# CS321 Homework 5 — Sample Solution

November 2012

**Question 1**: The purpose of this question is to generate one or more Mini test files that will cause the Mini checker to report an error. The question includes one example to show what happens when a logical not operator, `!` is given an integer argument instead of a Boolean argument as it expects; our task in this assignment is to write code that will trigger a further 10 distinct error messages.

At this point, it would be possible just to start writing simple Mini files, perhaps using the example supplied as a starting point, and trying to introduce errors until we have collected enough samples to show 10 distinct errors. With even a little programming experience, it is probably not too difficult to do this. But it is a bit of a hit and miss technique; we might be lucky enough to stumble on 10 different types of errors just by experimenting, but what if we run out of ideas? Is there a more methodical approach that we can try?

One option is to notice that the Mini checker creates a `new Failure` object every time it detects an error. Searching for this string in all the Java files in the `mini` quickly produces a candidate list of different error conditions (some lines omitted):

```
~/HW5/MiniStatic/mini $ grep "new Failure" *.java
ArrayAccess.java:        throw new Failure("Static analysis for ArrayAccess not implemented");
ArrayAssign.java:        throw new Failure("Static analysis for ArrayAssign not implemented");
Assign.java:           ctxt.report(new Failure("Types in assignment do not match"));
BNot.java:           ctxt.report(new Failure("Bitwise not expects boolean or int argument"));
. . .
. . .
. . .
IndentTest.java:                handler.report(new Failure("Exception: " + e));
InitVarIntro.java:                ctxt.report(new Failure("initializer has wrong type"));
LNot.java:           ctxt.report(new Failure("Logical not expects boolean argument"));
NewArray.java:           throw new Failure("Array size must be an integer");
Print.java:              ctxt.report(new Failure("print requires integer argument"));
UnArithExpr.java:           throw new Failure("Unary operation expects numeric argument");
~/HW5/MiniStatic/mini $
```

The first two items listed here are for `ArrayAccess` and `ArrayAssign`; we'll get to them in the next question. The list also includes some diagnostics produced by test programs, such as `IndentTest` in the list above; these are not related to static analysis or type checking, so we will ignore them here. For each of the remaining errors, I can consult the appropriate source code lines (if the error text is not already clear enough) to determine the circumstances under which the diagnostic should be triggered, and then write a small Mini program to demonstrate that error. In my case, I was able to create a single (nonsensical) Mini program that triggers 10 different error conditions; the following listing shows each line annotated with the corresponding error:

```
int mini_main() {
  int x;
  double y = x;// initializer has wrong type
  x = y;       // Types in assignment do not match
  ~y;          // Bitwise not expects boolean or int argument
```

```
    x && x;      // Boolean operands required
    if (x)       // Boolean test expected in if statement
      while (x)  // Boolean test expected in while loop
        print y; // print requires integer argument
    if (c)       // The variable c is not in scope
      if (x==y)  // Operands being compared have different types
        x + y;   // Arithmetic operands have different types
}
```

[As an aside: I chose the first seven errors in this list because they are all `reported` as errors without `throwing` an exception; this means that they are in some sense less disruptive, making it easier to combine multiple errors in a single file. For the final three, I had to resort to errors that are `thrown` as exceptions. For this reason, I wrapped the errors inside the test of an `if` statement because I remembered that the code shown in class can recover from an exception that is raised in that situation. This enabled me to squeeze 10 distinct error conditions in to a single file, even though 10 distinct error conditions in 10 distinct files would have been good enough for this assignment.]

For the purposes of testing, I can build and run the Mini checker to produce the following output:

```
~/HW5/MiniStatic $ java mini.Check errs.mini
ERROR: initializer has wrong type

ERROR: Types in assignment do not match

ERROR: Bitwise not expects boolean or int argument

ERROR: Boolean operands required

ERROR: Boolean test expected in if statement

ERROR: Boolean test expected in while loop

ERROR: print requires integer argument

ERROR: The variable c is not in scope

ERROR: Operands being compared have different types

ERROR: Arithmetic operands have different types

Total failures found: 10
~/HW5/MiniStatic $
```

By a quick visual inspection, we can confirm that the error messages reported here match the comments in the source file, and that all of these are distinct (from one another, and from the `LNot` case in the question).

