Written Assignment 3

CS 538, Spring 2020

February 28, 2020

1 Practicing `do`-notation (15)

These exercises give you practice with working with Haskell's do-notation. We will work with several monads we have seen in class: `Maybe`, `OrErr`, and `WithLog`. Take a look at the lecture slides for the definitions of these types and their monad instances.

1.1 Using >>=

Translate the following examples using `>>=`, eliminating the do-notation. Simplify as far as you can.

(a)
```
do ns <- Right 17
   ls <- return [ns, 2]
   Right (7:ls)
```

(b)
```
do s1 <- MkWithLog (10 - 1, "Subtract 1")
   tn <- MkWithLog (s1 * 10, "Times n")
   return tn
```

(c)
```
do rc <- if 3 /= 0 then Just (1 / 3) else Nothing
   Just (4 * rc)
```

1.2 Using >>

Translate the following examples using `>>`, eliminating the do-notation. Simplify as far as you can.

(a)
```
do return "hello"
   Nothing
   Just "world"
```

(b)
```
do MkWithLog (n, "First")
   MkWithLog (37, "Second")
   return 12
```

1.3 Nested `do`-blocks

Translate the following examples using `>>=`, eliminating the do-notation. Simplify as far as you can.

(a)
```
do c <- Just 3
   let f n = do m <- Just (n - 7)
                Just (m * 3)
   in (f c)
```

(b)
```
do nnn <- Right [0, 3, 2, 1]
   case nnn of
     []    -> Left "empty list"
     (n:_) -> if n /= 0
              then (do r <- Right (1/n)
                       return r)
              else Left "division by zero"
```

1.4 More simplifying

Simplify the following expressions. Begin by replacing the `do`-notation by `>>=` and `>>`, then unfold definitions and simplify. Show a few intermediate steps and the final expression you reach.

(a)
```
do if True
   then MkWithLog (5, "First")
   else MkWithLog (37, "Second")
   return 12
```

(b)
```
do MkWithLog (5, "badger")
   return 4
   return 3
```

(c)
```
do k <- Right [4, 7, 6]
   v <- Left "fail 1"
   w <- Left "fail 2"
   Right $ \x -> 9:k
```

2 Lazy datatypes (15)

In class so far, we have seen how to model *eager* datatypes. Roughly speaking, eliminating these types (via projection or pattern matching) requires evaluating *all* of their components to values. As we have mentioned, Haskell's datatypes are *lazy*, not eager. In this exercise, we will see how to model lazy datatypes. Since we are focusing on the operational behavior (eager versus lazy), we will not be working with typing rules.

To start, we will work with the lambda calculus with booleans and recursion. The syntax of values and expressions are as follows:

$$v ::= x \mid \mathit{true} \mid \mathit{false} \mid \lambda x.\ e$$
$$e ::= x \mid \mathit{true} \mid \mathit{false} \mid \lambda x.\ e \mid e_1\ e_2 \mid \mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3 \mid \mathit{fix}\ x.\ e$$

Throughout this section, we use specific letters to represent different kinds of programs:

- The letter e stands for any *expression* (any program)
- The letter x stands for any *variable*
- The letter v stands for any *value*

We will work with a small-step semantics. Function application will be lazy (call-by-name):

$$\frac{e_1 \to e_1'}{e_1\ e_2 \to e_1'\ e_2} \qquad \frac{}{(\lambda x.\ e_1)\ e_2 \to e_1[x \mapsto e_2]}$$

In particular, note that the argument of a function is *not* evaluated to a value before it is plugged into the function. As we have seen in class, the fixed-point construct steps by unfolding:

$$\frac{}{\mathit{fix}\ x.\ e \to e[x \mapsto (\mathit{fix}\ x.\ e)]}$$

As always, $e[x \mapsto e']$ stands for the expression e where all free occurrences of variable x have been replaced by e'.

