

Written Assignment 3

CS 538, Spring 2020

Vinay Patil

1 Practicing do-notation (15)

1.1 Using >>=

- (a) `func = (Right 17) >>= (\first ->
 return [first, 2] >>= (\second ->
 return (7:second)))`
- `=> Right [7,17,2]`
- (b) `func = MkWithLog (10 - 1, "Subtract 1") >>= (\first ->
 MkWithLog (first*10, "Times n"))`
- `=> MkWithLog (90,"Subtract 1Times n")`
- (c) `func = (if 3 /= 0 then Just (1 / 3) else Nothing) >>= (\first ->
 return (4 * first))`
- `=> Just 1.3333333333333333`

1.2 Using >>

- (a) `func = return "hello" >> Nothing >> Just "world"`
- `=> Nothing`
- (b) `func n = MkWithLog (n, "First") >> MkWithLog (37, "Second") >> return 12`
- `=> MkWithLog (12,"FirstSecond")`

1.3 Nested do-blocks

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

- (a) `func = Just 3 >>= (\first ->
 Just (first - 7) >>= (\second ->
 Just (second * 3)))`
- `=> Just (-12)`

```
(b) func = Right [0,3,2,1] >= (\first ->
    case first of
    [] -> Left "empty list"
    (n:_) -> if n/=0
        then Right (1/n)
        else Left "division by zero")

=> Left "division by zero"
```

1.4 More simplifying

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

-- Since boolean condition is always True, the express in then is selected for evaluation
= (MkWithLog (5, "First")) >> (return 12)

-- This will be basically translated to (MkWithLog (5, "First")) >= (\_ -> return 12)
-- Also (return 12) lazily evaluates to MkWithLog (12,"")
= MkWithLog (12, "First" ++ "") -- 5 is ignored by the definition of >>

-- Final Result will be
= MkWithLog (12, "First")

(b) func = MkWithLog (5, "badger") >> return 4 >> return 3

-- Evaluation is from left to right
-- After lazily evaluating (return 4) we get MkWithLog (4, "")
= MkWithLog (4, "badger" + "") >> return 3
= MkWithLog (4, "badger") >> return 3

-- After lazily evaluating (return 3) we get MkWithLog (3, "")
= MkWithLog (3, "badger" + "")

-- Final Result will be
= MkWithLog (3, "badger")

(c) func = (Right [4, 7, 6]) >= (\first ->
    Left "fail 1" >> Left "fail 2" >> (Right $ (\x -> 9:first)))

-- Evaluation is from left to right
-- first takes the value of [4, 7, 6]
```

```
-- But when Haskell see an error case which is Left "Fail 1", it stops executing further

-- Will halt execution and the result value will be

= Left "fail1"
```

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:

$$\begin{aligned} v &::= x \mid \text{true} \mid \text{false} \mid \lambda x. e \\ e &::= x \mid \text{true} \mid \text{false} \mid \lambda x. e \mid e_1 e_2 \mid \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \mid \text{fix } x. e \end{aligned}$$

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):

$$\frac{e_1 \rightarrow e'_1}{e_1 e_2 \rightarrow e'_1 e_2} \qquad \frac{}{(\lambda x. e_1) e_2 \rightarrow e_1[x \mapsto e_2]}$$

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:

$$\frac{}{\text{fix } x. e \rightarrow e[x \mapsto (\text{fix } x. e)]}$$

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:

$$e ::= \dots \mid (e_1, e_2) \mid \text{fst}(e) \mid \text{snd}(e)$$

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:

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

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.

- Write out the small-step rules for the three new constructs. Hint: you will need four rules, two for $\text{fst}(e)$ and two for $\text{snd}(e)$ —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 *fix* *x*. *x*.

2.2 Lazy lists

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

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

There are two constructors for lists: *nil* is empty list, and *cons*(*e*₁, *e*₂) is a non-empty list with head *e*₁ and tail *e*₂. Lists are eliminated/destroyed 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:

$$v ::= \dots \mid nil \mid cons(e_1, e_2)$$

In particular, note that the cons of two expressions *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`.