CS 558 Programming Languages Fall 2023 – Practice Final Exam Questions

This exam has six questions. Each question is worth 20 points. Some questions have multiple sub-parts, whose worth is indicated in brackets. You have 110 minutes for the exam. Please write your answers on the exam paper in the spaces provided.

You may use a single one-sided 8.5x11 sheet of handwritten notes. Otherwise, the exam is closed-book, closed-notes. No computers or other electronic devices are allowed. You must work independently, and you cannot share your notes with other students.

Write your name on your sheet of notes, and turn the sheet in with your exam when finished.

1. [20 pts.] **First class function implementation**

Consider the following Scala code.

```
1  def main() {
2    val c = 20
3    def f(x : Int) : Int => Int = {
4      def g(y : Int) =
5        x+y // what is on the stack here??
6      val a = g(c)
7      println(a)
8      return g
9    }
10   val h : Int => Int = f(c)
11   val b = h(1)
12   println(b)
13
14 }
```

(a) [4 pts.] What values should this program print for `a` and `b`?

(b) [4 pts.] List the names of any higher-order functions in this program.

(c) [4 pts.] List the free identifiers of function `f` and the free identifiers of function `g`.

(d) [4 pts.] Consider a hypothetical implementation of Scala that has these two characteristics:

- All values of functions, parameters and locals are stored on a stack and popped when they go out of scope. (Remember that Scala functions defined by `def` can be recursive.)
- All function values are represented by closures consisting of a code pointer and an environment mapping identifiers to locations.

List all the identifiers whose values are on the stack the *first* time that control reaches line 5.

(e) [4 pts.] Assuming the same implementation as in (d), now list all the identifiers whose values are on the stack the *second* time that control reaches line 5. Use this information to explain why this hypothetical stack-based Scala implementation does not work correctly.

2. [20 pts.] **Tail Recursion and Continuation-Passing Style**

Consider this function, given in Scala:

```
def f(x:Int) : Int = {
  if (x > 100)
    x-10
  else
    f(f(x+11))
}
```

(Note: This is called "McCarthy's 91-function," because it was invented by John McCarthy and returns 91 for all arguments ≤ 101.)

(a) [10 pts] The function makes two recursive calls, of which only one is tail-recursive. Rewrite the Scala code for `f` to remove the tail-recursive call, by replacing it with an appropriate `while` loop. Do *not* change the other recursive call. (Don't worry about making minor mistakes in Scala syntax.)

(b) [10 pts] Transform the Scala code of the *original* function into continuation-passing style, producing a function definition of the form

```
def g[A](x:Int,k:Int => A) : A = ...
```

such that `g(`e`,x => x)` returns the same result as `f(`e`)`. (Again, don't worry about making minor mistakes in Scala syntax.)

3. [20 pts.] **Type Inference**

Consider the following expressions written in a variant of our usual toy language syntax, including integers, booleans, and single-argument functions (constructed with `fun` and applied with `@`). The grammar of types is

$$\tau ::= a_i \mid \texttt{int} \mid \texttt{bool} \mid (\tau_1 \rightarrow \tau_2)$$

where type variables $a_1, a_2, a_3, \ldots$ denote arbitrary unknown types (i.e. universally quantified types).

Give the *most general* possible type for each entire expression, or state that the expression is ill-typed. For example, the most general type of `(fun x 42)` is $(a_1 \rightarrow \texttt{int})$. Hint: To infer the types you can extract and solve constraints, but this is not necessary; it is probably easier to figure them out informally.

(a) [4 pts.] `(@ (fun x x) 42)`

(b) [4 pts.] `(fun x (fun y x))`

(c) [4 pts.] `(fun x (@ x true))`

(d) [4 pts.] `(fun x (@ x (+ x 1)))`

(e) [4 pts.] `(fun x (fun y (not (@ x y))))`

4. [20 pts.] **Typing of Match Expressions**

In lecture, we examined formal typing rules of the form $TE \vdash e : t$ for a simple expression language, and in lab we considered typing of lists with `nil`, `cons`, `isnil`, `head`, and `tail` expression forms. Formal rules for an expanded language with these list expressions are listed at the bottom of this page.

Suppose we add a new expression form `(match` e e_n x_h x_t e_c`)` similar to the `match` construct of Scala. To evaluate this expression, first e is evaluated in the current environment to a value v, which should be a list value (if not, the `match` evaluation fails with an error). If v is the empty list, e_n is evaluated in the current environment, and the resulting value is the value of the whole `match`. If v is a non-empty list, the current environment is extended to bind x_h to the head of the v and x_t to the tail of v, and e_c is evaluated in that extended environment to produce the value of the whole `match`.

