Homework 3: Polymorphism

CSE 130: Programming Languages

Early deadline: July 18 23:59, Hard deadline: July 20 23:59

Names & IDs:

Solutions

1 [42pts] Type embodiment and parametricity

Parametricity is a property of parametrically polymorphic functions which roughly says that "the more generic a function type is, the more restricted we are in implementing the function." This is because parametric polymorphism *removes information* about the concrete values, leaving only structural properties, later allowing the type to be *instantiated* with different concrete types of values. Intuitively, if we consider the set of inhabitants of some given type, then the more parametric the type, the smaller will be the set of values with such a type. In practice, parametricity allows us to reason about our functions based on the types[1], and is the theory behind some fancy extensions to the typeclass derivation mechanism of Haskell.

In this problem, we will explore some consequences of this property and learn to use types as a reasoning tool, through deriving programs *by calculation*.

1.1 [$2 \times 6 = 12$ **pts**] The Algebra of Datatypes

Definition: an *inhabitant* of a type τ is a value v such that $v :: \tau$. $|\tau|$ denotes the size of the set of inhabitants of τ.

To get started, we'll look at inhabiting ADTs, and then functions. For this question, do not consider `undefined` (sometimes written $\perp$) as a valid answer.

```
data Animal = Cat | Dog | Mouse
```

1. How many inhabitants does `Animal` have? Give an example.

Answer:

3, e.g. `Cat`

[1] P. Wadler, Theorems for Free! (1989)

```
data AnimalPair = AnimalPair Animal Animal
```

2. How many inhabitants does `AnimalPair` have? Give an example.

   > **Answer:**
   > $3 \times 3 = 9$, e.g. `AnimalPair Cat Dog`

```
data Maybe a = Just a | Nothing
```

3. How many inhabitants does `Maybe Animal` have? Give an example.

   > **Answer:**
   > $3 + 1 = 4$, e.g. `Nothing`

4. How many inhabitants does `Maybe a` have? Give your answer in terms of `a`.

   > **Answer:**
   > $|a| + 1$

```
data Pair a b = Pair a b
```

5. How many inhabitants does `Pair a b` have? Give your answer in terms of `a` and `b`.

   > **Answer:**
   > $|a| \times |b|$

```
data Either a b = Left a | Right b
```

6. How many inhabitants does `Either (Maybe Animal) (Pair (Pair Animal Animal) Animal)` have?

> **Answer:**
>
> $(3 + 1) + ((3 \times 3) \times 3) = 31$

1.2 [$2 \times 6 = 12$ **pts**] Types and Lambda Calculus

Now we consider the effect polymorphism has on the inhabitation of function types. For each of the following types, (a) write down how many distinct functions there are for the given most general type and (b) give an example (if one exists). Write all your functions in λ form.

1. $a \to a$

> **Answer:**
>
> There is only one function satisfying the type:
>
> $$I :: a \to a$$
> $$I = \lambda x.\ x$$

2. $a \to b$

> **Answer:**
>
> There are **no** functions of this type! If you don't know what b is, how can you pull a b out of thin air?

3. $a \to b \to a$

> **Answer:**
>
> There is only one function satisfying the type:
>
> $$K :: a \to b \to a$$
> $$K = \lambda x.\ (\lambda y.\ x)$$

4. $a \to b \to b$

> **Answer:**
>
> There is only one function satisfying the type:
>
> $$K_* :: a \to b \to b$$
> $$K_* = \lambda x.\ (\lambda y.\ y)$$

5. $(a \to b \to c) \to (a \to b) \to a \to c$

> **Answer:**
>
> Only one:
>
> $$S :: (a \to b \to c) \to (a \to b) \to a \to c$$
> $$S = \lambda xyz.\ x\ z\ (y\ z)$$

6. $a \to \mathbb{Z}_6$. (Hint: $\mathbb{Z}_n$ is the set of integers modulo n.)

> **Answer:**
>
> There are 6 different functions with this type. For instance:
>
> $$\mathit{const}0 :: a \to \mathbb{Z}_6$$
> $$\mathit{const}0 = K\ 0$$

1.3 [6pts] Type Tetris [BONUS]

Now we shall put parametricity to work as a reasoning tool to write Haskell programs completely mechanically — by simply plugging together functions of the correct types. For each part, give a definition in Haskell using only the following functions:

$$
\begin{aligned}
(\$) &:: (a \to b) \to a \to b \\
(.) &:: (b \to c) \to (a \to b) \to (a \to c) \\
\mathit{flip} &:: (a \to b \to c) \to (b \to a \to c) \\
\mathit{map} &:: (a \to b) \to [a] \to [b] \\
\mathit{concat} &:: [[a]] \to [a]
\end{aligned}
$$

You may, however, compose the given functions by creating λ abstractions. These are all standard Haskell functions, so you can verify your solutions out in GHCi. (Enter `:t <function>` to print the type of the function). For example, consider the function *hog* with the type:

$$\mathit{hog} :: (c \to d) \to (a \to b \to c) \to a \to b \to d$$

Given the above type, we can derive the following function definition:

$$\mathit{hog} = (.).(.)$$

To solve these problems, you may find it useful to first write them using λ abstractions.

1. [6pts] Give a definition for $bog :: (a \to [b]) \to [a] \to [b]$

Answer:

$$\begin{aligned} bog &:: (a \to [b]) \to [a] \to [b] \\ bog &= \lambda f \to concat\ .\ map\ f \end{aligned}$$

2. [6pts] Give a definition for $zog :: a \to [a \to b] \to [b]$.

Answer:

$$\begin{aligned} zog &:: \ a \to [a \to b] \to [b] \\ zog &= \lambda c \to \ map\ (\$c) \\ &= map\ .\ flip\ (\$) \end{aligned}$$

3. [6pts] Write bog using hog.

Answer:

$$\begin{aligned} bog &:: (a \to [b]) \to [a] \to [b] \\ bog &= hog\ concat\ map \\ &= concat\ `hog`\ map \qquad \text{infix} \end{aligned}$$

2 [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.

2.1 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 [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 (let (_:xs') = xs in xs')
   -- OR: pop (Stack xs) = Stack (tail 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:**
   >
   > In contrast to the C++ code, the types of these functions tell us that the functions simply process stacks: they take one stack as an argument and return a new stack by processing the original one.

2.2 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.

```haskell
class Collection c where
  push :: c -> Int -> c
  pop  :: c -> c

data Stack = Stack [Int] deriving Show

instance Collection Stack where
  push (Stack xs) x = Stack (x:xs)
  pop  (Stack s)    = case s of
                        []     -> Stack []
                        (x:xs) -> Stack xs


data Queue = Queue [Int] deriving Show

instance Collection Queue where
  push (Queue xs) x = Queue (xs ++ [x])
  pop  (Queue [])   = Queue []
  pop  (Queue q)   = case q of
                        []     -> Queue []
                        (x:xs) -> Queue xs
{- OR:
  push (Queue xs) x = Queue (x:xs)
  pop  (Queue [])   = Queue []
  pop  (Queue q)   = case q of
                        [] -> Queue []
                        xs -> Queue (init xs)
-}
```

Acknowledgements

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