CS143 - Written Assignment 3 Reference Solutions

1. The following is the implementation of the Main class of a cool program:

```
1       class Main {
2         b: B;
3         main(): Int {{
4           b <- new B;
5           b.foo();
6           2;
7         }};
8       };
```

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) Implementation 1:

```
1         class A {
2           i : Int;
3           a : SELF_TYPE;
4         };
5
6         class B inherits A {
7           foo() : B {
8             if isvoid a then a <- new B else a fi
9           };
10        };
```

Answer: Does not pass type checking. `B` does not conform to SELF_TYPE_B and therefore the assignment `a <- new B` is invalid.

(b) Implementation 2:

```
1         class A {
2           i: Int;
3           a: A;
4           bar(): A {
5             a
6           };
7         };
8
9         class B inherits A {
10          foo(): A {
11            if i < 0 then a else a.bar() fi
12          };
13        };
```

Answer: Passes type checking but encounters a runtime error. The method `A.bar()` is dispatched from a void value (`a` is uninitialized).

(c) Implementation 3:

```
1         class A {
2           i: Int;
3           a: A;
4           bar(): A {
5             a
6           };
7         };
8
9         class B inherits A {
10          foo(): A {{
11            a <- self;
12            if i < 0 then a else a.bar() fi;
13          }};
14        };
```

Answer: Passes type checking and successfully runs.

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

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

Consider the following Cool program fragment:

```
1     class A {
2         i: Int;
3         j: Int;
4         k: Int;
5         yes: Bool;
6         foo(): SELF_TYPE { self };
7         bar(x : Int): Int { if x <= k then k <- i + k else k <- j fi
              };
8     };
9     class B inherits A {
10       p: SELF_TYPE;
11       test(): Object { (* [Placeholder C] *) };
12    };
```

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

$$O = \emptyset[\text{Int}/i][\text{Int}/j][\text{Int}/k][\text{Bool}/yes][\text{SELF\_TYPE}_B/p][\text{SELF\_TYPE}_B/self]$$

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

For each of the following expressions replacing [Placeholder C], 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 $\leq$ Bool).

(a) Substitution 1:

1 `p.bar(j <- 5)`

Answer:

$$\dfrac{\dfrac{O(p) = \text{SELF\_TYPE}_B}{O, M, B \vdash p : \text{SELF\_TYPE}_B} \quad \dfrac{O(j) = Int \quad \dfrac{5 \text{ is an integer constant}}{O, M, B \vdash 5 : Int}}{O, M, B \vdash j \leftarrow 5 : Int} \quad M(B, bar) = (Int, Int)}{O, M, B \vdash p.bar(j \leftarrow 5) : Int}$$

The final type is **Int**.

(b) Substitution 2:

```
1          if j = k then p <- p.foo() else p <- new B fi
```

Answer: This expression will not type check since `p <- new B` presents a type error. The reason is that B does not conform to SELF_TYPE_B and therefore the assignment `p <- new B` is invalid.

(c) Substitution 3:

```
1          let s: Int <- {if j < k then j else k fi;} in s
```

Answer:

$$\dfrac{\dfrac{\dfrac{\dfrac{O(j) = Int}{O, M, B \vdash j : Int} \quad \dfrac{O(k) = Int}{O, M, B \vdash k : Int}}{O, M, B \vdash j < k : Bool} \quad \dfrac{O(j) = Int}{O, M, B \vdash j : Int} \quad \dfrac{O(k) = Int}{O, M, B \vdash k : Int}}{O, M, B \vdash \text{if } j < k \text{ then } j \text{ else } k \text{ fi} : Int}}{O, M, B \vdash \{\text{if } j < k \text{ then } j \text{ else } k \text{ fi;}\} : Int} \quad \dfrac{O[Int/s](s) = Int}{O[Int/s], M, B \vdash s : Int}}{O, M, B \vdash \text{let } s\text{: } Int \leftarrow \{\text{if } j < k \text{ then } j \text{ else } k \text{ fi;}\} \text{ in } \ s : Int}$$

The final type is **Int**.

