Homework 4: Polymorphism

CSE 130: Programming Languages

Early deadline: Feb 20 23:59, Hard deadline: Feb 23 23:59

Names & IDs:

1 [16pts] Type Polymorphism

In this problem we are going to explore parametric and ad-hoc polymorphism. Recall that ad-hoc polymorphism is implemented via type-classes in Haskell.

1.1 Parametric Polymorphism

Haskell and C++ both have mechanisms for creating a generic stack implementation that can be used to represent stacks with any kind of underlying elements. In C++, we could write a template for stack objects of the following form

```
template <typename A>
class node {
public:
   node(A v, node<A>* n) : val (v), next(n) { }
   A val;
   node<A>* next;
};

template <typename A>
class stack {
    node<A>* first;
public:
    stack() : first(nullptr) { }
    void push(A x) {
      node<A>* n = new node<A>(x, first);
      first = n;
    }
    void pop() {
      node<A>* n = first;
      first = first->next;
      delete n;
    }
    A top() {
      return first->val;
    }
};
```

In Haskell, polymorphic stacks are simpler to implement, partly due to Haskell's type inference. In this problem, you will write the implementation of a Haskell generic stack. Below is a skeleton of the implementation that you need to complete. Keep in mind that, in Haskell, your API functions (e.g., `push`) do not modify the stack they are called on, rather they create a new immutable stack (which e.g., may contain additional elements).

1. **[8pts] Polymorphic abstract data type for stacks**

   (a) **[6pts]** Fill in the implementation for the `push` and `pop` stack functions below.

   ```
   data Stack a = Stack [__]

   push :: ________ -> ________ -> ________

   push (Stack xs) x = ___________________


   pop :: ________ -> ________

   pop (Stack []) = Stack []

   pop (Stack xs) = ___________________
   ```

   (b) **[2pts]** Explain what the types functions of these functions mean? You do not have the provide a formal argument; just explain why the average Haskell programmer would expect the code to have these types.

   Answer:

1.2 Ad-hoc Polymorphism

Haskell and C++ both have mechanisms for creating overloaded functions that behave differently depending on the types of the arguments. In C++, you simply write multiple functions that share the same name but have different argument types. For example, the `+` operator is overloaded to also accept string types in which case it performs string concatenation. Another example is the `push` and `pop` functions which are defined for both the stack and the queue standard data structures.

```
#include <stack>
#include <queue>

int main() {
  std::stack<int> stack;
  std::queue<int> queue;

  stack.push(1); stack.push(2); stack.push(3);
  queue.push(1); queue.push(2); queue.push(3);

  stack.pop();
  queue.pop();
}
```

After the above code executes, the stack will end up with [1,2] whereas the queue will contain [2,3].

In Haskell, defining a new function that has the same name as a previously defined function will produce a multiple-declarations error. The way Haskell supports ad-hoc polymorphism is by using *type classes*, as we saw in class. For this problem, you will implement the type class *Collection* and modify the stack implementation from above to be an instance of `Collection`. Additionally, you will implement a queue data structure which should also be an instance of `Collection` and define both `push` and `pop`.

1. **[8pts] Collection interface via type classes**

   (a) **[8pts]** Fill in the implementation for the `Collection` type class below. For simplicity we assume that both stacks and queues only handle `Int`s and not other, generic types.

```
class Collection c where

  push :: ____ -> Int -> ____

  pop  :: ____ -> ____


data Stack = Stack [Int] deriving Show

-- | Make Stack an instance of the Collection class

instance _______ _______ where

  push (Stack xs) x = __________________

  pop  (Stack s)    = __________________

                      __________________

                      __________________

data Queue = Queue [Int] deriving Show

-- | Make Queue an instance of the Collection class

instance _______ _______ where

  push (Queue xs) x  = __________________

  pop  (Queue q)     = __________________

                       __________________

                       __________________
```

2 [18pts] Implementing Haskell Typeclasses

Suppose we are interested in considering two Haskell `Int`s `i` and `j` equal if the absolute value of `i` is equal to the absolute value of `j`:

```
abs i == abs j
```

Suppose we are further interested in considering data structures containing `Int`s as equal if the corresponding `Int`s in those structures are equal up to absolute value. We can use Haskell's type class mechanism to define a new type class `MyEq` comprised of a single operator `===` that denotes this notion of equality:

```
class MyEq a where
  (===) :: a -> a -> Bool
```

Using an instance declaration, we can make `Int` an instance of this type class:

```
instance MyEq Int where
  i === j  = abs i == abs j
```

1. [4pts] When processing the type class declaration for `MyEq`, the Haskell compiler will generate the following internal type and function declarations:

   ```
   data MyEqD a = MkMyEqD (a -> a -> Bool)

   (===) :: MyEqD a -> a -> a -> Bool
   (===) (MkMyEqD eq) = eq
   ```

   Explain what the generated datatype `MyEqD` represents and what values of this type "store." Then, explain what the generated function `===` does. (All of this can be answered in a few sentences.)

   Answer:

2. [4pts] Suppose that we are using the following `Tree` datatype and want to compare such trees using the `===` operator.

   ```
   data Tree a = Leaf a | Node a (Tree a) (Tree a)
                deriving Show
   ```

   Fill in the following instance declaration to make `Tree`s an instance of the type class `MyEq`:

   ```
   instance _______________________ where

      (===) ________________ ________________ = ____________________________________

      (===) ________________ ________________ = ____________________________________

      (===) ________________ ________________ = ____________________________________
   ```

3. [3pts] From such an instance declaration, the compiler will generate code to construct `Tree` dictionaries for `MyEq`. Fill in the following dictionary construction code:

```
dMyEqTree :: _________________________

dMyEqTree elDict = MkMyEqD myEqTree
  where myEqTree ________________ ________________ = ____________________________

        myEqTree ________________ ________________ = ____________________________

                                                     ____________________________

                                                     ____________________________

        myEqTree ________________ ________________ = ____________________________
```

4. [3pts] The `cmp` function below compares two values from any type that belongs to the `MyEq` type class and returns a `String` indicating whether the values were equal according to `===`.

```
cmp :: (MyEq a) => a -> a -> String
cmp t1 t2 = if t1 === t2
              then "Equal" else "Not Equal"
```

The value `result = cmp test1 test2` uses the function `cmp` to compare two test trees where:

```
test1 :: Tree Int
test2 :: Tree Int
```

The compiler will rewrite the `cmp` function and its uses. Explain how it does so.

Answer:

5. [4pts] Assume that the compiler generated a dictionary named `dMyEqInt` for the `Int` instance of `MyEq`:

```
dMyEqInt :: MyEqD Int
```

Fill in the following rewritten versions of the `cmp` function and `result` definition:

```
cmp :: __________________ -> __________________ -> __________________ -> String

cmp ________________ ________________ ________________ = if ________________________
                                                           then "Equal"
                                                           else "Not Equal"

result = cmp ________________________ tree1 tree2
```

Here `cmp` should operate on both `Tree`s and `Int`s. `result`, on the other hand, is defined in terms trees; the value is “Equal” if the two trees are equal according to `===` and “Not Equal” otherwise.

Acknowledgements

Any acknowledgements, crediting external resources or people should be listed below.