CS143 - Written Assignment 3

Sample Solutions

1. **(9 pts)** The following is the implementation of the Main class of a cool program:

```
class Main {
  b : B;
  main() : Int {{
    b <- new B;
    b.h();
    2;
  }};
};
```

Now consider the following implementations of the classes A and B. Analyze each version of the classes to determine if the resulting program will pass type checking and, if it does, whether it will execute without runtime errors. Please include a brief (1 - 2 sentences) explanation along with your answer.

(a) **(3 pts)**

```
class A {
  i : Int;
  a : SELF_TYPE;
};

class B inherits A {
  h() : A {
    if (i <- 1) < 2 then a <- new A else a <- new B fi
  };
};
```

Answer: Does not pass type checking. Neither A nor B conform to $SELF\_TYPE_B$, and therefore the assignments `a <- new A` and `a <- new B` are invalid.

(b) **(3 pts)**

```
class A {
  i : Int;
  a : A;
};

class B inherits A {
  h() : A {
    case a of
      k: A => a;
      b: B => b;
      o: Object => a;
    esac
  };
};
```

Answer: Passes type checking but encounters a runtime error. The case is performed on a void value (a is uninitialized).

(c) **(3 pts)**

```
class A {
  i : Int;
  a : A;
  d : SELF_TYPE;
};

class B inherits A {
  h() : A {{
    d <- self;
    case d of
      k: A => d;
      b: B => d;
      o: Object => d;
    esac;
  }};
};
```

Answer: Passes type checking and successfully runs.

2. **(8 pts)**

(a) **(4 pts)** Consider the following program in Cool (using standard Cool type rules, scoping rules and general semantics). Provide the output of the labelled statement in Main.main() and explain why it prints that value.

```
1    class A {
2      f1(): Int {
3        let a: Int in {
4            a <- x + 1;
5            a;
6        }
7      };
8      f2(): Int {
9        let a: Int in {
10         { a <- 2; };
11         x <- a;
12         f1();
13         a + x;
14       }
15     };
16     x: Int <- 0;
17     a: Int <- 1;
18   };
19   class Main {
20     main(): Object {
21       let o: A, io: IO <- New IO in {
22         o <- new A;
23         io.out_int(o.f2()); -- Statement
24       }
25     };
26   };
```

Answer: Inside function f2() all the references to a refer to the let binding; thus the value of the attribute a is never modified. The block on line 10 does not open a new scope, so on line 10 a gets value 2 and on line 11 x gets value 2. The call to f1 does not modify the attribute x, and therefore on line 13 $a + x$ has the value 4, which ultimately is printed on line 23.

(b) **(4 pts)** In the following program, suppose [Placeholder B] will be filled by an integer literal that is unknown to you. Can you replace [Placeholder A] with a Cool expression that will allow you to predict the output of the labelled statement? If you are able to do so, provide your replacement for [Placeholder A] and the output of the statement. If you cannot, explain why.

```
1   class Main {
2     main(): Object {
3       let io: IO <- New IO, z: Int (* <- [Placeholder B] *) in {
4         let w: Int <- 2 in {
5           let z: Int <- w in {
6             (* [Placeholder A] *)
7             io.out_string("The secret will be: ");
8             io.out_int(z);
9           };
10        };
11        io.out_string("The secret is: ");
12        io.out_int(z); -- Statement
13      }
14    };
15  };
```

Answer: No expression on line 6 can predict the secret output on line 11. On line 6, the only objects in the environment are `self`, `io`, `w` and the object `z` bound on line 5. None of these objects contain any attribute which points to the object `z` bound on line 3; as a result, it is not possible for any expression on line 6 to compute this value.

