Homework 4: Polymorphism

Jack Shi - A92122910
Wyatt Guidry - A12994977
Austin Coleman - A12888539

Feb 23, 2019

1 [16pts] Type Polymorphism

1.1 Parametric Polymorphism

1. **Polymorphic abstract data type for stacks**

   (a) Complete implementation

```
data Stack a = Stack [a]
push :: Stack [a] -> a -> Stack [a]
push (Stack xs) x = Stack (x:xs)

pop :: Stack [a] -> Stack [a]
pop (Stack []) = Stack []
pop (Stack xs) = Stack (drop 1 xs)
```

   (b) Explain type functions **Answer:**
   `push` has type `Stack [a] -> a -> Stack [a]`. This eans that the `push` function takes a stack that is constructed from a list of type `a`, and a new element of type `a`, and return a new stack that is constructed from a list of type `a`. The return has the same type as the first argument since adding an element to the stack does not change it's type.
   `pop` has type `pop::Stack [a] -> Stack [a]`. It is a function that takes a `Stack` that is constructed from a list of type `a` and the return is the same type because removing an element from a `Stack` does not change the type of the `Stack`.

1.2 Ad-hoc Polymorphism

1. **Collection interface via type classes**

   (a) Complete implementation

```
class Collection c where
  push :: c -> int -> c
  pop :: c -> c

data Stack = Stack [int] deriving Show

-- | Make Stack an instance of the Collection class
instance Collection (Stack [c]) where
  push (Stack xs) x = Stack (x:xs)
  pop (Stack s) = if (lenght s) == 0
                  then Stack s
                  else Stack (drop 1 s)
```

```
data Queue = Queue [Int] deriving Show

-- | Make Queue an instance of the Collection class

instance Collection (Queue [c]) where
  push (Queue xs) x = Queue (xs ++ [x])
  pop (Queue s) = if (lenght q) == 0
                  then Queue q
                  else Queue (drop 1 q)
```

2 [18pts] Implementing Haskell Typeclasses

1. **Answer:**
   `MyEqD` represent a dictionary that contains the implementation for the operation `===`. The type generated is an operator that takes and dictionary and two arguments then return a Boolean. The generated function provide a template for pattern match for this specific dictionary. When the operator is invoked, the pattern matching will automatically type match to the correct dictionary therefore `===` would be able to find the correct implementation for that specific type.

2. Complete implementation

```
data Tree a = Leaf a | Node a (Tree a) (Tree a)
              deriving Show
instance MyEq (Tree a) where
  (===) (Leaf a) (Leaf b) = abs a == abs b
  (===) (Node a1 (Tree b1) (Tree c1)) (Node a2 (Tree b2) (Tree c2)) =
    abs a1 == abs a2 && b1 (===) b2 && c1 (===) c2
  (===) _ _                 = False
```

3. Complete implementation

```
dMyEqTree :: MyEqD a -> Tree a -> Tree a -> Bool
dMyEqTree elDict = MkMyEqD myEqTree
  where myEqTree (Leaf a) (Leaf b) = (===) elDict a  b
        myEqTree (Node a1 (Tree b1) (Tree c1)) (Node a2 (Tree b2) (Tree c2)) =
          (===) elDict a1 a2 && b1 === b2 && c1 === c2
        myEqTree _ _ = False
```

4. **Answer:**
   The compiler will change the type of `cmp` to `cmp::MyEqD a -> a -> a -> String`. This function will now be passed in a dictionary and the dictionary will be passed into the `===` operator for implementation lookup when invoked in cmp.

5. Complete implementation:

```
cmp :: MyEq a -> a -> a -> String
cmp dMyEqInt a b = if (===) dMyEqInt a b
                   then ``Equal''
                   else ``Not Equal''
result = cmp (dMyEqTree dMyEqInt) tree1 tree2
```