CS143 Compilers 2020 - Written Assignment 3 Solution

1. Consider the following Cool programs:

```
1        class A {
2            x: A; -- line 2
3            baz() : A {{x <- new A; x;}}; -- line 3
4            bar(): A {new A}; -- line 4
5            foo(): String {"World!"};
6        };
7        class B inherits A {
8            foo() : String {" "};
9        };
10
11       class C inherits A {
12           foo() : String {"Hello,"};
13       };
14
15       class Main {
16           main (): Object {
17               let io : IO <- new IO, b : B <- new B, c : C <- new C
                     in {{
18                   io.out_string (c.baz().foo());
19                   io.out_string(b.baz().baz().foo());
20                   io.out_string (b.bar().baz().foo());
21               }}
22           };
23       };
```

(a) What does this code currently print? By changing only the A types on at most lines 2, 3, and 4 to get this program to print "Hello, World!".

Answer:
This program currently prints "World!World!World!".
The following modification will get the program to print "Hello, World!":

```
2            x: SELF_TYPE; -- line 2
3            baz() : SELF_TYPE {{x <- new SELF_TYPE; x;}}; -- line 3
```

```
1     class Main {
2       main (): Object {
3           let io : IO <- new IO, x : Int <- 1 in {
4           {io.out_int (x);
5            let x : Int < - 5 in {{
6               (* x <- YOUR CODE HERE ;*)
7               io.out_int (x);
8             }};
9             if x == 0 then io.out_string ("x") else io.out_int (x)
                  fi;
10          }}
11        };
12      };
```

(b) Can you replace (* x < − YOUR CODE HERE ;*) with at most one line of code containing an assignment to x that gets this code to print "10x"? Why or why not?

Answer:
It is impossible to get this code to print "10x" by replacing (* x < − YOUR CODE HERE ;*) with at most one line of code containing an assignment to x. Any code at line 6 cannot modify the value of x on line 9. The value of x on line 9 can never be 0. Thus, the string "x", which is necessary for printing out "10x", will never be printed.

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 $O[Int/y] \vdash y + y : Int$:

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

Consider the following Cool program fragment:

```
1       class A {
2           i: Int;
3           j: Int;
4           b: Bool;
5           s: String;
6           o: SELF_TYPE;
7           foo(): SELF_TYPE { o };
8           bar(): Int { 2 * i + j - i / j - 3 * j };
9       };
10      class B inherits A {
11          p: SELF_TYPE;
12          baz(a: Int, b:Int): Bool { a = b };
13          test(c: Object): Object { (* [Placeholder C] *) };
14      };
```

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

$O = \emptyset[\text{Int}/i][\text{Int}/j][\text{Bool}/b][\text{String}/s][\text{SELF\_TYPE}_B/o][\text{SELF\_TYPE}_B/p][\text{Object}/c][\text{SELF\_TYPE}_B/self]$

$M = \emptyset[(\text{SELF\_TYPE})/(A, foo)][(\text{Int})/(A, bar)][(\text{Int,Int,Bool})/(B, baz)][(\text{Object})/(B, test)]$

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)

```
1              b <- p.baz(p.bar(),i)
```

Answer:

$$\cfrac{\cfrac{O(p) = \text{SELF\_TYPE}_B}{O, M, B \vdash p : \text{SELF\_TYPE}_B} \qquad M(B, baz) = (\text{Int,Int,Bool}) \qquad \cfrac{\cfrac{O(p) = \text{SELF\_TYPE}_B}{O, M, B \vdash p : \text{SELF\_TYPE}_B} \quad M(B, bar) = (\text{Int})}{O, M, B \vdash p.bar() : \text{Int}} \qquad \cfrac{O(i) = \text{Int}}{O, M, B \vdash i : \text{Int}}}{O, M, B \vdash p.baz(p.bar(), i) : \text{Bool}}$$

And:

$$\cfrac{O, M, B \vdash p.baz(p.bar(), i) : \text{Bool} \qquad O(b) = \text{Bool}}{O, M, B \vdash b \leftarrow p.baz(p.bar(), i) : \text{Bool}}$$

Thus, type is **Bool**.

