CS 444 - Assignments 2, 3 and 4

dburgoyn
udimitri
x2fang

Section 001
2015-03-19

This document describes the design of our abstract syntax; describes the environment building, type linking, hierarchy checking, name linking, type checking and reachability analysis phases of our compiler; discusses challenges that we encountered during these phases and how we tried to overcome them; and explains the testing that we did before submitting to Marmoset.

1 Abstract syntax design

Our abstract syntax tree is described by the following class hierarchy, rooted at `ASTNode`:

- `Program`: contains a list of `Classfiles`
- `Classfile`: contains a package name[1], a list of imports[1], and a `TypeDecl`.
- `TypeDecl`: defines a class or interface; contains a list of modifiers, as well as lists of `Constructors`, `Fields` and `Methods`.
- `Constructor`: contains a list of modifiers, a name[1], a list of `Formals`, and a `Block`.
- `Decl` (abstract): contains a type name[1] and a name[1].
  - `Field`: contains a list of modifiers and an optional initializer `Expression`.
  - `Method`: contains a list of modifiers, a list of `Formals`, and an optional `Block`.
- `Formal`: contains a type name[1] and a name[1].
- `BlockStatement` (abstract)
  - `Local`: contains a type name[1], a name[1], and an initializer `Expression`.
  - `Statement` (abstract)
    - `Block`: contains a list of `BlockStatements`.
    - `EmptyStatement`
    - `ForStatement`
    - `IfStatement`
    - `ReturnStatement`
    - `WhileStatement`

[1]Represented by an `Identifier`.

    * `Expression` (abstract)
        - `ArrayAccessExpression`
        - `ArrayCreationExpression`
        - `BinaryExpression`: contains a binary operator (including assignment but excluding `instanceof`), and left and right `Expressions`.
        - `CastExpression`: contains a type name[1] and an `Expression`.
        - `ClassInstanceCreationExpression`: contains a type name[1] and a list of argument `Expressions`.
        - `FieldAccessExpression`: contains a primary `Expression` and a field name.
        - `Identifier`: contains a list of component `Strings`. May represent a package name, a type name, an on-demand import, the keyword `this`, a variable reference, or a member access.
        - `InstanceofExpression`: contains an `Expression` and a type name[1].
        - `Literal`: contains a type and a lexeme.
        - `MethodInvocationExpression`: contains either a primary `Expression` and a single-component method name[1], or no primary `Expression` and a possibly multi-component method name[1].
        - `UnaryExpression`: contains a unary operator and an `Expression`.

During the construction of the AST, some collections of parse tree nodes are folded into flat collections in the corresponding AST node. For example, when visiting a parse tree node corresponding to the production `ConstructorDeclaration Modifiers ConstructorDeclarator ConstructorBody`, all the descendants of the `Modifiers` parse tree node will be folded into a `List<Modifier>` in the new `Constructor` AST node.

Most of the work done in assignments 2, 3 and 4 involves making passes through the AST. We do not make use of the Visitor pattern here, but instead add a new method to `ASTNode` that is overridden by its concrete subclasses. The `Compiler`'s `compile()` method (essentially the starting point of the compilation process) invokes the root `Program` node's new method. Each AST in turn invokes the new method on its children, until every AST node has had its method invoked.

Inherited attributes are implemented as parameters to these methods, and synthesized attributes are implemented as fields within `ASTNode` or its subclasses. During some passes through the AST, some nodes may store inherited attributes in a field for use in a later pass.

2 Environment building

The classes `Constructor`, `Decl`, `Formal`, `Local` and `TypeDecl` implement an `EnvironmentDecl` interface. We added to `ASTNode` an environment field of type `Cons<EnvironmentDecl>`, where `Cons` is a generic, immutable, singly-linked list. This enables sharing of environment information between environments of different AST nodes, avoiding the problems described in lecture associated with replicating this information.

We added the methods `buildEnvironment()` and `exportEnvironmentDecls()` to `ASTNode` for this pass. `buildEnvironment()` takes in a node's parent's environment, computes the node's local environment, and recursively builds the environments of the node's children. `exportEnvironmentDecls()` reports any new environment declarations introduced by a node into its parent's environment (e.g. `Locals` add themselves to their parent `Block`'s scope, while other types of `BlockStatement` add nothing).

