CSE 130 Midterm, Spring 20

Nadia Polikarpova

May 4, 2020

- Exam released: Monday, May 4, 9am PDT
- Exam due: Tuesday, May 5, 9am PDT
- You must submit your answers via an online quiz on Gradescope
- You **may** consult any course material (lecture notes, assignments, past exams, etc)
- You **may** test your code in Elsa/ghci if you'd like
- You **may** ask for clarifications on Piazza via a private message to instructors
- You **may not** communicate with other students or ask for anyone's help
- You **may not** search help forums (StackOverflow) and the Internet for a solution
- **Good luck!**

Q1: Lambda Calculus: Reductions [5 pts]

Check the box next to *each* term that is *in normal form.* You get 1 pt for each box you correctly marked or left unmarked.

(A) `\y -> y (\y -> y)` [ ]

(B) `(\y -> y) (\y -> y)` [ ]

(C) `g (\y -> y) (\y -> y)` [ ]

(D) `(\y -> y) g (\y -> y)` [ ]

(E) `(\y -> y y) (\y -> y y)` [ ]

Q2: Lambda Calculus: Functions [15 pts]

Below you will find *five* lambda terms, `F1` through `F5` (along with helper functions, `H3` through `H5`). Each one of them implements one of *eight* possible algorithms, `(A)` through `(H)`. Your task is to guess for each term, which algorithm it implements. You can use each algorithm *zero or more* times (i.e. some algorithms might remain unused and some might be used more than once):

```
(A) GCD
(B) natural number subtraction
(C) natural number division
(D) natural number modulo
(E) equality on booleans
(F) is natural number odd?
(G) fibonacci
(H) factorial
```

You get 3 points for each lambda term whose meaning you guessed correctly. You will find the definitions of all the functions used inside the terms in Appendix I at the end of the exam.

Lambda terms:

```
-- (1)
let F1     = \x y -> x y (NOT y)

-- (2)
let F2     = \n -> n NOT FALSE

-- (3)
let H3     = \rec n -> ITE (ISZ (DEC n))
                           n
                           (ADD (rec (DEC n)) (rec (DEC (DEC n))))
let F3    = FIX H3

-- (4)
let H4     = \n m i -> ITE (LEQ (MUL m (INC i)) n) (INC i) i
let F4     = \n m -> n (H4 n m) ZERO

-- (5)
let H5     = \m i j -> ITE (EQL (INC j) m)
                           (PAIR (INC i) ZERO)
                           (PAIR i (INC j))
let F5     = \n m -> FST (n (\p -> H5 m (FST p) (SND p))
                            (PAIR ZERO ZERO))
```

Q3: Binary Heaps [18 pts]

Meet Jean: she's been quarantined at home with her cat Josephine and her laptop, so she decided to learn some Haskell. As her next exercise, Jean wants to implement a binary heap data structure.

Figure 1: Jean, a Haskell and cat enthusiast

Recall that a **binary heap** is a binary tree with two additional properties:

- *Shape property:* all levels of the tree, except possibly the last one (deepest) are fully filled, and, if the last level of the tree is not complete, the nodes of that level are filled *from left to right*.
- *Value property:* the key stored in each node is `>=` the keys of its children.

Here are two examples of binary heaps:

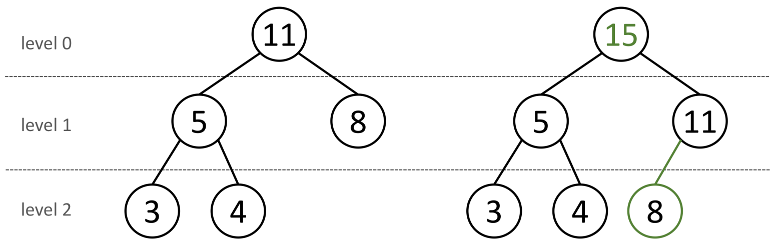

Figure 2: **(left)** A binary heap. **(right)** The same heap after inserting 15 into it.

Here are three examples of trees that are **not** binary heaps: in the first tree, level 1 is not fully filled; in the second tree, level 2 is not filled left-to-right

(right child of node 5 is missing); the last tree violates the value property, since $8 < 15$.

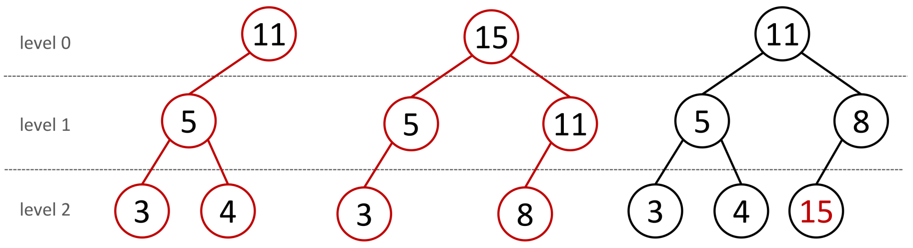

Figure 3: **Not** binary heaps: the first two violate the shape property, while the last one violates the value property.

