

CS143 Compilers - Written Assignment 3

Due Tuesday, May 22, 2018 at 11:59 PM

This assignment covers semantic analysis, including scoping and type systems. You may discuss this assignment with other students and work on the problems together. However, your write-up should be your own individual work, and you should indicate in your submission who you worked with, if applicable. Assignments can be submitted electronically through Gradescope as a PDF by 11:59 PM PDT. A \LaTeX template for writing your solutions is available on the course website.

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     };
```

- (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     };
```

- (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     };

```

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

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

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      };
10     class B inherits A {
11         p: SELF_TYPE;
12         test(): Object { (* [Placeholder C] *) };
13     };

```

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

$$O = \emptyset[Int/i][Int/j][Int/k][Bool/yes][SELF_TYPE_B/p][SELF_TYPE_B/self]$$

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

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. $\text{Bool} \leq \text{Bool}$).

(a) Substitution 1:

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

(b) Substitution 2:

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

(c) Substitution 3:

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

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     };
```

- (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     };

```

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

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

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 an expression like “`my_array[i] ← value`”.

- (a) Provide new typing rules for Cool which handle the typing judgments for: $O, M, C \vdash \text{new } T[e_1]$, $O, M, C \vdash e_1[e_2]$ and $O, M, C \vdash e_1[e_2] <- e_3$. Make sure your rules work with subtyping.
- (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.

- (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.

- (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.

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.

- (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.