3. **(12 pts)** Type derivations are expressed as inductive proofs in the form of trees of logical expressions. For example, the following is the type derivation for $O[Int/y] \vdash y + y : Int$:

$$\dfrac{\dfrac{O[Int/y](y) = Int}{O[Int/y], M, C \vdash y : Int} \qquad \dfrac{O[Int/y](y) = Int}{O[Int/y], M, C \vdash y : Int}}{O[Int/y], M, C \vdash y + y : Int}$$

Consider the following Cool program fragment:

```
class A {
  a: Int;
  b: Int;
  c: Int;
  yes: Bool;
  foo(): SELF_TYPE { self };
  bar(k : A) : Bool { if a < 0 then yes else false fi };
};
class B inherits A {
  me: SELF_TYPE;
  test(): Object { (* [Placeholder] *) };
};
```

Note that the environments O and M at the start of the method test(...) are as follows:

$$O = \emptyset[\text{Int}/a][\text{Int}/b][\text{Int}/c][\text{Bool}/yes][\text{SELF\_TYPE}_B/me]$$

$$\begin{aligned} \text{M(A, foo)} &= \text{(SELF\_TYPE)} \\ \text{M(A, bar)} &= \text{(A, Bool)} \\ \text{M(B, foo)} &= \text{(SELF\_TYPE)} \\ \text{M(B, bar)} &= \text{(A, Bool)} \\ \text{M(B, test)} &= \text{(Object)} \end{aligned}$$

For each of the following expressions replacing [Placeholder], provide the type derivation and final type of the expression, if it is well typed, otherwise explain why it isn't. Assume Cool type rules (you may omit subtyping relationships from the rules when the type is the same, e.g. Bool ≤ Bool).

(a)

```
a <- c + { if a < b then a else b fi; }
```

Answer:

$$\dfrac{O(a) = \text{Int} \qquad \dfrac{\dfrac{O(c) = \text{Int}}{O, M, B \vdash c : \text{Int}} \qquad \dfrac{\dfrac{\dfrac{\dfrac{O(a) = \text{Int}}{O, M, B \vdash a : \text{Int}} \quad \dfrac{O(b) = \text{Int}}{O, M, B \vdash b : \text{Int}}}{O, M, B \vdash a < b : Bool}\;\text{Compare} \quad \dfrac{O(a) = Int}{O, M, B \vdash a : Int} \quad \dfrac{O(b) = Int}{O, M, B \vdash b : Int}\;\text{Var}}{O, M, B \vdash \text{ if } a < b \text{ then } a \text{ else } b \text{ fi } : \text{Int}}\;\text{If}}{O, M, B \vdash \{ \text{ if } a < b \text{ then } a \text{ else } b \text{ fi };\} : \text{Int}}\;\text{Sequence}}{O, M, B \vdash c + \{ \text{ if } a < b \text{ then } a \text{ else } b \text{ fi };\} : \text{Int}}\;\text{Arith}}{O, M, B \vdash a \leftarrow c + \{ \text{ if } a < b \text{ then } a \text{ else } b \text{ fi };\} : \text{Int}}\;\text{Assign}$$

(b)

```
        me.bar(me.foo())
```

Answer:

$$\dfrac{\dfrac{O(me) = \text{ SELF\_TYPE}_B}{O, M, B \vdash me : \text{ SELF\_TYPE}_B}\,\text{Var} \qquad \dfrac{\dfrac{O(me) = \text{ SELF\_TYPE}_B}{O, M, B \vdash me : \text{ SELF\_TYPE}_B}\,\text{Var} \quad M(B, foo) = (\text{SELF\_TYPE})}{O, M, B \vdash me.foo() : \text{ SELF\_TYPE}_B}\,\text{Dispatch} \qquad M(B, bar) = (A,\ \text{Bool}\ ) \quad \text{SELF\_TYPE}_B \leq A}{O, M, B \vdash me.bar(me.foo()) : Bool}\,\text{Dispatch}$$

(c)

```
        me.bar(me <- new B)
```

Answer: This is not well typed because B does not conform to SELF_TYPE_B. Thus the assignment `me <- new B` is not valid.

4. **(16 pts)** Consider the following extension to the Cool syntax as given on page 16 of the Cool Manual, which adds arrays to the language:

$$
\begin{aligned}
\text{expr} ::= &\ \text{new TYPE[expr]} \\
&|\ \text{expr[expr]} \\
&|\ \text{expr[expr]}\ < -\ \text{expr}
\end{aligned} \tag{1}
$$

This adds a new type T[] for every type T in Cool, including the basic classes. Note that the entire hierarchy of array types still has Object as its topmost supertype. An array object can be initialized with an expression similar to "my_array:T[] ← new T[n]", where n is an Int indicating the size of the array. In the general case, any expression that evaluates to an Int can be used in place of n. Thereafter, elements in the array can be accessed as "my_array[i]" and modified using a expression like "my_array[i] ← value".

