CSC324 Assignment 3: Language Features (due Dec. 2)

Learning Objectives

- Write a non-trivial program in Haskell.
- Understand an implementation of mutable data in a pure functional programming paradigm.
- Use types and functions to add new program structure to a language.
- Implement an instance of a Monad.
- The starter code contains three files, `AList.hs, Mutation.hs` and `MutationUser.hs`.

Part 0: Warm-up with association lists

One step above the primitive data types we've been studying in Haskell so far is the *association list*, a poor man's dictionary, storing key-value pairs:

```
type AList a b = [(a, b)]
```

In this section, you'll write some utility functions to practice programming in Haskell, and because you might find them handy later on. Note that Haskell provides much better ways of storing maps (e.g., via hashing), but the point of this exercise is to get you writing your own code!

1. Complete each of the following functions inside `AList.hs`.
   (a) `lookupA :: Eq a => AList a b -> a -> b`: returns the value in the association list corresponding to the given key. Assumes that the key is in the list.
   (b) `insertA :: Eq a => AList a b -> (a, b) -> AList a b`: returns a new association list which is the old one, except with the new key-value pair inserted. However, it returns the same list if the key already exists in the list.
   (c) `updateA :: Eq a => AList a b -> (a, b) -> AList a b`: returns a new association list which is the old one, except with the value corresponding to the given key changed to the given new value. However, it returns the same list if the key does not appear in the list.

Part 1: Starter code

We have given you starter code in `Mutation.hs` already. Read through this section to make sure you understand what the different types defined there represent.

Memory

Our first step is to represent mutable values in Haskell (remember: representing is not the same as actually being). To do this, we'll follow a common approach of using a dictionary-like data structure to map names to values, representing what "variables" are stored in "memory."

First, we create a data type `Value` to represent the different types of data we'll store and allow the user to change.

```
data Value = IntVal Integer |
             BoolVal Bool |
             deriving Show
```

Then, we'll define the `Memory` data type as simply an association list mapping numeric keys to `Value`s.

```
type Memory = AList Integer Value deriving Show
```

You might notice that it's quite cumbersome to explicitly have different constructors for each `Value` type, but this is how we must proceed using what we currently know about Haskell's type system, since lists must contain values of the same data type. For ways of achieving heterogeneous lists (lists containing values of different types) in Haskell, see `https://wiki.haskell.org/Heterogenous_collections`

Pointers

You might think that we're going to the helpers in Part 0 directly to access and modify this memory. Indeed, that's not a bad idea, except we want the extra guarantee that when we access a particular value in memory, we know what type it's supposed to be, so that we don't, for example, accidentally access a `Bool` and try to treat it like an `Integer`. There are many ways of getting around this problem. We're going to use a one that draws inspiration from C: typed pointers. Traditionally, a pointer is a memory address, specifying the location of a particular stored value; in our implementation, it will just be a unique integer.

```
data Pointer a = P Integer
```

This is a little funny - we've defined a pointer with a type parameter, but not actually used that parameter in the constructor of that type. We use this to enable the Haskell compiler to statically check the types of our pointers in our code.

Our final ingredient to enable static type checking is to make the built-in types `Integer` and `Bool` instances of a type class that can interact with our memory representation. We need to define a new type class, which supports three actions. Notice that the type signatures ensure that pointers of a given type will only ever be used to store values of that same type!

```
class Mutable a where
    get :: Memory -> Pointer a -> a
    set :: Memory -> Pointer a -> a -> Memory
    def :: Memory -> Integer -> a -> (Pointer a, Memory)
```

The functions are essentially wrappers around the association list functions you implemented earlier, except that they constrain the types of the values manipulated in the memory. Note that `def` is a little special, in that it both returns a new `Pointer`, as well as new Memory.

2. Inside `Mutation.hs`, make `Integer` and `Bool` instances of the `Mutable` typeclass by implementing the three functions for each type (you may need to look up the syntax for making a type an instance of a type class).

   You can use `case` expressions (`http://learnyouahaskell.com/syntax-in-functions#case-expressions`) here to pattern match on the `Values` stored in the memory. For all erroneous cases (`get` a key which doesn't exist, `def` on a key which already exists, etc.) your functions should use the `error` function to raise a runtime error. Note that your code for `Integer` and `Bool` will probably be quite similar, other than the particular `Value` constructor you pattern match against.

3. To see whether you are following along, implement the following function inside `MutationUser.hs`. You will probably want to use `let` here.

```
-- | Takes a number <n> and memory, and stores two new values in memory:
--   - the integer (n + 3) at location 100
--   - the boolean (n > 0) at location 500
--   Return the pointer to each stored value, and the new memory.
--   You may assume these locations are not already used by the memory.
pointerTest :: Integer -> Memory -> ((Pointer Integer, Pointer Bool), Memory)
```

```
> let ((p1, p2), mem) = pointerTest 5 []
> get mem p1
8
> get mem p2
True
```

Even though you haven't written much code yet, it's quite important that you understand what you're doing at this stage, as the rest of the assignment builds on this.

Part 2: Chaining

Next, we're going to modify our type class to allow us to chain together stateful operations. This strategy exactly follows what we develop in lecture for Stack, so it is recommended that you do not do this part until we have reached that point in lecture, or you have read the corresponding section in the course notes.

4. In `Mutation.hs`, introduce a new type. Notice that we're using not using a type synonym here, so it's a little different from `StackOp` in lecture.