3. (a) Consider the following program in Cool (using standard Cool type rules, scoping rules and general semantics). Provide the output of each of the labeled statements in Main.main() and explain for each statement why it prints that value.

```
1       class A {
2           i: Int <- j + k;
3           j: Int <- i + 5;
4           k: Int <- i + j;
5           f1(): Int {
6               let i: Int in {
7                   i <- j + k;
8                   i;
9               }
10          };
11          f2(): Int {
12              let i: Int <- i in {
13                  j <- j + 2;
14                  i <- f1() + f1();
15              }
16          };
17          f3(): Int { i };
18      };
19      class Main {
20          main(): Object {
21              let o: A <- new A, io: IO <- new IO in {
22                  io.out_int(o.f1());  -- Statement 1
23                  io.out_int(o.f2());  -- Statement 2
24                  io.out_int(o.f3());  -- Statement 3
25              }
26          };
27      };
```

Answer: Due to how attribute initialization in Cool is handled, at first `i`, `j`, and `k` are all assigned a default value of 0 before initialization. Afterwards, attribute initialization happens sequentially, so `i` has value 0, `j` has value 5, and `k` has value 5 after initialization. With the above in mind:

i. Statement 1 prints 10. It prints the value of method `A.f1()`, which is `j + k`. The values of `j` and `k` both 5, so `j + k` has value 10. Notice here that the attribute `i` of class `A` still has value 0 since `i` is redefined in the let expression of `f1`.

ii. Statement 2 prints 24. It prints the value of method `A.f2()`, which first sets the value of attribute `j` of class `A` to 7 before calling method `A.f1()` twice and summing their values together. After the method call of `A.f2()`, the value of `j` remains as 7 while the value of attribute `i` is still 0 (`i` is redefined in the let expression of `A.f2()` before it gets assigned a value).

iii. Statement 3 prints 0. It prints the value of method `A.f3()`, which simply returns the value of attribute `i`. Since the value of `i` does not change after calling methods `A.f1()` and `A.f2()`, the value of `i` is that after initialization, which we have seen earlier to be 0.

(b) 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 statement 1 to print the unknown integer (output of statement 2)? If you are able to do so, provide your replacement for [Placeholder A]. 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             (* [Placeholder A] *)
6             let z: Int <- w in {
7               io.out_string("The secret will be: ");
8               io.out_int(z);  -- Statement 1
9             };
10          };
11          io.out_string("The secret is: ");
12          io.out_int(z);  -- Statement 2
13        }
14      };
15    };
```

Answer: We may replace [Placeholder A] with the following:

```
1           w <- z;
```

As a result, once the innermost `z` variable is introduced, it will be assigned the value of `w`, which holds the desired value to be printed at statement 2.

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

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

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) 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. Make sure your rules work with subtyping.

Answer:
ArrayNew:

$$\frac{O, M, C \vdash e_1 : T_1 \quad T_1 \leq Int \quad T' = \begin{cases}\text{SELF\_TYPE}_C \ if\ T = \text{SELF\_TYPE}\\ T \text{ otherwise}\end{cases}}{O, M, C \vdash \text{new T[} e_1 \text{] } : T'[]}$$

Note that you may alternatively chose to disallow SELF_TYPE as the type of a new array. Additionally, the Cool Manual states that it is an error to inherit from Int, so the only subtype of Int is Int. With this in mind, the above rule can also be given in a simplified form as:

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

ArrayLoad:

$$\frac{O, M, C \vdash e_1 : T[] \quad O, M, C \vdash e_2 : T_2 \quad T_2 \leq Int}{O, M, C \vdash e_1[\ e_2\ ] : T}$$

Again, we can alternatively list T_2 as Int directly, since Int cannot be subclassed. Note that whether or not we allow the expression "new SELF_TYPE[ ]", we don't need to check for SELF_TYPE here, since our rule for New already handles that case and gives the array a SELF_TYPE_C for the specific class C. There are some subtleties that arise if we want to allow a method to return a SELF_TYPE[ ] value or take arguments of that type as arguments. However, as we will see soon, array subtyping is more restrictive than normal class subtyping, making SELF_TYPE[ ] a lot less useful than SELF_TYPE.

ArrayStore:

$$\frac{O, M, C \vdash e_1 : T[] \quad O, M, C \vdash e_2 : T_2 \quad T_2 \leq Int \quad O, M, C \vdash e_3 : T_3 \quad T_3 \leq T}{O, M, C \vdash e_1[\ e_2\ ] \ <- e_3 : T_3}$$

Note that we assign the whole expression the type of e_3. This is not specified by the description of our array extension in itself, and is not the only valid answer for this exercise, but it most closely resembles the rule for ASSIGN.

(b) 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.

Answer:

```
1     class A { };
2     class B inherits A {
3         g(): Int { 1 };
4     };
5     class Main {
6         va: A[];
7         vb: B[] <- New B[1];
8         main(): Int {{
9             va <- vb;
10            va[0] <- New A;
11            vb[0].g(); -- error
12        }};
13    };
```

This will type check at compile time, since each of the statements in Main.main() type checks correctly: the first uses the subtyping rule given above and the standard ASSIGN rule to assign a B[] array to va, which has type A[]; the second simply initializes va[0] with a new A object; and the last one retrieves the first object (of static type B) in the array vb and calls g() on it, which is a valid method for B. The runtime error arises from the fact that vb[0] = va[0] has an actual type of A, not B. Because we can assign a mutable array of class B to a mutable array of the superclass A, we can end up with an array containing objects that don't match the static type of the array, which will violate our assumptions the moment we extract those objects from the array and try to use them.

(c) In the format of the subtyping rule given above, provide the least restrictive rule for the relationship between array types (i.e. under which conditions is it true that $T_1[] \leq T'$ for a certain T' or $T'' \leq T_1[]$ for a certain T''?) which preserves the soundness of the type system. The rule you introduce must not allow assignments between non-array types that violate the existing subtyping relations of Cool.

Answer:

The least restrictive rule for the relationship between mutable array types is as follows:

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

An array type is only a subtype of another array type if their allowable contents are of the same type (which implies that they are the exact same array type: $T_1[\,] = T_2[\,]$). Additionally, every array is a subtype of Object and has no subtyping relationship with any other non-array type.

This typing rule for mutable arrays is generally referred to as invariant typing. The rule of the last exercise is called covariant typing and is incorrect for arrays. Mutable arrays are neither covariant, nor contravariant (a related rule in which a dependent type X[[T]] is only a subtype of another X[[G]] if T is a supertype of G).

(d) Add another extension to the language for immutable arrays (denoted by the type T()). 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:

Syntax: Note that for immutable arrays, we must somehow combine the rule for New with a rule that sets the values of all elements in the array. The syntax for array loading can be identical to that of the mutable case.

$$\begin{aligned} \text{expr} ::= &\text{ new TYPE[ ]} < - \{[\text{expr}[, \text{expr}]^*]\} \\ &| \text{ expr[expr]} \end{aligned}$$

Typing rules:

$$\frac{\forall i \in [1,n] \quad O, M, C \vdash e_i : T_i \quad T_i \leq T}{O, M, C \vdash \text{new T[ ]} < - \{e_1, ...e_n\} : T()}$$

$$\frac{O, M, C \vdash e_1 : T() \quad O, M, C \vdash e_2 : T_2 \quad T_2 \leq Int}{O, M, C \vdash e_1[\ e_2\ ] : T}$$

Subtyping rules: Because immutable arrays don't allow modification using a reference of a subtype, the issue shown in question 4b does not arise. Thus, immutable arrays are covariant:

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

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

```
1     push r: copies the value of r and pushes it onto the stack.
2     top r: copies the value at the top of the stack into r.  This
         command does not modify the stack.
3     pop: discards the value at the top of the stack
4     r1 *= r2: multiplies r1 and r2 and saves the result in r1.  r1 may
          be the same as r2.
5     r1 /= r2: divides r1 with r2 and saves the result in r1.  r1 may
         be the same as r2.  remainders are discarded (e.g., 5 / 2 = 2).
6     r1 += r2: adds r1 and r2 and saves the result in r1.  r1 may be
         the same as r2.
7     r1 -= r2: subtracts r2 from r1 and saves the result in r1.  r1 may
          be the same as r2.
8     jump r: jumps to the line number in r and resumes execution.
9     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) Write code to enumerate the factorial number sequence, beginning with 1! (1, 2, 6, 24, ...), without termination. Assume that the code will be placed at line 100, and will be invoked by setting `reg1`, `reg2`, and `reg3` to 100, 1, and 1 respectively and running '`jump reg1`'. Your code should use the '`print`' opcode to display numbers in the sequence. You may not hardcode constants nor use any other instructions besides the ones given above.

Answer: Note that several possible solutions exist to this problem. One is the following:

```
100       reg2 *= reg3
101       print reg2
102       push reg2
103       reg2 /= reg2
104       reg3 += reg2
105       top reg2
106       pop
107       jump reg1
```

(b) 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`, and `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=[]
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`'.