CS558 Programming Languages – Fall 2023 – Study Questions Lecture 3a

These questions are intended for self-study, to help review and deepen your understanding of the lecture. Sample answers are available. There is nothing to hand in.

1. Consider the following expression in the language used in slides 5-13. Recall that `let` introduces a single simple non-recursive binding for a variable and `letfun` introduces one or more mutually recursive bindings for functions.

```
01   let a = 4
02   in letfun g c =
03               let a = c - 1
04               in let d = f(a)
05                  in (d := d * 2;   // assignment; updates the value of d
06                      letfun h e =
07                        e + d
08                      in h(7))
09      and    f b =
10               a + b
11      in letfun j() =
12           let b = g(a)
13           in b + a
14         in 42
```

(a) Identify all the places where an identifier is *bound*.

(b) Identify all the places where an identifier is *used*.

(c) We can describe the environment at any given location in the program by listing the set of variables and functions in scope at that location and the place where each was bound. For example, at the beginning of line 03, the environment is

{`a` : line 01, `c`: line 02, `g`: line 02, `f`: line 09 }

Write down the environment for (the beginning of) each line in the program.

(d) What are the free identifiers of the expression `let b = g(a) in b + a` ? Of function `f`? Of the entire expression above?

2. Do Scott exercise 3.14.

3. Slide 26 claims that initialization checking in Java is an undecidable problem in general; that is, there is no uniform way to determine, for any arbitrary program, whether all variables have been defined before they are used. (Note that we aren't allowed to just run the program to find out, because it might go into an infinite loop, and we want a definite yes-or-no answer.) The most famous undecidable problem is the *halting problem*. Alan Turing established that there is no algorithm (as the term has come to be generally understand) that can decide whether an arbitrary program halts or instead goes into an infinite loop. For languages like Java, this essentially amounts to deciding whether the `main` function returns, so an equivalent formulation is that there is no algorithm that can decide whether an arbitrary function returns or instead goes into an infinite loop.

Assuming this fact (you can Google "halting problem" if you haven't seen it before), a standard way to show that some other problem X is undecidable is to *reduce* the halting problem to it; that is, show that any instance of the halting problem can be converted to an instance of the X problem. Then, if we had a way to decide X, we could also decide about halting. Hence, since we cannot decide about halting, there must be no way to decide X either!

Give such an reduction (informally) from the halting problem to initialization checking. That is, suppose we have an arbitrary function `f` and we want to know whether it returns or not. Show how to construct another program `g` such that determining whether `g` uses a variable before it has been initialized would let us deduce whether `f` returns or not. (You can assume the language is Java, but that's not very important; any "Turing-complete" language will do.)