Introduction to Compiler Construction

Type Checking: Pre-analysis of *j*-- Programs

Outline

Introduction

Introduction

The semantic analysis of *j--* programs requires two traversals of the AST

Introduction

The semantic analysis of *j--* programs requires two traversals of the AST

The traversals are accomplished by the methods `preAnalyze()` and `analyze()`

Introduction

The semantic analysis of *j--* programs requires two traversals of the AST

The traversals are accomplished by the methods `preAnalyze()` and `analyze()`

The `preAnalyze()` method must traverse down the AST only far enough for

- Declaring imported type names
- Declaring user-defined type names
- Declaring fields
- Declaring methods

Introduction

The semantic analysis of *j--* programs requires two traversals of the AST

The traversals are accomplished by the methods `preAnalyze()` and `analyze()`

The `preAnalyze()` method must traverse down the AST only far enough for

- Declaring imported type names
- Declaring user-defined type names
- Declaring fields
- Declaring methods

Therefore, `preAnalyze()` needs to be defined only in the following AST nodes

- `JCompilationUnit`
- `JClassDeclaration`
- `JMethodDeclaration`
- `JConstructorDeclaration`
- `JFieldDeclaration`

Pre-analysis of a Compilation Unit

Pre-analysis of a Compilation Unit

- Creates a `CompilationUnitContext`

Pre-analysis of a Compilation Unit

- Creates a `CompilationUnitContext`
- Declares the implicitly imported types (`java.lang.String` and `java.lang.Object`)

Pre-analysis of a Compilation Unit

- Creates a `CompilationUnitContext`
- Declares the implicitly imported types (`java.lang.String` and `java.lang.Object`)
- Declares the explicitly imported types

Pre-analysis of a Compilation Unit

- Creates a `CompilationUnitContext`
- Declares the implicitly imported types (`java.lang.String` and `java.lang.Object`)
- Declares the explicitly imported types
- Declares the types defined by class declarations

Pre-analysis of a Compilation Unit

- Creates a `CompilationUnitContext`
- Declares the implicitly imported types (`java.lang.String` and `java.lang.Object`)
- Declares the explicitly imported types
- Declares the types defined by class declarations
- Invokes itself for each of the type declarations in the compilation unit

Pre-analysis of a Compilation Unit

</> JCompilationUnit.java

```java
public void preAnalyze() {
    context = new CompilationUnitContext();

    context.addType(0, Type.OBJECT);
    context.addType(0, Type.STRING);

    for (TypeName imported : imports) {
        try {
            Class<?> classRep = Class.forName(imported.toString());
            context.addType(imported.line(), Type.typeFor(classRep));
        } catch (Exception e) {
            JAST.compilationUnit.reportSemanticError(imported.line(), "unable to find %s", imported.toString());
        }
    }

    CLEmitter.initializeByteClassLoader();
    for (JAST typeDeclaration : typeDeclarations) {
        ((JTypeDecl) typeDeclaration).declareThisType(context);
    }

    CLEmitter.initializeByteClassLoader();
    for (JAST typeDeclaration : typeDeclarations) {
        ((JTypeDecl) typeDeclaration).preAnalyze(context);
    }
}
```

Pre-analysis of a Class Declaration

Pre-analysis of a Class Declaration

Creates a new `ClassContext`, whose `surroundingContext` points to the `CompilationUnitContext`

Pre-analysis of a Class Declaration

- Creates a new `ClassContext`, whose `surroundingContext` points to the `CompilationUnitContext`
- Resolves the class's super type

Pre-analysis of a Class Declaration

- Creates a new `ClassContext`, whose `surroundingContext` points to the `CompilationUnitContext`
- Resolves the class's super type
- Creates a new `CLEmitter` instance, which will eventually be converted to the `Class` object for representing the declared class

Pre-analysis of a Class Declaration

- Creates a new `ClassContext`, whose `surroundingContext` points to the `CompilationUnitContext`
- Resolves the class's super type
- Creates a new `CLEmitter` instance, which will eventually be converted to the `Class` object for representing the declared class
- Adds a class header, defining a name and any modifiers, to this `CLEmitter` instance

Pre-analysis of a Class Declaration

- Creates a new `ClassContext`, whose `surroundingContext` points to the `CompilationUnitContext`
- Resolves the class's super type
- Creates a new `CLEmitter` instance, which will eventually be converted to the `Class` object for representing the declared class
- Adds a class header, defining a name and any modifiers, to this `CLEmitter` instance
- Recursively invokes `preAnalyze()` on each of the class' members

Pre-analysis of a Class Declaration

- Creates a new `ClassContext`, whose `surroundingContext` points to the `CompilationUnitContext`
- Resolves the class's super type
- Creates a new `CLEmitter` instance, which will eventually be converted to the `Class` object for representing the declared class
- Adds a class header, defining a name and any modifiers, to this `CLEmitter` instance
- Recursively invokes `preAnalyze()` on each of the class' members
- Produces a `Class` object, and that replaces the `classRep` for the `Type` of the declared class

