

PLD Assignment 3

Matilde Broløs (jtw868)

March 25, 2020

A3.1

a)

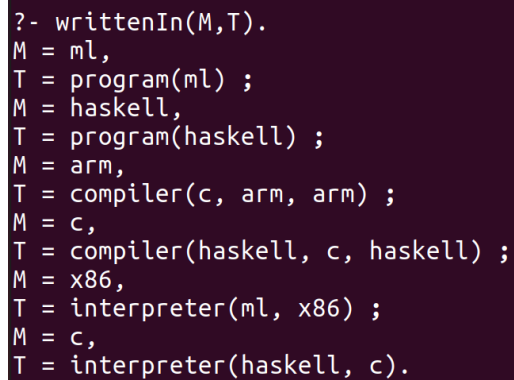
The predicate `writtenIn(M,T)` can be implemented as follows.

```
writtenIn(M,program(M)) :-  
    program(M).
```

```
writtenIn(M,compiler(X, M, Y)) :-  
    compiler(X, M, Y).
```

```
writtenIn(M,interpreter(X,M)) :-  
    interpreter(X,M).
```

The result of running `?-writtenIn(M,T)` is seen below.



```
?- writtenIn(M,T).  
M = ml,  
T = program(ml) ;  
M = haskell,  
T = program(haskell) ;  
M = arm,  
T = compiler(c, arm, arm) ;  
M = c,  
T = compiler(haskell, c, haskell) ;  
M = x86,  
T = interpreter(ml, x86) ;  
M = c,  
T = interpreter(haskell, c).
```

Figure 1: Result of running `writtenIn`.

b)

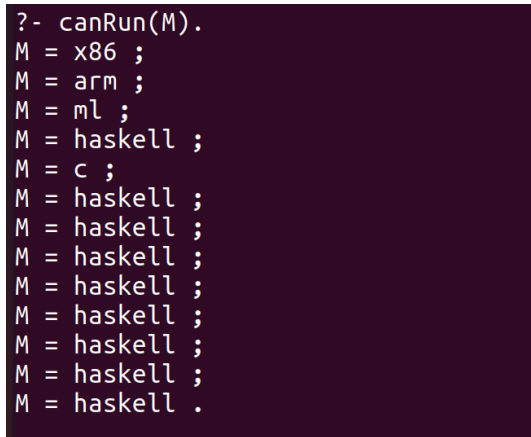
The predicate `canRun(L)` could be implemented as follows.

```
canRun(L) :-  
    machine(L).
```

```
canRun(L) :-  
    interpreter(L,X),  
    canRun(X).
```

```
canRun(L) :-  
    compiler(L,X,Y),  
    canRun(X),  
    canRun(Y).
```

The result of running `?-canRun(L)` is seen below.



```
?- canRun(M).  
M = x86 ;  
M = arm ;  
M = ml ;  
M = haskell ;  
M = c ;  
M = haskell ;  
M = haskell ;  
M = haskell ;  
M = haskell ;  
M = haskell ;  
M = haskell ;  
M = haskell ;  
M = haskell ;  
M = haskell .
```

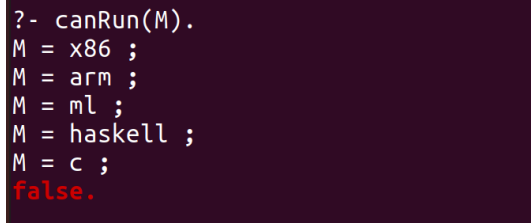
Figure 2: Result of running `canRun`.

c)

To avoid getting `L = haskell` an infinite number of times, we add the condition that the compiler cannot translate the program into a new program in the same language.

```
canRun(L) :-  
    compiler(L,X,Y),  
    L \= Y,  
    canRun(X),  
    canRun(Y).
```

Now when running the query we get the following result.



```
?- canRun(M).  
M = x86 ;  
M = arm ;  
M = ml ;  
M = haskell ;  
M = c ;  
false.
```

Figure 3: Result of running `canRun`.