Alternative: Note that the last rule directly takes $O(b) = $ Bool as a premise. This is consistent with the Cool Manual type rule for ASSIGN. However, the following alternative derivation, which uses a slightly different, but natural, version of the rule, is also acceptable:

$$\cfrac{O, M, B \vdash p.baz(p.bar(), i) : \text{Bool} \qquad \cfrac{O(b) = \text{Bool}}{O, M, B \vdash b : \text{Bool}}}{O, M, B \vdash b \leftarrow p.baz(p.bar(), i) : \text{Bool}}$$

(b)

```
1              p <- o.foo()
```

Answer:

$$\cfrac{O(p) = \text{SELF\_TYPE}_B \qquad \cfrac{M(B, foo) = (\text{SELF\_TYPE}) \qquad \cfrac{O(o) = \text{SELF\_TYPE}_B}{O, M, B \vdash o : \text{SELF\_TYPE}_B}}{O, M, B \vdash o.foo() : \text{SELF\_TYPE}_B}}{O, M, B \vdash p \leftarrow o.foo() : \text{SELF\_TYPE}_B}$$

Type is SELF_TYPE**B**.

(c)

```
1              b <- baz(i+j,p.bar(i,o.foo()))
```

Answer: This expression will not type check, since $p.bar(i, o.foo())$ presents a type error. This is due to the fact that bar takes zero arguments and the expression passes two. The type mismatch is $(Int, Int, Bool) \neq (Int)$.

(d)

```
1        case c of
2            s: Int => s;
3            i: String => j;
4            b: Object => i;
5        esac
```

Answer:

$$\frac{O(p) = \text{Object}}{O, M, B \vdash c : \text{Object}} \quad \frac{O[s/\text{Int}](s) = \text{Int}}{O[s/\text{Int}], M, B \vdash s : \text{Int}} \quad \frac{O[i/\text{String}](j) = \text{Int}}{O[i/\text{String}], M, B \vdash j : \text{Int}} \quad \frac{O[b/\text{Object}](i) = \text{Int}}{O[b/\text{Object}], M, B \vdash i : \text{Int}}$$

$$O, M, B \vdash \text{case c of s: Int => s; i: String => j; b: Object => i; esac : Int}$$

Type is **Int**.

3. Consider the following Cool program:

```
1      class A {
2          i: Int <- 1;
3          foo(): Int {i};
4      };
5      class B inherits A {
6          baz(): Int {{i <- 2 + i; i;}};
7      };
8      class C inherits B {
9          baz(): Int {{i <- 3 + i; i;}};
10     };
11     class Main {
12         main(): Object {
13             let a : A <- new C, io: IO <- new IO, j : Int in {{
14                 j <- a@B.baz() + a.baz() + a.foo();
15                 io.out_int (j); -- print statement
16             }}
17         };
18     };
```

(a) This code does not compile. Provide a complete but succinct explanation as to why that is the case. Please note that the error message of coolc does not count as an explanation, your answer must show that you understand the problem.

Answer:
On line 14, a is of static type A. The static dispatch expression "a@B.baz()" is not allowed since class B is not a parent of class A. Moreover, the dispatch expression "a.baz()" is not allowed since class A does not a method named baz.

(b) Assume you are given a variant of Cool which is dynamically typed instead of statically typed. Would the behavior of the code above be safe, in the sense of not triggering a runtime type error, in such variant of Cool? Why or why not? If it is safe, what would the print statement output?

Answer:

If Cool is dynamically typed, the behavior of the code above would be safe. a is of dynamic type C. The the expression "a@B.baz()" is allowed since class B is a parent of class C. The expression "a.baz()" is also allowed since class C has a method named baz. The print statement output is 15. a@B.baz() returns 3, a.baz() returns 6, and a.foo() returns 6.

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

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

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[] $\leftarrow$ 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] $\leftarrow$ 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:

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

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:

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

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:

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

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 refereed 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.

$$\text{expr} ::= \text{new TYPE[ ]} < -\{[\text{expr}[,\text{expr}]^*]\}$$
$$\mid \text{expr}[\ \text{expr}\ ] \tag{2}$$

Typing rules:

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

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

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=[]
```

15

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`'.