(a) **(4 pts)** Provide new typing rules for Cool which handle the typing judgments for: $O, M, C \vdash$ new T[e_1], $O, M, C \vdash e_1[\ e_2\ ]$ and $O, M, C \vdash e_1[\ e_2\ ]\ < -\ e_3$. The type of the expression $e1[e2]\ < -\ e3$, if well-typed, should be the type of e3. Make sure your rules work with subtyping.
Answer:

$$\frac{O, M, C \vdash e_1 : \text{ Int}}{O, M, C \vdash \text{ new } T[e_1] : T[]}$$

$$\frac{\begin{array}{c} O, M, C \vdash e_1 : T[] \\ O, M, C \vdash e_2 : \text{ Int} \end{array}}{O, M, C \vdash e_1[e_2] : T}$$

$$\frac{\begin{array}{c} O, M, C \vdash e_1 : T'[] \\ O, M, C \vdash e_2 : \text{ Int} \\ O, M, C \vdash e_3 : T \\ T \leq T' \end{array}}{O, M, C \vdash e_1[e_2] \leftarrow e_3 : T}$$

It's also possible to write these rules allowing for SELF_TYPE in place of T. However, arrays of SELF_TYPE are not incredibly useful because the subtyping rule is very restrictive (as seen in the next part).

(b) **(4 pts)** Consider the following subtyping rule for arrays:

$$\frac{T_1 \leq T_2}{T_1[\ ] \leq T_2[\ ]}$$

This rule means that $T_1[\ ] \leq T_2[\ ]$ whenever it is the case that $T_1 \leq T_2$, for any pair of types T_1 and T_2.

While plausible on first sight, the rule above is incorrect, in the sense that it doesn't preserve Cool's type safety guarantees. Provide an example of a Cool program (with arrays added) which would type check when adding the above rule to Cool's existing type rules, yet lead to a type error at runtime. The error should be related to the rules for arrays and not be among the kinds of runtime errors defined on page 29 of the Cool manual.

Answer: Consider the following Cool program,

```
let x : Int[], y : Object[] in {
  x <- new Int[10];
  y <- x;
  y[0] = "abc";
  out_int(x[0]+1);
}
```

Because the rule implies that Int[] $\leq$ Object[] we can reference an array of integers, x, with a variable, y, whose type is Object[]. Because y is an array of objects, we can add a string to it. However, that string is added to the underlying array of integers! Then, if we try and perform an operation like addition on an element of the integer array, we encounter an error at runtime.

(c) **(4 pts)** Suppose you wish to extend the subtyping rule as far as possible while preserving the soundness of the typesystem. What are all types X such that $T[] \leq X$? What are all types Y such that $Y \leq T[]$?

Answer:

We can only say that $T[] \leq T[]$ and $T[] \leq Object$.

(d) **(4 pts)** Add another extension to the language for immutable arrays (denoted by the type T()). Immutable arrays are initialized by providing a list of expressions for the array to hold. After initialization, the contents of an immutable array may not be modified. Analogous to questions 4a and 4c, for this extension, provide: the additional syntax constructs to be added to the listing of page 16 of the Cool manual, the typing rules for these constructs and the least restrictive subtyping relationship involving these tuple types. It is not necessary that this extension interact correctly with mutable arrays as defined above, but feel free to consider that situation.

Answer:

New syntax:

$$\begin{array}{rcl} expr & ::= & \text{new TYPE}(expr\_list) \\ & | & expr[expr] \\ expr\_list & ::= & expr \\ & | & expr, expr\_list \end{array}$$

It's also acceptable to have immutable arrays of length zero. New typing rules:

$$\frac{\begin{array}{c} O, M, C \vdash e_i : T_i \qquad 1 \leq i \leq n \\ T_i \leq T \qquad 1 \leq i \leq n \end{array}}{O, M, C \vdash \text{ new } T(e_1, e_2, \ldots, e_n) : T()}$$

$$\frac{\begin{array}{c} O, M, C \vdash e_1 : T() \\ O, M, C \vdash e_2 : \text{ Int} \end{array}}{O, M, C \vdash e_1[e_2] : T}$$