A3.2

a)

First let's expand the type system with the rule for the **let**-expression. First we must check three things:

1. When evaluating e_1 with an environment τ_1 it evaluates to a type t_1
2. The pattern p matches a type t_1 and produces an environment τ_2
3. When evaluating e_2 with an environment $\tau_1 \circ \tau_2$ it evaluates to a type t_2

Now if these conditions are true, we can form the **let**-expression. Thus the final type rule is:

$$\frac{\tau_1 \vdash e_1 : t_1 \quad t_1 \vdash p \hookrightarrow \tau_2 \quad \tau_1 \circ \tau_2 \vdash e_2 : t_2}{\tau \vdash \mathbf{let} \ p : t_1 = e_1 : t_1 \ \mathbf{in} \ e_2 : t_2}$$

Now we define the type rules for patterns. If the pattern matches the type **int** then we produce an empty environment.

$$\overline{\mathbf{int} \vdash n \hookrightarrow []}$$

If the pattern matches a variable of type t then we map the variable to the type.

$$\overline{t \vdash x \hookrightarrow [x \mapsto t]}$$

For the last rule, if p_1 is compatible with t_1 , and p_2 is compatible with t_2 , then the resulting type environment is the two environments combined.

$$\frac{t_1 \vdash p_1 \hookrightarrow \tau_1 \quad t_2 \vdash p_2 \hookrightarrow \tau_2}{(t_1 * t_2) \vdash (p_1, p_2) \hookrightarrow \tau_1 \circ \tau_2}$$

b)

Again we start by writing the semantic rule for the **let**-expression. The structure follows the same logic as in the previous task.

$$\frac{\rho_1 \vdash e_1 \rightsquigarrow v_1 \quad v_1 \vdash p \rightarrow \rho_2 \quad \rho_1 \circ \rho_2 \vdash e_2 \rightsquigarrow v_2}{\rho \vdash \mathbf{let} \ p \rightsquigarrow v_1 = e_1 \rightsquigarrow v_1 \ \mathbf{in} \ e_2 \rightsquigarrow v_2}$$

Now we define the semantic rules for patterns. If $v = n$, where n is a number, then when v matches n we produce the empty value environment.

$$\frac{v = n}{v \vdash n \rightarrow []}$$

When the pattern is a variable x matching a value v , the produced environment maps the variable to the value v .

$$\overline{v \vdash x \rightarrow [x \mapsto v]}$$

Last last rule follows the logic of the type system rule very closely. If the pattern p_1 matches a value v_1 and produces an environment ρ_1 , and the pattern p_2 matches a value v_2 and produces an environment ρ_2 , then the environment of the pair of the two patterns will be the two environments combined.

$$\frac{v_1 \vdash p_1 \rightarrow \rho_1 \quad v_2 \vdash p_2 \rightarrow \rho_2}{(v_1 * v_2) \vdash (p_1, p_2) \rightarrow \rho_1 \circ \rho_2}$$

A3.3

a)

The probability that the third-largest of 39 nine-sided dice will be 8 can be calculated with the following command in Troll.

```
least 1 largest 3 39d9
```

The result can be seen in figure 4.

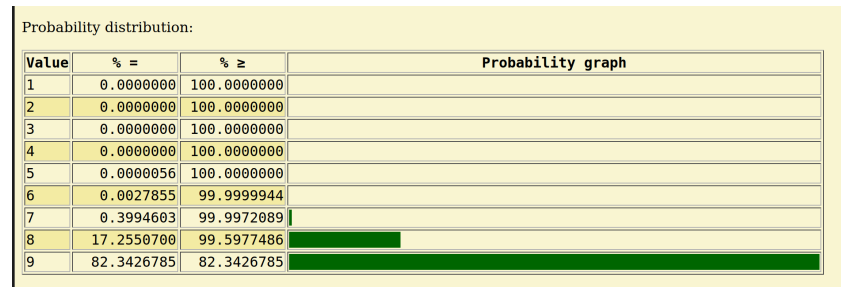


Figure 4: Result of running the above code.