   ```
   data StateOp a = StateOp (Memory -> (a, Memory))

   runOp :: StateOp a -> Memory -> (a, Memory)
   runOp (StateOp op) mem = op mem
   ```

   Change the definitions of `get, set`, and `def` so that their types are all written in terms of `StateOp`. (Remember currying, and reorder the arguments.)

   After making this change, your `pointerTest` function will no longer compile. Don't worry, you'll return to it soon.

   Then, define the chaining operations "then", `(>>>) :: StateOp a -> StateOp b -> StateOp b`, and "bind", `(>~>) :: StateOp a -> (a -> StateOp b) -> StateOp b`, so that you can chain together invocations of the three basic operations. Once you are done this, you should be able to write code like:

   ```
   f :: Integer -> StateOp Bool
   f x =
       def 1 4 >~> \p1 ->
       def 2 True >~> \p2 ->
       set p1 (x + 5) >>>
       get p1 >~> \y ->
       set p2 (y > 3) >>>
       get p2
   ```

5. In `Mutation.hs`, implement `returnVal :: a -> StateOp a`, which is a function that takes a value, then creates a new `StateOp` which doesn't interact with the memory at all, and instead just returns the value as the first element in the tuple. Example usage:

   ```
   g :: Integer -> StateOp Integer
   g x =
       def 1 (x + 4) >~> \p ->
       get p >~> \y ->
       returnVal (x * y)

   > runOp (g 10) []
   (140, [(1, IntVal 14)])
   ```

Then in `MutationUser.hs`, use your work from this section to modify the type annotation and implementation of `pointerTest` so that it, too, is now a `StateOp`. After this, you should be able to compile your program with this function.

Part 3: Calling with references

With our creation of pointers, we have achieved essentially the same semantics as C function calls. We still pass all arguments by value, but now we have a type of value which really represents a reference to another value. If we pass such a reference to a function, we can then change the value it points to!

6. To illustrate this idea, implement the following functions in `MutationUser.hs`:

   (a) `swap :: Mutable a => Pointer a -> Pointer a -> StateOp ()`, which takes two pointers and swaps the values they refer to. Reminder that this is *not* something we knew how to do otherwise in either Racket (without mutation) or Haskell.

   (b) `swapCycle :: Mutable a => [Pointer a] -> StateOp ()`, which takes a list of pointers `p1, ..., pn`, with corresponding values `v1, ..., vn`, and sets `p1`'s value to `v2`, `p2`'s value to `v3`., etc., and `pn`'s value to `v1`. This function should not change anything if its argument has length less than 2.

Part 4: Safety Improvements

The implementation of memory and pointers that you have completed in the first parts of this assignment has a number of safety and memory concerns. Let's fix two of them.

Allocation

Our current `def` function specifies allows the user to specify an integer "address" for the value that will be stored in memory. This is pretty ridiculous: not only does it allow the user to accidentally try to claim some memory which is already allocated, the actual memory-access functions get and set operate on pointer values, not the raw integers specified by the user.

7. In Mutation.hs, write a function `alloc :: Mutable a => a -> StateOp (Pointer a)`, which is similar to `def`, except that the function automatically generates a *fresh* (i.e., unused) number to bind in the value in memory, rather than accepting a number as a parameter.

Deallocation

A part of what we mean when we say that memory lacks scope is that names never go *out* of scope - once we use `def` (or `alloc`) to create a new storage location in memory, this is a global name. This is particularly egregious when we realize that the pointers themselves are simply Haskell values, and so are local to whatever funtion they're defined in.

8. Define a function `free :: Mutable a => Pointer a -> StateOp ()` which takes a pointer, and removes the corresponding name-value binding from the memory. You should add to `AList.hs` to do this.

Part 5: Compound mutable values

Now that we are able to mutate booleans and integers, our final task will be mutating a simple compound data type (parallel to a struct in C). First, copy-and-paste your code from `Mutation.hs` into a new file, `CompoundMutation.hs`. You will do all your work for this part in this file, to allow us to mark your work from the previous parts separately. Copy-and-paste the following type into `CompoundMutation.hs`.

```
-- A type representing a person with two attributes:
-- age and whether they are a student or not.
data Person = Person Integer Bool deriving Show
```

Your task is to make `Person` an instance of the `Mutable` type class, **WITHOUT** adding an extra constructor to `Value`, but instead adding another constructor to `Pointer`. Think about this as the amount of data stored at each "address" in memory being small, and so we can't store a `Person` at one location; instead, a `Pointer Person` should contain one pointer to each attribute. This has the nice property that we should be able to access pointers to the attributes of a `Person` separately. We will leave the implementation open-ended, other than the requirement that you do not change `Value`. However, we constrain the interface you *must* provide: once you are finished, the following function must compile with your code.

```
personTest :: Person -> Integer -> StateOp (Integer, Bool, Person)
personTest person x =
    -- not using alloc, but we could
    def 1 person >~> \personPointer ->
    get (personPointer @@ age) >~> \oldAge ->
    set (personPointer @@ age) x >>>
    get (personPointer @@ isStudent) >~> \stu ->
    get (personPointer @@ age) >~> \newAge ->
    set personPointer (Person (2 * newAge) (not stu)) >>>
    get personPointer >~> \newPerson ->
    get (personPointer @@ isStudent) >~> \newStu ->
    returnVal (oldAge, newStu, newPerson)

> fst (runOp (personTest (Person 2 True) 10) [])
(2,False,Person 20 False)
```

Hint: `(@@)`, `age`, and `isStudent` should be implemented as functions.