CS444 Assignment 2,3,4

Kevin Suwala, Kyle Verhoog

Mar 18, 2019

Contents

Assignment 2: Environment Building **1**
Environments 1
Building Environments 2

Assignment 3: Type checking **2**
Type classes 2
Field access and method invocation 3
Type checking 3

Assignment 4: Static analysis **3**
Reachability 3
Constant expression evaluation 4
Definite assignment 5

Testing **5**

Assignment 2: Environment Building

Environments

Location: `src/main/scala/compiler/joos1w/environment`

We broke environment building into 7 different classes of environment:

1) Generic environment: is the parent class for all environments and includes logic for handling classes, methods and variables. It provides lookup methods for each of these.
2) Root environment: is a singleton used to represent the global namespace. This is used for resolving packages.
3) Package environment: is used to represent a package. This stores class and interface related information.

4) Class environment: is used to represent methods and fields for a given class. The class environment also does the `contains`, `nodecl`, `inherit`, `super`, etc classes.
5) Method environment: just holds the method metadata.
6) Variable environment: an environment containing a variable along with metadata about the variable.
7) Block environment: this represents `{ }`.

Building Environments

First pass

For the first pass through the AST we generate package, class and variable environments. These environments are attached to their corresponding AST node so that type checking can be performed later. For example, a variable AST node will have reference to the enclosing environment it is in. In this pass we do checks including the following:

- check duplicate class/method declarations
- duplicate variable declarations

Second pass (verification pass)

In this pass we check that types exist and all qualified names resolve to a class or interface. Here the `contains`, `super`, etc. sets are generated and used. Essentially all checks are done in this pass. We perform, among others, the following checks:

- import priorities
- verify return types for methods exist
- no cycles exist in hierarchies
- no collisions in imports
- class doesn't extend a final class

Assignment 3: Type checking

Location: `src/main/scala/compiler/joos1w/types`

All type-checking is done in a single pass through the AST following the environment building, type-linking and hierarchy building done in A2.

Type classes

So for each Java type we define, in turn, our own classes representing each of these types. We have the following classes

- `AbstractType`: parent of all following types

- `PrimitiveType`: parent class for primitive types like `int`, `boolean`, `char`, etc.
  - `IntType`: represents a Java `int`
  - `BooleanType`: represents a Java `boolean`
- `StringType`: `java.lang.String`
- `ArrayType`: wraps a given `AbstractType` and represents this type as an array
- `CustomType`: represents user-defined classes and interfaces

Special cases like numeric types are defined in `AbstractType` and overridden in child classes like `IntType` to specify that they are numeric.

Field access and method invocation

We just followed the rules as described in class for field and method invocations. The rules as defined by the *Java Language Specification* for these accesses are checked. For example, when a protected field is used it must be checked that the enclosing class is a subclass of the class that declares the field or that the enclosing class is in the same package.

Type checking

Type checking is done in a single pass through the AST. We use the references assigned to the AST nodes linking them with their environment to get type information. A method `determineType` is used to evaluate the type of a given AST node. This method pattern matches and performs, recursively, the type lookup, while also performing verification on the type. So, for example, `determineType(assignmentAST)` will recursively look up the type of the left and right hand sides of the assignment and verify that the types are legal to be assigned. Legal meaning that they obey the type rules for inheritance.

Assignment 4: Static analysis

Location: `src/main/scala/compiler/joos1w/Joos1WReachability`

Reachability

Relevant methods: - `reachableCheck`: recurses down to method bodies to get to statements. - `statementReachableCheck`: evaluates statement-by-statement the reachability

Reachability is checked in a top-down traversal of the AST. The `in` information is propagated downwards through the recursion. The rules are checked for `if`, `if-else`, `while` and `for` statements.

Constant expression evaluation

Since Joos1W does not support finals, constant expression checking is quite straightforward. In fact, here is (most) of the code that performs the evaluation:

```
def evalGeneralExpression(expr: AST): Any = {
  expr match {
    case _: ConditionalExpression | _: MethodInvocation | _: Name => None
    case expr: literals.IntegerLiteral => Some(expr.integerValue)
    case expr: literals.BooleanLiteral => Some(expr.value)
    // ...
    case expr: UnaryExpression =>
      expr.operator match {
        case "!" =>
          evalGeneralExpression(expr.subExpression) match {
            case Some(b: Boolean) => !b
          }
      }
  }
}
def evalConstantExpression(expr: AST): Option[Boolean] = {
  expr match {
    case expr: ConditionalExpression =>
      val l = evalGeneralExpression(expr.firstExpr)
      val r = evalGeneralExpression(expr.secondExpr)
      (l, expr.operator, r) match {
        // == operator
        case (Some(i: Int), "==", Some(j: Int))         => Some(i == j)
        case (Some(i: Boolean), "==", Some(j: Boolean)) => Some(i == j)
        // ...
        // != operator
        case (Some(i: Int), "!=", Some(j: Int))         => Some(i != j)
        case (Some(i: Boolean), "!=", Some(j: Boolean)) => Some(i != j)
        // ...
        case (Some(i: Boolean), "||", Some(j: Boolean)) => Some(i || j)
        case (Some(i: Boolean), "&&", Some(j: Boolean)) => Some(i && j)
        // > operator
        case (Some(i: Int), ">", Some(j: Int))   => Some(i > j)
        case (Some(i: Char), ">", Some(j: Char)) => Some(i > j)
        // < operator
        case (Some(i: Int), "<", Some(j: Int))   => Some(i < j)
        case (Some(i: Char), "<", Some(j: Char)) => Some(i < j)
        case _ => None
      }
    case _: Name | _: MethodInvocation => None
    case _ =>
```

```
        evalGeneralExpression(expr) match {
          case Some(b: Boolean) => Some(b)
          case None             => throw new RuntimeException(s"$expr")
        }
      }
    }
```

Definite assignment

Again, with Joos1W restrictions, definite assignment is rather straightforward. All that needs to be checked is that a variable is initialized when declared, ensuring that the initializer does not include the variable itself. To ensure that an initializer does not include the variable itself, the environment references attached to AST nodes during the environment building phase are used to lookup whether each name that matches the name of the declared variable is previously defined in the environment, else we raise an exception.

We perform this check in the same AST traversal as the reachability checks.

Testing

Location: `src/test/scala/joos1w`

For each phase we attempted to write unit tests but were unable to rigorously test our code due to time limitations and software design choices. The few unit tests we have are in

- `src/test/scala/joos1w/EnvironmentTest.scala`
- `src/test/scala/joos1w/ReachabilityTest.scala`

When pressed for time we resorted to using our own Marmoset test runners located in `src/test/scala/joos1w/MarmosetTestRunner.scala`.