The following is `Block`'s implementation of `buildEnvironment()`, which demonstrates the use of `exportEnvironmentDecls()` to build the node's environment incrementally:

```
public void buildEnvironment(Cons<EnvironmentDecl> parentEnvironment)
        throws NameConflictException, ImportException {
    this.environment = parentEnvironment;

    for (BlockStatement statement : this.statements) {
        statement.buildEnvironment(this.environment);
        EnvironmentDecl export = statement.exportEnvironmentDecls();
        if (export != null) {
            this.environment = new Cons<EnvironmentDecl>(export, this.environment);
        }
    }
}
```

2.1 Import resolution

Import resolution is performed during this phase, in `Classfile`'s `buildEnvironment()` method. This is possible as early as this phase because the parent environment (inherited from the root `Program` node) is a list of all type declarations in the program. Imports are resolved in the manner dictated by the JLS, with single imports resolved first, followed by the implicit current-package on-demand import, followed by explicit on-demand imports, followed by the implicit `java.lang.*` on-demand import.

3 Type linking

We added the method `linkTypes()` to `ASTNode` for this pass. It introduces an inherited attribute `allTypes`, which is a list of all types declared in the program. This list is eventually passed down to the `resolveType()` methods of those `Identifier` nodes which are known to refer to types grammatically (e.g. the `Identifier` in a `ClassInstanceCreationExpression`).

Since type names can refer non non-reference types, we introduced a `Type` interface with the following implementation hierarchy:

- `Type`
  - `ArrayType`: wraps a non-`ArrayType` child `Type`

- NullType: represents the type of the null literal
- PrimitiveType: represents any primitive type
- TypeDecl: represents any class or interface type in the program

These classes provide the methods canCastTo() and isAssignableTo(), which are useful later on in the type checking pass. We use a null Type to signify the return type of methods that return void.

In resolveType(), we disambiguate between array types, primitive types, unqualified reference types, and qualified reference types. If the Identifier names a qualified reference type, we check that no prefix of the Identifier resolves to a type. For both qualified and unqualified reference types, we check that the whole Identifier resolves to exactly one type.

4 Hierarchy checking

The hierarchy checking phase begins with the construction of a directed graph (stored as an adjacency matrix) from the set of all declared types in the program. For each declared type, we locate the row i for that type and the columns j for each of its direct supertypes (extended classes, and extended or implemented interfaces), and set the matrix elements (i, j) to 1.

In order to detect any cycles in this graph, we then attempt to create a topological ordering of the nodes of this graph. This is implemented with the standard depth-first search algorithm and should succeed iff the graph is acyclic. If successful, this step produces a topologically-sorted list of types (where every declared type appears to the right of all its supertypes).

We then process the declared types in the topological order computed in the previous step. For each declared type, we populate its memberset field, which is an object containing the constructors, fields,and methods declared by the type; the fields and methods inherited from the type's supertypes; and a set of the type's supertypes. Abstract and concrete methods are stored in distinct lists.

Processing the declared types in topological order ensures that a TypeDecl's supertypes, if any, already have valid membersets when the TypeDecl's memberset is being created. This facilitates the propagation of inherited fields and methods. Each addition to the memberset is validated as it occurs.

A validate() method is called on the memberset at the end of this procedure to enforce constraints that cannot be checked until the entire memberset has been populated.

5 Name linking

We added the method linkNames() to ASTNode for this pass. This method introduces the inherited attributes curType, which represents the containing type declaration (if any) of the node being processed; curDecl, which represents the containing class member declaration (if any) of the node being processed; curLocal, which represents the containing local variable declaration (if any) of the node being processed; `staticCtx`, which is true iff the containing class member declaration (if any) is static; and `lValue`, which is true iff the node being processed is an expression on the left-hand side of an assignment.

Most implementations of `linkNames()` simply forward the method call to their children. Two non-trivial implementations occur in `Identifier` and `MethodInvocationExpression`.