Note that once the language has `match`, we no longer need the `isnil`, `head`, and `tail` expression forms, since we can use `match` instead. For example, assuming we have recursive function definitions (not actually covered by the rules below), the function definition

```
(sum (l)
   (if (isnil l)
       0
       (+ (head l) (@ sum (tail l)))))
```

can be rewritten as

```
(sum (l)
   (match l
          0
          h t (+ h (@ sum t))))
```

Give a formal typing rule for the new `match` expression form.

$$\frac{TE(x) = t}{TE \vdash x : t} \text{ (Var)}$$

$$\frac{}{TE \vdash i : \texttt{int}} \text{ (Int)}$$

$$\frac{}{TE \vdash \texttt{true} : \texttt{bool}} \text{ (True)}$$

$$\frac{}{TE \vdash \texttt{false} : \texttt{bool}} \text{ (False)}$$

$$\frac{TE \vdash e_1 : \texttt{int} \quad TE \vdash e_2 : \texttt{int}}{TE \vdash \texttt{(+ } e_1 \; e_2\texttt{)} : \texttt{int}} \text{ (Add)}$$

$$\frac{TE \vdash e_1 : \texttt{int} \quad TE \vdash e_2 : \texttt{int}}{TE \vdash \texttt{(<= } e_1 \; e_2\texttt{)} : \texttt{bool}} \text{ (Leq)}$$

$$\frac{TE \vdash e_1 : t_1 \quad TE + \{x \mapsto t_1\} \vdash e_2 : t_2}{TE \vdash \texttt{(local } x \; e_1 \; e_2\texttt{)} : t_2} \text{ (Local)}$$

$$\frac{TE(x) = t \quad TE \vdash e : t}{TE \vdash \texttt{(:= } x \; e\texttt{)} : t} \text{ (Assgn)}$$

$$\frac{TE \vdash e_1 : \texttt{bool} \quad TE \vdash e_2 : t \quad TE \vdash e_3 : t}{TE \vdash \texttt{(if } e_1 \; e_2 \; e_3\texttt{)} : t} \text{ (If)}$$

$$\frac{TE \vdash e_1 : \texttt{bool} \quad TE \vdash e_2 : t}{TE \vdash \texttt{(while } e_1 \; e_2\texttt{)} : \texttt{int}} \text{ (While)}$$

$$\frac{TE + \{x \mapsto t_1\} \vdash e : t_2}{TE \vdash \texttt{fun } x \texttt{ -> } e : t_1 \to t_2} \text{ (Fn)}$$

$$\frac{TE \vdash e_1 : t_1 \to t_2 \quad TE \vdash e_2 : t_1}{TE \vdash e_1 e_2 : t_2} \text{ (Appl)}$$

$$\frac{}{TE \vdash \texttt{(nil } t\texttt{)} : \texttt{(list } t\texttt{)}} \text{ (Nil)}$$

$$\frac{TE \vdash e_1 : t \quad TE \vdash e_2 : \texttt{(list } t\texttt{)}}{TE \vdash \texttt{(cons } e_1 \; e_2\texttt{)} : \texttt{(list } t\texttt{)}} \text{ (Cons)}$$

$$\frac{TE \vdash e : \texttt{(list } t\texttt{)}}{TE \vdash \texttt{(isnil } e\texttt{)} : \texttt{bool}} \text{ (IsNil)}$$

$$\frac{TE \vdash e : \texttt{(list } t\texttt{)}}{TE \vdash \texttt{(head } e\texttt{)} : t} \text{ (Head)}$$

$$\frac{TE \vdash e : \texttt{(list } t\texttt{)}}{TE \vdash \texttt{(tail } e\texttt{)} : \texttt{(list } t\texttt{)}} \text{ (Tail)}$$

5. [20 pts.] **Variant Records**

Consider the following Haskell program fragment:

```
data T = P Int T T
       | Q Bool

f :: T -> Int
f x =
  case x of
    P i u v -> i * f u + f v
    Q b -> if b then 1 else 0

h :: T -> T -> Int
h x1 x2 = f x1 + f x2
```

Note that functions and constructors in Haskell are all Curried, and they can be applied without putting parentheses around their arguments. As an example, just to check your understanding:

`h (Q True) (P 3 (Q True) (Q False))` evaluates to `4`.

Suppose we wish to write a more-or-less equivalent object-oriented Scala program fragment, but using *dynamic dispatch* in place of pattern matching over constructors. Type `T` will be represented by

```
abstract class T {
  def f() : Int
}
```

with two subclasses `P` and `Q`. Function `h` will be implemented as

```
def h(x1:T,x2:T) : Int = x1.f() + x2.f()
```

and we want `h(Q(true),P(3,Q(true),Q(false)))` to evaluate to `4`.

Give definitions for the subclasses `P` and `Q`.

Note: Of course, it would be possible to reproduce the Haskell program's pattern matching using Scala pattern matching, but that is not what this question asks you to do; you must *not* use pattern matching!

6. [10 pts.] **Nested Procedures and Objects**

There is a relationship between nested functions in a procedural language and class member functions in an object-oriented language, under which the fields of the class correspond to the free variables of the functions.

Consider the following Scala code:

```
case class M(x:Boolean, y:Int, z:Int) {
  def f(w:Int) = if (x)
                   z + w
                 else
                   w - 42
  def g(w:Int) = w + y
}

def h() = {
  val m = M(true,10,20)
  m.f(30) + m.g(40)
}
```

Calling `h()` returns the answer `100`.

(a) [2 pts.] Identify the free variables of function `f` and the free variables of function `g`.

(b) [8 pts.] Let `R` be the record type defined by

```
case class R(f: Int => Int, g: Int => Int)
```

Write down a Scala function

```
M : (x:Boolean,y:Int,z:Int) => R
```

such that, if we replace the class definition `M` above by your function definition `M`, function `h` still behaves the same way (e.g. calling it still returns `100`).