Type Checking in MINI Compiler

Jingke Li

Portland State University

Type Checking MINI

Similar to the symbol table module, MINI's type-checker is to be implemented as a visitor over the AST nodes.

- The `TypeVisitor` visits all expression and statement nodes in an AST, including those embedded in declarations (e.g. initialization expressions in `VarDecl` nodes), and verifies that the operations satisfy MINI's typing rules.
- A symbol table is assume to have been built, and is passed in to the visitor as a parameter.

MINI's Typing Rules (Highlights)

- Every name (class name, method name, variable name, etc.) used in a MINI program must be declared first.

```
class typeErr {
  public static void main (String[] a) {
    C x;    // class C is not defined
    f();    // method f is not defined
    y = 1;  // variable y is not defined
  } }
```

- Arithmetic operations must have `int` or `float` operands; logical operations must have `boolean` operands; equality comparisons ("==" and "!=") must have operands with the same (arbitrary) type; other relational operations must have `int` or `float` operands.

```
int[] x = new int[5];
int[] y = new int[6];
x != y      // OK
x == 1      // error
3 + true    // error
3 && true   // error
true > 1    // error
```

MINI's Typing Rules (Highlights) (cont.)

- The object for accessing methods or fields must be a class object; the object for indexing operation must be an array object; and array indices must be of type `int`.

```
int[] a;
int x = a.f();  // a is not a class object
this[3] = 2;    // this is not an array
a[true] = 2;    // index is of wrong type
```

- The lhs of an `assignment` must be an l-value, i.e. it must be an `Id`, a `Member`, or an `ArrayElm` node; the rhs must either be of the same type or a subtype of the lhs.

```
new a[3] = ... // wrong type for lhs
int i = true   // type-mismatch between lhs and rhs
```

- The test of `while` and `if` must be of type `boolean`.

```
if (3) ...     // the test is not boolean
```

MINI's Typing Rules (Highlights) (cont.)

- The number and types of actual arguments to a method call must match those of the method's formal parameters, although an actual argument could be of a subtype of its corresponding formal parameter.

```
class test {
  public int f(int i) { return i; }
  public static void main(String[] a) {
    int x = this.f(5); // OK
    x = this.f();      // wrong number of args
    x = this.f(true);  // wrong type of args
  } }
```

- A return value for a method must match that of the declared type; a method with a non-void return type must contain a return statement.

```
public boolean m() { return; }       // error
public boolean m() { }               // error
public void foo() { ...; return 3; } // error
```

MINI's Typing Rules (Highlights) (cont.)

- The argument of `System.out.println` must be of a basic type (i.e. `int`, or `boolean`) or "`StrVal`" type.

```
System.out.println("Result:"); // OK
System.out.println(this);      // the argument is of wrong type
```

An Issue: How to represent a `StrVal` node's type?

Solution:

We borrow `BasictType(BasicType.Int)` to represent the type of a `StrVal` node. Since `StrVal` nodes only appear in a print statement, this is not going to cause any problem.

TypeVisitor.java

```
public class TypeVisitor implements TypeVI {
  private Table symTable;
  private ClassRec currClass;
  private MethodRec currMethod;
  private boolean hasReturn;

  // constructor -- a symbol table is passed in as a parameter
  public TypeVisitor(Table symtab) { ... }

  // top level visit routine
  public void visit(Program n) throws Exception {
    n.cl.accept(this);
  }

  // LISTS
  public void visit(ClassDeclList n) throws Exception {
    for (int i = 0; i < n.size(); i++)
      n.elementAt(i).accept(this);
  }
```

Type Checking Decls

```
// TYPES
public Type visit(BasicType n) { return n; }
public Type visit(ArrayType n) { return n; }
public Type visit(ObjType n)   {
  // need to verify that the class exists
}

// DECLARATIONS
public void visit(VarDecl n) throws Exception {
  // need to check class existance if type is ObjType
  // need to check init expr if it exists
}
public void visit(ClassDecl n) throws Exception {
  // don't forget to set 'currClass' to the class table
}
public void visit(MethodDecl n) throws Exception {
  // don't forget to set 'currMethod' to the method table
}
...
```

Type Checking Stmts and Exprs

```
// STATEMENTS
public void visit(Assign n) throws Exception {
  Type t1 = n.lhs.accept(this);
  Type t2 = n.rhs.accept(this);
  // check for well-formedness of lhs
  // check for type compatibility of both sides
}
...

// EXPRESSIONS
public Type visit(Binop n) throws Exception {
  // based on op type, perform corresponding checks
}
public Type visit(Id n) throws Exception {
  // lookup the Id's type, and return it
}
public Type visit(This n) {
  // lookup current object's type, return a corresponding ObjType
}
```

The Constant Nodes

A simple solution:

```
public Type visit(IntVal n)   { return new BasicType(BasicType.Int);
public Type visit(FloatVal n) { return new BasicType(BasicType.Float);
public Type visit(BoolVal n)  { return new BasicType(BasicType.Bool);
```

A slightly better solution:

```
// define the basic type nodes only once ...
private BasicType IntType = new BasicType(BasicType.Int);
private BasicType FloatType = new BasicType(BasicType.Float);
private BasicType BoolType = new BasicType(BasicType.Bool);

// use them whenever needed
public Type visit(IntVal n)   { return IntType; }
public Type visit(FloatVal n) { return FloatType; }
public Type visit(BoolVal n)  { return BoolType; }
```

Type Compatibility

A routine that implements MINI's (i.e. Java's) type-equivalence model.

Type `t2` is compatible with type `t1` if `t2` is equivalent to `t1`, or `t2` is a subtype of `t1`.

```
private boolean compatible(Type t1, Type t2) throws Exception {
  // if t1==t2 or both are the same basic type
  //   return true
  // else if both are ObjType   // name equivalence
  //   if (their class ids are the same, or
  //       t1's cid matches an ancestor's cid of t2)
  //     return true
  // else if both are ArrayType // structure equivalence
  //   recursively test the compatibility of their elements' types
  // else
  //   return false
}
```