capacity of all roor will again need a bunch of traversal co

Approaches in Haskell

- Can't do this in (vanilla) Haskell98. Ne to use some extensions.
- "Generic H∀SKELL" (Andres Löh, Johan Jeuring, e
  - Uses *Type-Indexed Values* to "index" over differe type constructors. Can behave differently over different types to achieve polymorphism.
  - Generic applications: Applies a generic definition certain type.
  - Separate gh compiler – doesn't work on GHC 6.4 of the box.

Approaches in Haskell

- "Scrap your boilerplate" (Lämmel & Peyton-Jo
  - *Type Extensions*: Lift a function to apply it be abl apply it to any type.
  - *Generic Traversal Combinators*: Follows a certai method to traverse through a data structure and a generic function.
  - Only needs GHC 6.4 extensions.

Revisiting the example

```
decCap :: Int → Capacity → Capacity
decCap p (Cap c) = Cap $ 100 * c `div` (100 - p)
```

We want to only use this in a generic program:

```
everywhere (gDecCap 25) mcMaster
```

The `gDecCap` function is the generic versi of `decCap` which applies to any node in data structure, and behaves like ident on nodes that are not of type `Capacity`.

Required Extensions

- Reflection: Need to be able to determi the type of a polymorphic argument, a compare types.
- Second-rank polymorphism: We want be able to have a polymorphic type or left hand side of a function.
  `e.g. (forall a. a → a) → (forall b. b → b)`
- Unsafe coercion: Identity function that changes the type of its argument.

Type Extension

- We are going to lift a function to accep any type as its argument, and has the same behavior as before on its origina domain.
- To do this, we need to be able to determine and compare the type of variables (i.e. reflection) and change t behavior of the function accordingly.

Type Casting

- Assume we have a class `Typeable` tha allows us to get a type representation any instance of it, and that all the type we have are instances of `Typeable`.
- GHC 6.4 allows us to automatically de a data type from `Typeable` without worrying about manually instantiating

```
data Campus = C Name [Building]
              deriving (Typeable)
…
```

Type Extension

- To make a lift, we need a transformati of a certain type (`decCap p` in our exam and extend it to any type.
- If casting the function to our target typ returns a `Nothing` (i.e. types didn't mat then we do normal identity map, other we apply the function to the argument which we *know* is of correct type.

Type Casting

- `Typeable` class provides a `typeOf` met that we can use for getting a type representation of its argument. The argument of `typeOf` is *never* evaluate

```
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
```

Make Transformation

- To make a transformation in SYB, use `mkT`, defined as:

```
mkT :: (Typeable a, Typeable b) => (b→b) → a→
mkT f = case cast f of
    Just g → g
    Nothing → id
```

- For our example, we can make gDecC easily from `mkT` and `decCap`:

```
gDecCap p = mkT $ decCap p
```

Generic Traversal

- To traverse through the sub-items in a data structure, we need a generic transformation that traverses the valu of that type.
- Defined through class `Data`:

```
class Typeable a => Data a where
  gmapT :: (forall b. Data b => b→b) → a→a
```

- This is a rank-2 type, which is a Haske extension supported by GHC 6.4

Generic Traversal

- GHC allows us to automatically derive from `Data` without having to write all th code:

```
data Campus = C Name [Building]
              deriving (Typeable, Data)
…
```

- Now the 'everywhere' traversal combinator is easy to make:

```
everywhere ::   (forall b. Data b => b→b) →
                (forall a. Data a => a→a)
everywhere f = f . gmapT (everywhere f)
```

Single-Layer Traversals

- We need an instance of `Data` for ever type now.
- Primitive types:

```
instance Data Int where gmapT f x = x
…
```

- Everything is already defined.
- Our sample types:

```
instance Data Room where
    gmapT f (R n c) = R (f n) (f c)
```

Other Generic Traversals

- Now we can apply a certain transformation everywhere in data structure. What if we wanted to apply operator to everything in the structure get a summary?
- Adding all the nodes of a type togethe e.g. find the total capacity of all the ro in university.
- Search for something specific (use laz evaluation)

Generic Queries

- Make a query:

```
mkQ :: (Typeable a, Typeable b) => r → (b→r) → a→r
```

- Generic query map in `Data` class:

```
gmapQ :: (forall b. Data b => b→r) → a→[r]
```

- Generic Traversal:

```
everything ::   (r → r → r) →
                (forall b. Data b => b→r) →
                (forall a. Data a => a→r)
everything k f x =
    foldl k (f x) (gmapQ (everything k f) x)
```

References

- *Exploring Generic Haskell*
  Andres Löh, 2004
- *The Generic Haskell user's guide*
  Andres Löh and Johan Jeuring (editors), et al
- *Scrap Your Boilerplate: A practical Design Pattern f Generic Programming*
  Ralf Lämmel and Simon Peyton-Jones, 2003
- *Scrap More Boilerplate: Reflection, Zips, and Generalized Casts*
  Ralf Lämmel and Simon Peyton-Jones, 2004

Examples

- Find the total capacity of all rooms:

```
getCap :: Capacity -> Int
getCap (Cap c) = c
everything (+) (mkQ 0 getCap) mcMaster
```

- Find a building in university:

```
findB :: String -> Building -> Maybe Building
findB n b@(B n' _ _)     | n == n'     = Just b
                         | otherwise   = Nothing
orElse :: Maybe a -> Maybe a -> Maybe a
x `orElse` y = case x of
                  Just _ -> x
                  Nothing -> y
everything orElse (mkQ Nothing $ findB "ITB") mcMast
```