Pre-analysis of a Class Declaration

Pre-analysis of a Class Declaration

</> JClassDeclaration.java

```
public void preAnalyze(Context context) {
    this.context = new ClassContext(this, context);

    superType = superType.resolve(this.context);

    thisType.checkAccess(line, superType);
    if (superType.isFinal()) {
        JAST.compilationUnit.reportSemanticError(line, "cannot extend a final type: %s", superType.toString());
    }

    CLEmitter partial = new CLEmitter(false);

    String qualifiedName = JAST.compilationUnit.packageName().isEmpty() ?
            name : JAST.compilationUnit.packageName() + "/" + name;
    partial.addClass(mods, qualifiedName, superType.jvmName(), null, false);

    for (JMember member : classBlock) {
        member.preAnalyze(this.context, partial);
        hasExplicitConstructor = hasExplicitConstructor || member instanceof JConstructorDeclaration;
    }

    if (!hasExplicitConstructor) {
        codegenPartialImplicitConstructor(partial);
    }

    Type id = this.context.lookupType(name);
    if (id != null && !JAST.compilationUnit.errorHasOccurred()) {
        id.setClassRep(partial.toClass());
    }
}
```

Pre-analysis of a Method Declaration

Pre-analysis of a Method Declaration

- Resolves the types of the formal parameters

Pre-analysis of a Method Declaration

- Resolves the types of the formal parameters
- Resolves the return type

Pre-analysis of a Method Declaration

- Resolves the types of the formal parameters
- Resolves the return type
- Computes the method descriptor

Pre-analysis of a Method Declaration

- Resolves the types of the formal parameters
- Resolves the return type
- Computes the method descriptor
- Checks proper use of the `abstract` modifier

Pre-analysis of a Method Declaration

- Resolves the types of the formal parameters
- Resolves the return type
- Computes the method descriptor
- Checks proper use of the `abstract` modifier
- Generates (partial) code for the method

Pre-analysis of a Method Declaration

Pre-analysis of a Method Declaration

</> JMethodDeclaration.java

```java
public void preAnalyze(Context context, CLEmitter partial) {
    descriptor = "(";
    Type[] argTypes = new Type[params.size()];
    for (int i = 0; i < params.size(); i++) {
        JFormalParameter param = params.get(i);
        param.setType(param.type().resolve(context));
        descriptor += param.type().toDescriptor();
        argTypes[i] = param.type();
    }
    returnType = returnType.resolve(context);
    descriptor += ")" + returnType.toDescriptor();
    signature = Type.signatureFor(name, argTypes);

    if (isAbstract && body != null) {
        JAST.compilationUnit.reportSemanticError(line(), "abstract method cannot have a body");
    } else if (body == null && !isAbstract) {
        JAST.compilationUnit.reportSemanticError(line(), "method without body must be abstract");
    } else if (isAbstract && isPrivate) {
        JAST.compilationUnit.reportSemanticError(line(), "private method cannot be abstract");
    } else if (isAbstract && isStatic) {
        JAST.compilationUnit.reportSemanticError(line(), "static method cannot be abstract");
    }

    partialCodegen(context, partial);
}
```

Pre-analysis of a Constructor Declaration

Pre-analysis of a Constructor Declaration

Ensures that the constructor is not declared to be `static` or `abstract`

Pre-analysis of a Constructor Declaration

- Ensures that the constructor is not declared to be `static` or `abstract`
- Checks for the proper use of `super()` and `this()`

Pre-analysis of a Constructor Declaration

Pre-analysis of a Constructor Declaration

</> JConstructorDeclaration.java

```java
public void preAnalyze(Context context, CLEmitter partial) {
    super.preAnalyze(context, partial);
    if (isStatic) {
        JAST.compilationUnit.reportSemanticError(line(), "constructor cannot be static");
    } else if (isAbstract) {
        JAST.compilationUnit.reportSemanticError(line(), "constructor cannot be abstract");
    }
    if (!body.statements().isEmpty() && body.statements().get(0) instanceof JStatementExpression) {
        JStatementExpression first = (JStatementExpression) body.statements().get(0);
        if (first.expr instanceof JSuperConstruction) {
            ((JSuperConstruction) first.expr).markProperUseOfConstructor();
            invokesConstructor = true;
        } else if (first.expr instanceof JThisConstruction) {
            ((JThisConstruction) first.expr).markProperUseOfConstructor();
            invokesConstructor = true;
        }
    }
}
```

Pre-analysis of a Field Declaration

Pre-analysis of a Field Declaration

Enforces the rule that fields may not be declared `abstract`

Pre-analysis of a Field Declaration

- Enforces the rule that fields may not be declared `abstract`
- Resolves the field's declared type

Pre-analysis of a Field Declaration

- Enforces the rule that fields may not be declared `abstract`
- Resolves the field's declared type
- Generates the JVM code for the field declaration, via the `CLEmitter` created for the enclosing class declaration

Pre-analysis of a Field Declaration

Pre-analysis of a Field Declaration

</> JFieldDeclaration.java

```
1  public void preAnalyze(Context context, CLEmitter partial) {
2      if (mods.contains("abstract")) {
3          JAST.compilationUnit.reportSemanticError(line(), "field cannot be declared abstract");
4      }
5
6      for (JVariableDeclarator decl : decls) {
7          decl.setType(decl.type().resolve(context));
8          if (partial.containsFieldName(decl.name())) {
9              JAST.compilationUnit.reportSemanticError(line(), "redefining field " + decl.name());
10         } else {
11             partial.addField(mods, decl.name(), decl.type().toDescriptor(), false);
12             partial.addFieldName(decl.name());
13         }
14     }
15 }
```

Pre-analysis of a Field Declaration

Pre-analysis of a Field Declaration

The symbol table for the `Factorial` program once pre-analysis is complete