Lastly, since the arrays are immutable, there's no problem with the subtyping rule proposed earlier,

$$\frac{T_1 \leq T_2}{T_1() \leq T_2()}$$

5. **(8 pts)** Consider the following assembly language used to program a stack machine (r, r1, r2 denote arbitrary registers):

```
push r: pushes the value in r onto the stack
top r: copies the value at the top of the stack into r. This
    command does not modify the stack.
pop: discards the value at the top of the stack
r1 += r2: adds r1 and r2 and saves the result in r1. r1 may be the
     same as r2.
r1 -= r2: subtracts r2 from r1 and saves the result in r. r1 may
    be the same as r2.
jump r: jumps to the line number in r and resumes execution
print r: prints the value in r to the console
```

The machine has three registers available to the program: reg1, reg2, and reg3. The stack is permitted to grow to a finite, but very large, size. If an invalid line number is invoked, pop is executed on an empty stack, or the maximum stack size is exceeded, the machine crashes.

(a) **(4 pts)** Write code to enumerate the Fibonnacci series (0, 1, 1, 2, ...) without termination. Assume that the code will be placed at line 100, and will be invoked by setting reg1, reg2, and reg3 to 100, 0, 1 respectively and running 'jump reg1'. Your code should use the 'print' opcode to display numbers in the series.

Answer:

Note that several possible solutions exist to this problem.

```
100   print reg2
101   print reg3
102   reg2 += reg3
103   reg3 += reg2
104   jump reg1
```

(b) **(4 pts)** This 'helper' function is placed at line 1000:

```
1000      push reg1
1001      reg1 -= reg2
1002      reg2 -= reg1
1003      reg3 += reg2
1004      top reg1
1005      pop
1006      jump reg1
```

This 'main' procedure is placed at line 2000:

```
2000      push reg1
2001      push reg3
2002      top reg1
2003      top reg3
2004      pop
2005      pop
2006      jump reg2
2007      print reg3
2008      jump reg2
```

reg1, reg2, reg3 are set to 0, 1000 and 2000 respectively, and 'jump reg3' is executed. What output does the program generate? Does it crash? If it does, suggest a one-line change to the helper function that results in a program that does not crash.

Answer:

The program crashes the second time it reaches line 2006 when it tries to jump to address 0. It does not generate any output. Here is a trace of the execution before the crash:

```
2000: reg1=0 reg2=1000 reg3=2000 stack=[]
2001: reg1=0 reg2=1000 reg3=2000 stack=[0]
2002: reg1=0 reg2=1000 reg3=2000 stack=[2000, 0]
2003: reg1=2000 reg2=1000 reg3=2000 stack=[2000, 0]
2004: reg1=2000 reg2=1000 reg3=2000 stack=[2000, 0]
2005: reg1=2000 reg2=1000 reg3=2000 stack=[0]
2006: reg1=2000 reg2=1000 reg3=2000 stack=[]
1000: reg1=2000 reg2=1000 reg3=2000 stack=[]
1001: reg1=2000 reg2=1000 reg3=2000 stack=[2000]
1002: reg1=1000 reg2=1000 reg3=2000 stack=[2000]
1003: reg1=1000 reg2=0 reg3=2000 stack=[2000]
1004: reg1=1000 reg2=0 reg3=2000 stack=[2000]
1005: reg1=2000 reg2=0 reg3=2000 stack=[2000]
1006: reg1=2000 reg2=0 reg3=2000 stack=[]
2000: reg1=2000 reg2=0 reg3=2000 stack=[]
2001: reg1=2000 reg2=0 reg3=2000 stack=[2000]
2002: reg1=2000 reg2=0 reg3=2000 stack=[2000, 2000]
2003: reg1=2000 reg2=0 reg3=2000 stack=[2000, 2000]
2004: reg1=2000 reg2=0 reg3=2000 stack=[2000, 2000]
2005: reg1=2000 reg2=0 reg3=2000 stack=[2000]
2006: reg1=2000 reg2=0 reg3=2000 stack=[]
```

The only way to prevent the program from crashing is to force it to enter an infinite loop. One way to do this is to replace the first instruction of the helper with `jump reg2`.