CSE 401 - Semantics & Type Checking Worksheet - Week 7 - Solution

All problems in this worksheet will use the following global scope:

```java
class Bar { boolean field; public int method(int i, int j); }
class Foo extends Bar { int val; public boolean whoop(int x); }
```

1. Consider the following hypothetical method definition for `Bar.method`:

```java
public int method(int i, int j) {
    int r;
    boolean b;
    Foo o;
    if (this.field) {
        o = this;
        b = o.whoop(i + j);
        r = o.val;
    } else {
        r = i * j + 3;
    }
    return r;
}
```

   a. What is the local scope in the method body?

   ```java
   Bar this; int i; int j; int r; boolean b; Foo o;
   ```

   Remember that every MiniJava method has an implicit parameter “this” for the receiver object. For the sake of type-checking the method body, it makes sense to treat it like a normal parameter, though your real code need not represent it as such in symbol tables.

   b. The method body is ill-behaved. Can you prove this by describing a possible execution trace of the method that would “go wrong”? (It suffices to provide possible runtime values for variables in the local scope the parameters.)

   ```java
   this = Bar(field: true); i = *; j = *;
   ```

   * is a stand-in for any integer value.

   The ill-behavior is the potential failure of the downcast in the assignment “o = this.” Unlike real Java, MiniJava’s dynamic semantics defines no behavior for a failing downcast, so the static semantics forbids downcasts altogether.
c. The method body is also ill-typed. Can you describe which type check(s) deduce this fact?

Since MiniJava’s static semantics forbids downcasts, a MiniJava compiler must check that the type of an assignment statement’s right-hand side is either the same as the left-hand side’s type or a subclass type of the left-hand side’s class type.

One of the suggested project extensions is actually to add support for downcasts safely checked at run-time, defining the behavior of failed downcasts as “terminate with error.”

d. Is every possible execution trace of that method ill-behaved? Can you describe one that happens to be perfectly well-behaved? (Again, possible runtime values for variables in the local scope the parameters suffice.)

No, some possible executions of the method avoid the ill-behaving branch, for example given the following parameter values:

```
this = Bar(field: false); i = *; j = *;
```

Alternatively, some possible executions could enable the “downcast” to succeed, if the receiver object (this) ends up really being an instance of the subclass Foo, like so:

```
this = Foo(field: true, val: *); i = *; j = *;
```

* is a stand-in for any integer value.

e. Suppose that we replaced the use of this.field in the method body to call a boolean method that always returns false. How would this change your answers to the previous questions?

Even though the ill-behaving branch would never get run, type checking composes through types and type signatures (not the specific values!), so a type checker for MiniJava will not verify the method body (i.e., will report a type error), despite the forbidden behavior being impossible according to the dynamic semantics.