**Question 2**: This question requires us to complete the implementation of the `typeOf()` methods in the `ArrayAccess` and `ArrayAssign` classes. By looking at the source code, we can see that an `ArrayAccess` object contains two fields:

```
class ArrayAccess extends Expr {
    private Expr arr;
    private Expr idx;
    ...
}
```

For the purposes of type checking, we need to ensure that:

- The index, `idx`, will evaluate to an integer. This requires us to test that `idx.typeOf(ctxt, env)` is not equal to the `int` type, which is represented by `Type.INT`.
- The array expression, `arr`, will evaluate to an array value, meaning that it must have a type of the form `T[]` (i.e., a type that is represented by the `ArrayType` class). If we compute the type of `arr` as `tarr = arr.typeOf(ctxt, env)`, then we can use the `elementType()` method to make sure that `tarr` is an array type: the result of `tarr.elementType()` will either be `null` if `arr` is not an array, or else it will produce the representation, `elem`, for the type of elements in the array. In this case, of course, we can also infer that the result of the whole expression, `arr[idx]` must have type `elem`.

These ideas can be captured by the following method, which I'll add to the `Expr` class (because that is a common ancestor of both `ArrayAccess` and `ArrayAssign`):

```
protected Type arrayAccessType(Context ctxt, VarEnv env, Expr arr, Expr idx)
  throws Failure {
    // Check that the index is an integer:
    if (!idx.typeOf(ctxt, env).equal(Type.INT)) {
        ctxt.report(new Failure("Array index is not an int"));
    }

    // Check that the arr argument has an array type
    Type tarr = arr.typeOf(ctxt, env);
    Type elem = tarr.elementType();
    if (elem==null) {
        throw new Failure("Expected expression of array type");
    }
    return elem;
}
```

The reason for defining a method like this instead of placing the code directly in to the `ArrayAccess` class is that it will allow us to reuse this same code for type checking an `ArrayAssign` expression too. (With hindsight, perhaps we should even have defined `ArrayAssign` as a subclass of `ArrayAccess`, but I won't dwell on that further here). The case for `ArrayAccess` is just a direct call to `arrayAccessType()`:

```
public Type typeOf(Context ctxt, VarEnv env)
  throws Failure {
    return arrayAccessType(ctxt, env, arr, idx);
}
```

For `ArrayAssign`, we use the same function to check the array portion of the expression, but we need to add an additional test to ensure that the right hand side expression, `rhs`, has the appropriate type:

```
public Type typeOf(Context ctxt, VarEnv env)
  throws Failure {
    // Find the type of the array portion:
    Type elem = arrayAccessType(ctxt, env, arr, idx);

    // Check that the rhs has type elem
    if (!rhs.typeOf(ctxt, env).equal(elem)) {
        ctxt.report(new Failure("Right hand side of array assignment has wrong type"));
    }
    return elem;
}
```

Of course, we should perform at least some simple tests to ensure that our implementations are working correctly. The following Mini code shows one simple example, annotating each of the lines to indicate the places where we are declaring an array value, allocating an array value, accessing an array value (the `ArrayAccess` case), or assigning to an array value (the `ArrayAssign` case). Note also that I've used two types of array, one (represented by the variable `a`) containing `int` values, and one (represented by the variable `b`) containing `int[]` values:

```
void mini_main() {
  int[]   a = new int[10];      // Declare (array of int), Allocate
  int[][] b = new int[][2];     // Declare (array of array of int), Allocate

  a[0] = a[1] + a[2];           // Assign (using int values), Access x 2
  b[0] = a;                     // Assign (using int[] values)
  b[1] = new int[12];           // Assign (using int[] values), Allocate

  print a[0];                   // Access
}
```