Thus, the probability is 0.172550700.

b)

To compute the probability of 8 being the third highest roll when rolling 39 9-sided dice, let's first look at the first three rolls. There are a number of different combinations of numbers in the first three rolls. Let's represent any 8's by e and 9's by n , and all other values by x . Now the first thing the program should do is calculate the probabilities of the different throws. I chose to write this program in python, and this first calculation is shown below.

```
# Number of rolls and sides on dice
die_rolls = 39
die_sides = 9

# Probability that result is lower than 8
lo_prob = (7 / die_sides)
# Probability that result a specific roll, here either 8 or 9
hi_prob = (1 / die_sides)

# Cases for the first 3 dice. 'e' is for 8, 'n' is for 9, and 'x' is for 1-7.
# The probability of the case is multiplied by the number of permutations.

# All cases where neither 8 nor 9 is included
xxx = ((7**3) / (9**3))
# Cases xx8 x8x 8xx
exx = (lo_prob**2 * hi_prob) * 3
# Cases xx9 x9x 9xx
```

```

nxx = (lo_prob**2 * hi_prob) * 3
# Cases 88x 8x8 x88
eex = (lo_prob * hi_prob**2) * 3
# Cases 99x 9x9 x99
nnx = (lo_prob * hi_prob**2) * 3
# Cases 98x 89x 8x9 9x8 x89 x98
nex = (lo_prob * hi_prob**2) * 6
# Cases 998 989 899
nne = (hi_prob**3) * 3
# Cases 889 898 988
nee = (hi_prob**3) * 3
# Case 888
eee = hi_prob**3

```

Now we have the probability for each of these combinations, and we need to add the remaining 36 rolls. For each roll we update the probability for each of the above cases, thus for example the probability that xxx remains the highest three dice is equal to the current possibility times `lo_prob`, as we need to count the cases where the new die is neither 8 nor 9.

The code for this is presented below.

```

for i in range(die_rolls - 3):
    tmp_xxx = xxx * lo_prob
    tmp_exx = (exx * lo_prob) + (xxx * hi_prob)
    tmp_nxx = (nxx * lo_prob) + (xxx * hi_prob)
    tmp_eex = (eex * lo_prob) + (exx * hi_prob)
    tmp_nnx = (nnx * lo_prob) + (nxx * hi_prob)
    tmp_nex = (nex * lo_prob) + (nxx * hi_prob) + (exx * hi_prob)
    tmp_eee = (eee * lo_prob) + (eex * hi_prob) + (eee * hi_prob)
    tmp_nne = (nne * lo_prob) + (nne * hi_prob) + (nee * hi_prob)
    tmp_nee = (nee * lo_prob) + (nee * hi_prob) + (eee * hi_prob)
    tmp_nex = (nex * lo_prob) + (nex * hi_prob) + (eex * hi_prob)

    xxx = tmp_xxx
    exx = tmp_exx
    nxx = tmp_nxx
    eex = tmp_eex
    nnx = tmp_nnx
    nex = tmp_nex
    eee = tmp_eee
    nne = tmp_nne
    nee = tmp_nee

```

```

# Print the result with 9 decimals
print("The probability is %.9f" % (eee + nee + nne))

```

When running this code, we get that the probability is 0.172550700 which is precisely equal to the result we got from the Troll code.

c)

The size of the Python program is remarkably bigger than the Troll program, and it was much more difficult to write. The length of the program also decreases understandability, as it is not immediately clear what the code does. When understanding the syntax of Troll, the program was easy to write, and easy for the reader to understand as well. Furthermore, the Troll program can easily be modified to calculate a different number of rolls, or different number of largest rolls. The python program, on the other hand, can relatively easily be change to compute a different number of rolls, but to change it to a different number of largest rolls would not be easy. Concerning runtime, it seems that both programs compute the result very fast, at least when dealing with such a small calculation as this.