Jean wants to implement **insertion** into a binary heap. Fig. 1 (right) illustrates insertion of the new key 15 into the binary heap on Fig. 1 (left):

- To maintain the *shape property*, a new node is inserted at the left-most free position in the last level (or, if the last level is full, a new level is started);
- To maintain the *value property*, the new key is placed at the appropriate level on the path from the root to the new node (here, 15 ends up at the root, while 11 and 8 are pushed down).

Last night Jean wrote a simple version of the binary heap insertion in Haskell, but when she woke up this morning, she realized that her cat was walking over her keyboard all night, so the lines in her code got all messed up! Help Jean figure out which lines of code `(A)-(W)` go into the holes `(1)-(9)` below. Each correctly filled hole is worth 2 points.

```haskell
-- | Binary heap datatype:
data BHeap = Leaf | Node Int BHeap BHeap

-- | Height of the heap
height :: BHeap -> Int
height Leaf         = 0
height (Node _ l r) = 1 + max (height l) (height r)

{- Task 3.1: Is full? -}

-- | Is this binary tree full?
-- | We say that a tree is full if has no partially filled levels.
-- | For example:
-- | isFull (Node 5 (Node 3 Leaf Leaf) (Node 4 Leaf Leaf)) ==> True
-- | isFull (Node 5 (Node 3 Leaf Leaf) Leaf)               ==> False
isFull :: BHeap -> Bool
isFull Leaf         = {- (1) -}
isFull (Node _ l r) = {- (2) -}

{- Task 3.2: Insert -}

-- | Insert a new key into the binary heap.
-- | For example:
-- | insert 5 (Node 11 (Node 3 Leaf Leaf) (Node 8 Leaf Leaf)) ==>
-- |     Node 11 (Node 5 (Node 3 Leaf Leaf) Leaf) (Node 4 Leaf Leaf)
-- | (see one more example in fig 1)
-- |
-- | Recall that the pattern h@(Node y l r) matches a value against Node,
-- | but also binds the whole value to h
insert :: Int -> BHeap -> BHeap
insert x Leaf          = {- (3) -}
insert x h@(Node y l r)
    | isFull h         = {- (4) -}
    | {- (5) -}        = {- (6) -}
    | otherwise        = {- (7) -}
  where
      lo = min x y
      hi = max x y
```

```
{- Task 3.3: Tail-recursive insert all -}

-- | Insert all values from a list into a binary heap.
-- | For example:
-- | insertAll [5,3,4] Leaf ==> Node 5 (Node 3 Leaf Leaf) (Node 4 Leaf Leaf)
-- |
-- | Your function must be *tail-recursive*!
insertAll :: [Int] -> BHeap -> BHeap
insertAll []     h = {- (8) -}
insertAll (x:xs) h = {- (9) -}
```

You can use each of the following lines of code **zero or more** times (i.e. some lines might remain unused and some might be used more than once):

```
{- A -} 0
{- B -} 1
{- C -} h
{- D -} True
{- E -} False
{- F -} isFull l
{- G -} isFull r
{- H -} Node x l r
{- I -} Node x Leaf Leaf
{- J -} isFull l || isFull r
{- K -} isFull l && isFull r
{- L -} Node y (insert x l) r
{- M -} Node x (insert y l) r
{- N -} Node y l (insert x r)
{- O -} Node x l (insert y r)
{- P -} height l == height r
{- Q -} height l >= height r
{- R -} insert x (insert xs h)
{- S -} Node hi (insert lo l) r
{- T -} Node hi l (insert lo r)
{- U -} insertAll xs (insert x h)
{- V -} insert x (insertAll xs h)
{- W -} isFull l && isFull r && (height l == height r)
```

Lambda Calculus Cheat Sheet

Here is a list of definitions you may find useful for Q2

```
-- Booleans ---------------------------------

let TRUE  = \x y -> x
let FALSE = \x y -> y
let ITE   = \b x y -> b x y
let NOT   = \b x y -> b y x
let AND   = \b1 b2 -> ITE b1 b2 FALSE

-- Pairs ------------------------------------

let PAIR  = \x y b -> b x y
let FST   = \p     -> p TRUE
let SND   = \p     -> p FALSE

-- Numbers ----------------------------------

let ZERO  = \f x -> x
let ONE   = \f x -> f x
let TWO   = \f x -> f (f x)
let INC   = \n f x -> f (n f x)
let ADD   = \n m -> n INC m
let ISZ   = \n -> n (\z -> FALSE) TRUE
let SKIP1 = \f p -> PAIR TRUE (ITE (FST p) (f (SND p)) (SND p))
let DEC   = \n   -> SND (n (SKIP1 INC) (PAIR FALSE ZERO))
let SUB   = \n m -> m DEC n
let LEQ   = \n m -> ISZ (SUB n m)
let EQL   = \n m -> AND (LEQ n m) (LEQ m n)

-- Fixpoint -------------------------------

let FIX   = \stp -> (\x -> stp (x x)) (\x -> stp (x x))
```