`Identifier`'s implementation of `linkNames()` produces an object of type `Interpretation`, which represents the entity that the `Identifier` resolves to. An `Identifier` may resolve to any of 1) a formal parameter, local variable, or field; 2) a non-static field access; 3) a package or a prefix of a package; 4) a declared type; or 5) the keyword `this`.

These cases are disambiguated using the method described in lecture. Interpretations are built inductively: the interpretation of a single-component `Identifier` is computed directly from the inherited attributes, and the interpretation of a multi-component `Identifier` builds on that of its longest proper prefix.

Our grammar allows `MethodInvocationExpression`s to be constructed from either a primary `Expression` and a single-component `Identifier` (method name), or no primary `Expression` and a possibly multi-component `Identifier`. `MethodInvocationExpression`'s implementation of `linkNames()` attempts to cleanly separate the method name from the target expression or type. In the first case, where the `MethodInvocationExpression` contains a non-null primary `Expression`, we simply interpret the `Expression` as the target. The second case is more involved: the method name may have a single component, in which case the target is an implicit `this`, or the method name may have multiple components, in which case the interpretation of the longest proper prefix of the method name is used as the target. This target could either be a `TypeDecl` (in the case of static calls), a `FieldAccessExpression`, a `Local`, a `Formal`, a `Field`, or the keyword `this`.

6 Type checking

We added the method `checkTypes()` to `ASTNode` for this pass. We did not introduce any inherited attributes in this pass, though we did make use of previously-propagated inherited attributes as needed. Most concrete implementations of `checkTypes()` are fairly straightforward in that they first type-check the node's children and then verify that these types are as expected. Three non-trivial implementations occur in `MethodInvocationExpression`, `ClassInstanceCreationExpression` and `FieldAccessExpression`. These implementations must resolve the node to a member of a type declaration, and must check that this access is valid according to visibility and typing rules. Furthermore, to resolve a constructor or method, we must disambiguate between overloaded methods and constructors by matching their signatures with the types of the provided arguments in the invocation.

7 Reachability analysis

We added the method `checkReachability()` to `ASTNode` for this pass. This method introduces an inherited attribute `canLeavePrevious`, which indicates whether the previous statement may terminate normally, and computes a synthesized attribute `canLeave`, which indicates whether the current statement may terminate normally.

Constant folding (but not constant propagation) is performed during this phase as required by section 15.28 of the JLS, and is used to determine whether for- and while-loops may run forever or not run at all based on the truth values of their conditions.

Detection of unreachable code or of non-void methods that may not return causes an exception to be raised.

8 Challenges

One challenge we encountered during these phases of our compiler design involved the special treatment of the classes `java.lang.Object` and `java.lang.String` in Joos 1W. Since the supertype of a class or interface is assumed to be `java.lang.Object` if none is explicitly declared, we must remember the location of this class for use in the hierarchy checking phase. To solve this problem, during the environment building phase, the root `Program` node checks the canonical name of all `TypeDecl`s in the program, and stores the `TypeDecl` in a public static field if its canonical name is "java.lang.Object". We store the `TypeDecl` for `java.lang.String` in an analogous way to use when resolving the types of string literals.

Another challenge arose from decisions we made in the design of our concrete and abstract grammars. We made simplifications in our concrete grammar to make it LR(1), and some of these simplifications persisted in the abstract grammar. These simplifications led, for example, to our `Identifier` class being used for a large number of different purposes. The inconsistency of `MethodInvocationExpressions` is another casualty of our grammar simplifications. Some decisions, such as making `this` an `Identifier`, were made for convenience early on but contributed to the described problems as our compiler became more complex.

9 Testing

We tested our code extensively using the public Marmoset tests for each assignment. We separated the positive and negative Marmoset tests into separate folders. Our compiler contains a class called `RunCompilerTests` whose main entry point runs these tests against our compiler and reports whether each test succeeded or failed. If any tests failed, we would arbitrarily pick a failed test, run it individually to determine the cause of failure, fix it, and then re-run the entire test suite.