We expect this program to be accepted by the static checker without any errors. (And, in particular, without showing either of the original diagnostics to suggest that type checking for `ArrayAccess` and `ArrayAssign` had not been implemented!) The following output confirms this behavior:

```
~/HW5/MiniStatic $ java mini.Check array.mini
Program appears to be valid!
Total failures found: 0
~/HW5/MiniStatic $
```

It is also a good idea to write some tests that are designed specifically to trigger the error conditions that our implementations of `typeOf()` are supposed to detect. There are five such conditions (two for `ArrayAccess` and three for `ArrayAssign`) as shown in the following code (I've used the same trick from Question 1, placing the third and fourth errors inside an `if` so that they do not force the type checking process to abort):

```
void mini_main() {
   int[] a;
   int x;

   a[1==1];         // Access: index is not an int!
   a[0==0] = 42;    // Assign: index is not an int!

   if (x[0])        // Access: expression does not have array type
     if (x[0] = 1)  // Assign: expression does not have array type
       a[0] = a;    // Assign: rhs type differs from array elem type
}
```

The following output shows that our implementation produces the five expected error conditions:

```
~/HW5/MiniStatic $ java mini.Check badarray.mini
ERROR: Array index is not an int

ERROR: Array index is not an int

ERROR: Expected expression of array type

ERROR: Expected expression of array type
```

```
ERROR: Right hand side of array assignment has wrong type

Total failures found: 5
~/HW5/MiniStatic $
```

**Question 3**: The remaining task is to extend the Mini checker to support static analysis of C/Java-stye for loops, building on the implementation of this construct in the previous homework. Different people might use slightly different approaches here depending on the specifics of their solution to Homework 4 although the high-level details should be the same in each case. The details that I present here are based on the approach that I described in my solution to Homework 4 in which the initializer part of a `for` loop is represented by an object (of type `NoInit`, `ExprInit`, or `VarDeclInit`, all of which are subclasses of the class `ForInit`), while the `test` and `step` portions of the `for` loop are represented either by an expression, or else by a null pointer if that component was not included in the source program.

Now we can discuss the task of checking that a statement of the form `for (init; test; step) body` is valid, breaking it down in to steps and assuming that the environment at the start of the loop is `Env`.

1. The initializer, `init`, can be either empty (`NoInit`), an expression (`ExprInit`), or a variable declaration (`VarDeclInit`). In the latter case, the initializer could introduce some new variables that will be in scope inside the loop, which will be reflected by using an extended version of `Env` that I'll refer to by the name `IEnv` (for "inner environment", although the name isn't actually significant). We can represent the restrictions on the initializer by the following judgement: $\texttt{Env} \vdash \texttt{init} : \texttt{IEnv}$.
2. If present, the `test` expression must be a Boolean value, and because it can make use of new variables that are introduced by the initializer, it should be type checked using the `IEnv` environment. Thus we can represent the restrictions on the test using the judgement: $\texttt{IEnv} \vdash \texttt{test} : \texttt{bool}$.
3. If present, the `step` expression must be well-typed using the `IEnv` environment, but we do not actually care what type of result it will produces. We can capture this using a judgement $\texttt{IEnv} \vdash \texttt{step} : t$ where the variable $t$ represents an arbitrary type.
4. The `body` is a statement that can access any variables introduced by the initializer (so it starts with the `IEnv` environment). It is also possible that the `body` will introduce some new variables, resulting in a final environment $\texttt{Env}'$ although those variables will go out of scope at the end of the body. As such, we can capture the restrictions on the `body` using the judgement $\texttt{IEnv} \vdash \texttt{body} : \texttt{Env}'$.