2.1 Lazy products

Let's start with lazy products, also known as tuples. We start by extending the expression grammar:

$$e ::= \cdots \mid (e_1, e_2) \mid \mathit{fst}(e) \mid \mathit{snd}(e)$$

For the *eager* tuples we saw in class, we added a new value of the form (v_1, v_2). For lazy tuples, we will add something slightly different:

$$v ::= \cdots \mid (e_1, e_2)$$

In particular, note that a tuple of two expressions (e_1, e_2) is a *value*, even if e_1 and e_2 are not values.

(a) Write out the small-step rules for the three new constructs. Hint: you will need four rules, two for $\mathit{fst}(e)$ and two for $\mathit{snd}(e)$—these should be almost exactly the same as the rules we saw in class for eager tuples, only now we have defined values differently. You do not need a rule to step tuples, since tuples are values now.

(b) Consider the following bit of Haskell code using lazy tuples:

```
loopForever :: Bool
loopForever = loopForever

getFst :: Bool
getFst = fst (True, loopForever)

getSnd :: Bool
getSnd = snd (True, loopForever)
```

Translate each definition into the lambda calculus, and show how the three expressions step. You should confirm that `getFst` terminates, while `getSnd` does not. Hint: a quick way to write a non-terminating expression is $\mathit{fix}\ x.\ x$.

2.2 Lazy lists

Now, let's add lazy lists. We start by extending the expression grammar:

$$e ::= \cdots \mid \mathit{nil} \mid \mathit{cons}(e_1, e_2) \mid \mathit{case}\ e\ \mathit{of}\ \{\mathit{nil} \to e_1; \mathit{cons}(x_1, x_2) \to e_2\}$$

There are two constructors for lists: nil is empty list, and $\mathit{cons}(e_1, e_2)$ is a non-empty list with head e_1 and tail e_2. Lists are eliminated/destructed by pattern matching—the case expression has two cases, one for empty list and one for non-empty lists. Note that in the second case, the body e_2 can use the newly bound variables x_1 and x_2, which name the first and tail elements of e.

We'll extend the values of our language as follows:

$$v ::= \cdots \mid \mathit{nil} \mid \mathit{cons}(e_1, e_2)$$

In particular, note that the cons of two expressions $\mathit{cons}(e_1, e_2)$ is a *value*, even if e_1 and e_2 are not values.

(a) Write out the small-step rules for the three new constructs. Hint: you will need three rules, all for the pattern match. You do not need a rule to step *nil* and $cons(e_1, e_2)$, since these are values now.

(b) Consider the following bit of Haskell code using lazy lists:

```
allTrue :: [Bool]
allTrue = True : allTrue

negate :: Bool -> Bool
negate b = if b then False else True

myMap :: (Bool -> Bool) -> [Bool] -> [Bool]
myMap f ls = case ls of
                [] -> []
                (b:bs) -> (f b) : (myMap f bs)

isEmpty :: Bool
isEmpty = case (myMap negate allTrue) of
             [] -> True
             (b:bs) -> False
```

Translate each definition into the lambda calculus, and show how `allTrue` steps. Then, show how `isEmpty` steps. You should confirm that these two expressions both terminate.

As we saw in class, `myMap` can also be defined using "accumulating" recursion; this leads to a tail-recursive function and is often more efficient. What would happen if we defined `myMap` this way?

```
myAccMap :: (Bool -> Bool) -> [Bool] -> [Bool]
myAccMap f ls = go f ls []
    where go f' ls' acc = case ls' of
                             [] -> acc
                             (b:bs) -> go f' bs (acc ++ [f' b])

isAccEmpty :: Bool
isAccEmpty = case (myAccMap negate allTrue) of
                [] -> True
                (b:bs) -> False
```

Would `isAccEmpty` still terminate? You don't need to translate these definitions to the lambda calculus (and it's not important how `++` is defined), but you can check your predictions with GHCi.