Homework 5 (Runtime Stack, Scoping, and Parameter Passing)

CS 381, Spring 2019

June 16, 2019

Prepared by

Adam Stewart *(stewaada)*
Anish Asrani *(asrania)*
Lucas Frey *(freyl)*
Mazen Alotaibi *(alotaima)*
Sergei Poliakov *(poliakos)*

Contents

1 Exercise 1. A Rank-Based Type Systems for the Stack Language 2

2 Exercise 2. Shape Language 3

3 Exercise 3. Parametric Polymorphism 4

1 Exercise 1. A Rank-Based Type Systems for the Stack Language

```
module Homework4E1 where

import Data.List

-- Exercise 1 A Rank-Based Type Systems for the Stack Language

data Cmd = LD Int
         | ADD
         | MULT
         | DUP
         | INC
         | SWAP
         | POP Int
         deriving (Eq, Show)

type Prog = [Cmd]

type Stack = [Int]

type D = Stack -> Stack

semCmdHelper:: Cmd -> D
semCmdHelper (ADD)  (x:y:xs) = ((x+y):xs)
semCmdHelper (MULT) (x:y:xs) = ((x*y):xs)
semCmdHelper (DUP)  (x:xs)   = ([x,x] ++ xs)
semCmdHelper (INC) (x:xs) = ((x+1):xs)
semCmdHelper (SWAP) (x:y:xs) = (y:x:xs)
semCmdHelper (POP x) xs = drop x (reverse xs)


semCmd :: Cmd -> D
semCmd (LD x) xs = xs ++ [x]
semCmd (ADD)  x | (length x) == 0 || (length x) == 1 = error ("CANNOT ADD")
                | otherwise = semCmdHelper (ADD) x
semCmd (MULT) x | (length x) == 0 || (length x) == 1 = error ("CANNOT MULT")
                | otherwise = semCmdHelper (MULT) x
semCmd (DUP)  x | (length x) == 0 = error ("NOTHING TO DUPLICATE")
                | otherwise = semCmdHelper (DUP) x
semCmd (INC)  x | (length x) == 0 = error ("NOTHING TO INCREMENT")
                | otherwise = semCmdHelper (INC) x
semCmd (SWAP) x | (length x) == 0 || (length x) == 1 = error ("CANNOT SWAP")
                | otherwise = semCmdHelper (SWAP) x
semCmd (POP x) xs | (length xs) < x = error ("CANNOT POP")
                  | otherwise = semCmdHelper (POP x) xs

sem :: Prog -> D
sem [] y = y
sem (x:xs) y = sem xs (semCmd x y)
```

```
-- compute:: Prog -> Stack
-- compute p = sem p ([])

-- a)
type Rank = Int
type CmdRank = (Int,Int)

rankC :: Cmd -> CmdRank
rankC (LD _)  = (0,1)
rankC (ADD)   = (2,1)
rankC (MULT)  = (2,1)
rankC (DUP)   = (1,2)
rankC (INC)   = (1,1)
rankC (SWAP)  = (2,2)
rankC (POP x) = (x,0)

rank :: Prog -> Rank -> Maybe Rank
rank [] x | x >= 0 = Just x
          | otherwise = Nothing
rank (x:xs) y = rank xs ((y-n)+m)
                where n = fst(rankC x)
                      m = snd(rankC x)

rankP :: Prog -> Maybe Rank
rankP p = rank p 0
```

```
-- b)

typeCorrect :: Prog -> Bool
typeCorrect e = rankP e /= Nothing

semStatTC :: Prog -> Maybe Stack
semStatTC e | typeCorrect e = Just (sem e ([]))
            | otherwise = Nothing
```

2 Exercise 2. Shape Language

```
--
-- Exercise 2
--
module HW4E2 where
import Data.Tuple
import Data.Maybe

data Shape = X
           | TD Shape Shape
           | LR Shape Shape
           deriving Show

type BBox = (Int, Int)
```

```
-- (a)

bbox :: Shape -> BBox

bbox X              = (1, 1)
bbox (LR s1 s2)     = (x1+x2, max y1 y2)
                        where (x1, y1) = bbox s1
                              (x2, y2) = bbox s2

bbox (TD s1 s2)     = (max x1 x2, y1+y2)
                        where (x1, y1) = bbox s1
                              (x2, y2) = bbox s2
```

```
rect :: Shape -> Maybe BBox

rect X                        = Just (1,1)
rect (LR s1 s2) | y1==y2      = Just (x1+x2, y2)
                | otherwise   = Nothing
                              where Just (x1, y1) = rect s1
                                    Just (x2, y2) = rect s2

rect (TD s1 s2) | x1==x2      = Just (x1, y1+y2)
                | otherwise   = Nothing
                              where Just (x1, y1) = rect s1
                                  Just (x2, y2) = rect s2
```

3 Exercise 3. Parametric Polymorphism

```
module Homework4E3 where

import Data.List

-- Exercise 3 Parametric Polymorphism

-- a)

f x y = if null x then [y] else x

g x y = if not (null x) then [] else [y]
g [] y = []
{-
(1) What are the types of f and g?
     f :: [a] -> a -> [a]
     g :: [a] -> b -> [b]
(2) Explain why the functions have these types.
     Both functions take in two parameters, the
     first being a list of ”a’s” and the second
     parameter being either a value ”a” or ”b”,
     depending on the function. Then both functions
     return a list of ”a’s”.
```

```
(3) Which type is more general?
     The type of the function g is more general,
     because the output of the function will
     always be a list, regardless whether the
     condition is met or not.

(4) Why do f and g have different types?
     Function g has a different type from f, because
     in g, x is never part of the output.
     -}
```

```

-- b)
h :: [b] -> [(a,b)] -> [b]
h x _ = x
```

```
-- c)

k :: (a -> b) -> ((a -> b) -> a) -> b
k x y = x (y x)
```

```
-- d)
{- The function of type a -> b is difficult to defince because
   not enough information is given about type b. Therefore
   we cannot produce an accurate function defenition, that will
   perform the proper type conversions. -}
```