Putting all four of these judgements together as hypotheses, we can construct the following inference rule:

$$\frac{\texttt{Env} \vdash \texttt{init} : \texttt{IEnv} \quad \texttt{IEnv} \vdash \texttt{test} : \texttt{bool} \quad \texttt{IEnv} \vdash \texttt{step} : t \quad \texttt{IEnv} \vdash \texttt{body} : \texttt{Env}'}{\texttt{Env} \vdash \texttt{for(init;test;step) body} : \texttt{Env}}$$

Note that the output environment in the conclusion is just `Env`, which is the same as the input environment. This reflects the fact that any variables introduced either in the initializer or in the body of the loop will no longer be in scope once the loop terminates.

These ideas translate directly in to the following implementation of the `check()` method in the `For` class, with the correspondence between sections of code and the descriptions above being captured in the comments:[1]

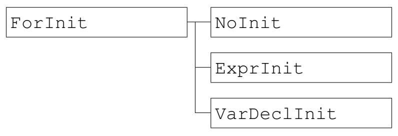
```
class For extends Stmt { ...
    public VarEnv check(Fundef def, Context ctxt, VarEnv env)
```

[1] I'll write an ellipsis (`...`) in each of the following class definitions shown here to indicate that some parts of the full class definition have been omitted from the presentation here.

```
        throws Failure {

          // 1. Check the initializer:
          VarEnv ienv = init.check(def, ctxt, env);

          // 2. Check the test:
          try {
              if (test!=null && !test.typeOf(ctxt, ienv).equal(Type.BOOLEAN)) {
                 ctxt.report(new Failure("Boolean test expected in while loop"));
              }
          } catch (Failure f) {
              // report any error that occured while checking the expression.
              ctxt.report(f);
          }

          // 3. Check the step:
          try {
              if (step!=null) {
                 step.typeOf(ctxt, ienv);
              }
          } catch (Failure f) {
              // report any error that occured while checking the expression.
              ctxt.report(f);
          }

          // 4. Check body, but discard any modified environment that it produces.
          body.check(def, ctxt, ienv);

          // We can't guarantee that a for loop will execute the body
          def.returns = false;

          // The final environment is the same!
          return env;
      }
  }
```

Note that the code for Steps 2 and 3 checks for the possibility that the `test` and `step` expressions, respectively, might be `null`, in which case, it does not attempt to perform static checking on that component of the loop.

The code above is a fairly direct translation of the English language description and the (much more concise) inference rule. One part that needs more explanation, however, is the treatment of the initializer in Step 1. This code uses a version of the `check()` method to compute the inner environment, `ienv`, but in this situation we are calling `check()` on a value of type `ForInit` and not on a value of type `Stmt` as we have seen previously. This indicates that our implementation of static analysis must also include an implementation of `check()` for the `ForInit` class and each of its subclasses shown in the following diagram:



The `ForInit` class is an abstract base class, so we can just define `check()` as an abstract method:

```
  public abstract class ForInit { ...
      public abstract VarEnv check(Fundef def, Context ctxt, VarEnv env)
```

```
        throws Failure;
  }
```

Given that definition, we must now provide implementations for each of the three subclasses. An empty initializer, for example, is represented by the `NoInit` class, which does not create any new environment entries and instead just returns the original environment, `env`, unchanged:

```
  public class NoInit extends ForInit { ...
      public VarEnv check(Fundef def, Context ctxt, VarEnv env)
        throws Failure {
          return env;
      }
  }
```

The `ExprInit` class is used to represent expression initializers. In this case, we need to make sure that the expression has some type, but we do not care what type that is, and the resulting environment is just the same as the original:

```
  public class ExprInit extends ForInit { ...
      public VarEnv check(Fundef def, Context ctxt, VarEnv env)
        throws Failure {
          expr.typeOf(ctxt, env);   // check expr is well-typed, discard result type
          return env;
      }
  }
```

Finally, `VarDeclInit` represents variable declaration initializers and holds a single `VarDecl`. In this case, the variable declaration will introduce one or more new variables to the environment, but it is easy to calculate the environment that will result on this by using a recursive call of the `check()` method on the variable declaration:

```
  public class VarDeclInit extends ForInit { ...
      public VarEnv check(Fundef def, Context ctxt, VarEnv env)
        throws Failure {
          return decl.check(def, ctxt, env);
      }
  }
```

