

## Solutions to Written Assignment 5

1. Show the full type derivation (as done in slide 49 in the lecture notes) for the following judgement:

$$O[\text{Bool}/x] \vdash x \leftarrow (\text{let } x:\text{Object} \leftarrow x \text{ in } x = x)$$

*Solution:* Let  $\text{Bo}$  stand for the type `Bool` and  $\text{Ob}$  stand for the type `Object`.

$$\frac{\frac{O[\text{Bo}/x](x) = \text{Bo}}{O[\text{Bo}/x] \vdash x: \text{Bo}} \quad \text{Bo} \leq \text{Ob}}{O[\text{Bo}/x] \vdash x: \text{Ob}} \quad \text{Ob} \leq \text{Ob} \quad \frac{\frac{O[\text{Bo}/x][\text{Ob}/x](x) = \text{Ob}}{O[\text{Bo}/x][\text{Ob}/x] \vdash x: \text{Ob}} \quad \frac{O[\text{Bo}/x][\text{Ob}/x](x) = \text{Ob}}{O[\text{Bo}/x][\text{Ob}/x] \vdash x: \text{Ob}}}{O[\text{Bo}/x][\text{Ob}/x] \vdash x = x: \text{Bo}}}{O[\text{Bo}/x] \vdash \text{let } x:\text{Ob} \leftarrow x \text{ in } x = x: \text{Bo}}$$

$$\frac{O[\text{Bo}/x] \vdash x \leftarrow (\text{let } x:\text{Ob} \leftarrow x \text{ in } x = x): \text{Bo}}{O[\text{Bo}/x] \vdash x \leftarrow (\text{let } x:\text{Object} \leftarrow x \text{ in } x = x): \text{Bo}}$$

2. Suppose we extend the grammar for cool with a ‘‘void’’ keyword

$$\begin{array}{l} \text{expr} ::= \mathbf{void} \\ \quad | \dots \end{array}$$

that is analogous to null in Java. (Currently objects are initialized to void if they have no other initializer specified, but there is no general-purpose `void` keyword.) We want to be able to use `void` wherever an object can be used, as in

```
let foo:Int <- if some_test
  then 5
  else void
fi
in ...
```

Give a sound typing rule that we can add to the Cool specification to accommodate this new keyword.

*Solution:*

$$\overline{O \vdash \text{void}: T}$$

You could also do this by defining a new type ‘‘Void’’:

$$\overline{O \vdash \text{void}: \text{Void}}$$

and declaring that  $\text{Void} \leq T$  for all  $T$ . Note that your subtype graph is now a DAG, not a tree.

3. Suppose we extend Cool with exceptions by adding two new constructs to the cool language.

$$\begin{array}{l} \text{expr} ::= \mathbf{try} \text{ expr } \mathbf{catch} \text{ ID } \Rightarrow \text{ expr} \\ \quad | \mathbf{throw} \text{ expr} \\ \quad | \dots \end{array}$$

Here `try`, `catch` and `throw` are three new terminals. ‘‘`throwexpr`’’ returns `expr` to the closest dynamically enclosing `catch` block. Note that since `throw` expression returns control to a different location, we do not really care about the context in which `throw` is used. For example, `(throw false) + 2` is a valid Cool expression (However, note that `(throw false) + (2 + true)` is not a valid Cool expression). Following is an example that uses the `try-catch` and `throw` constructs. It executes ‘‘`do_something1`’’ (with `x` bound to the value 34) if ‘‘`some_test1`’’ evaluates to `true`.

```
try
  if some_test1 then throw 34
  else if some_test2 then throw ‘‘undefined error’’
  else do_something fi fi
catch x =>
  case x of
    x:Int => do_something1
    x:String => do_something2
  esac
```

It executes ‘‘`do_something2`’’ (with `x` bound to the value ‘‘undefined error’’) if ‘‘`some_test1`’’ evaluates to `false` but ‘‘`some_test2`’’ evaluates to `true`. It executes ‘‘`do_something`’’ if both ‘‘`some_test1`’’ and ‘‘`some_test2`’’ evaluate to `false`.

Give a set of new sound typing rules that we can add to the Cool specification to accomodate these two new constructs.

**Solution:**

$$\frac{O \vdash e: T_1}{O \vdash \text{throw } e: T_2}$$

$$\frac{O \vdash e_1: T_1 \quad O[\text{Object}/x] \vdash e_2: T_2}{O \vdash \text{try } e_1 \text{ catch } x => e_2: T_1 \sqcup T_2}$$

4. The Java programming language includes arrays. The Java language specification states that if `s` is an array of elements of class `S`, and `t` is an array of elements of class `T`, then the assignment `s = t` is allowed as long as `T` is a subclass of `S`. This typing rule for array assignments turns out to be unsound. (Java works around the fact that this rule is not statically sound by inserting runtime checks to generate an exception if arrays are used unsafely. For this question, assume there are no special runtime checks.)

Consider the following Java program, which type checks according to the preceding rule:

```
class Mammal { String name; }

class Dog extends Mammal { void beginBarking() { ... } }

class Main {
  static public void main(String argv[]) {
    Dog x[] = new Dog[5];
    Mammal y[] = x;

    /* Insert code here */
  }
}
```

```
}  
}
```

Add code to the main method so that the resulting program is a valid Java program (i.e., it type checks statically and so it will compile), but the program could result in an operation being applied to an inappropriate type when executed. Include a brief explanation of how your program exhibits the problem.

*Solution:*

```
Dog x[] = new Dog[5];  
Mammal y[] = x;  
  
Mammal a_cat = new Mammal();  
y[0] = a_cat; // ###  
x[0].beginBarking();
```

The problem here is that arrays are not just lists of values -- they represent memory locations into which we can store new data.

Normally we say that  $A \leq B$  if objects of type A can safely be used anywhere that objects of type B can be used. That's not quite true with  $\text{Dog}[] \leq \text{Mammal}[]$ , since a  $\text{Mammal}[]$  object can be safely used on the left-hand side of the assignment marked by `###`, while a  $\text{Dog}[]$  object cannot. Java handles this by adding a runtime check to every array assignment that determines whether the right-hand side of the assignment matches the dynamic type of the array.