This completes our implementation of static analysis for `For`, which means that it's time to do some testing! One obvious candidate for a quick sanity check is the simple sample program that we used in the previous assignment to check that parsing was working correctly:

```
  void mini_main() {
      int j;

      for (int i=0; i < 10; i = i+1)
        print i;

      for (; j < 10; )
        ;

      for (; ; )
        j = j+1;
  }
```

Running this code through our expanded checker confirms that, at least for these examples, our new checker appears to be giving the correct results:

```
~/HW5/MiniStatic $ java mini.Check simplefor.mini
Program appears to be valid!
Total failures found: 0
~/HW5/MiniStatic $
```

We can also try to craft some examples to test specific details of our implementation, as shown in the following example.

```
void mini_main() {
    boolean i;

    for (int i=0; i < 10; i = i+1) { // uses local i
      print i;
      double i; // should not be in environment after this loop
    }

    for (i=(1==1); i; i)             // uses boolean i
      if (i)
        print 42;

    for (; 1; j)  // error in test type
                  // error in step expression
      ;
}
```

In the first loop, we use an variable declaration as an initializer; this allows us to confirm that the initializer declaration takes precedence over the outer declaration for `i` as a `boolean` and we would get type errors in the test, step, and body portions if the wrong environment had been used in any of those places. In the second for loop, we verify that the original environment in which `i` is a `boolean` has been restored. Finally, in the third loop, I've included tests in which the test part has the wrong type (to confirm that our code enforces the requirement for a `boolean` test) and I've used a step expression that references an unbound variable (to confirm that our implementation is still checking the step expression, even though its type is not used). As a result of the third loop, we would expect this program to have two static typing errors, which we can verify by running it through our checker:

```
~/HW5/MiniStatic $         java mini.Check morefor.mini
ERROR: Boolean test expected in while loop

ERROR: The variable j is not in scope

Total failures found: 2
~/HW5/MiniStatic $
```

Finally, just to make sure that our checker works in a reasonable way with each of the different combinations that can occur when some components are omitted from a for loop, we can modify the `maketests.java` code from the previous assignment as follows:

```
public class maketests {

  public static void main(String[] args) {
    String[] inits  = new String[] { "int i=0", "42", "" };
```

```
        String[] tests  = new String[] { "j < 10", "" };
        String[] steps  = new String[] { "j = j+1", "" };
        String[] bodies = new String[] { "print j;", ";" };

        int count = 0;
        System.out.println("void mini_main() {");
        System.out.println("    int j;");
        for (int i=0; i<inits.length; i++) {
          for (int t=0; t<tests.length; t++) {
            for (int s=0; s<steps.length; s++) {
              for (int b=0; b<bodies.length; b++) {
                System.out.print("    for (");
                System.out.print(inits[i]);
                System.out.print("; ");
                System.out.print(tests[t]);
                System.out.print("; ");
                System.out.print(steps[s]);
                System.out.println(")");
                System.out.println("      " + bodies[b]);
                System.out.println();
                count++;
              }
            }
          }
        }
        System.out.println("}");
        System.out.println("// " + count + " tests in total");
      }
```

This variant of the original program only produces 24 test cases because we have reduced the number of choices for each component to a minimum, but we have also been careful in the choice of expressions and declarations to ensure that each combination should be valid. (For example, because the declaration `int i=0` will not be included in all of the generated tests, we don't use the variable `i` in the test, step, and body components.)

```
~/HW5/MiniStatic $ javac maketests.java
~/HW5/MiniStatic $ java maketests > fortests.mini
~/HW5/MiniStatic $ java mini.Check fortests.mini
Program appears to be valid!
Total failures found: 0
~/HW5/MiniStatic $
```

With these tests, we can now have reasonably good confidence that our implementation of static analysis for `for` loops is able to handle all of the different combinations of components, missing or present.

Now if only we could actually execute these programs too . . . but for that, we'll have to wait until next term!