

Modular name analysis for Java using JastAdd

Torbjörn Ekman and Görel Hedin

Department of Computer Science, Lund University, Sweden
(torbjorn|gorel)@cs.lth.se

Abstract. Name analysis for Java is challenging with its complex visibility rules involving nested scopes, inheritance, qualified access, and syntactic ambiguities. We show how Java name analysis including ambiguities related to names of variables, fields, and packages, can be implemented in a declarative and modular manner using the JastAdd compiler construction system.

Declarative attributes and context-dependent rewrites enable the implementation to be modularized in the same way as the informal Java language specification. The individual rules in the specification transfer directly to equations in the implementation. Rewrites are used to define new concepts in terms of existing concepts in an iterative manner in the same way as the informal language specification. This enables equations to use both context-free and context-dependent concepts and leads to improved separation of concerns. A full Java 1.4 compiler has been implemented to validate the technique.

1 Introduction

The computations done on abstract syntax trees in compilers and related tools are often highly context sensitive. E.g., there are often symbolic names that have different meanings depending on their context. The purpose of name analysis is to bind each name to a declaration and hence resolve the meaning of that name. Name analysis for the Java programming language is challenging with its complex visibility rules involving nested scopes, inheritance, qualified access, and syntactic ambiguities. The purpose of this paper is to show how ambiguities related to names of variables, types, and packages, can be solved in a declarative and modular manner, using the JastAdd compiler construction system.

Consider the qualified name `A.B.C` and the task of binding each individual simple name to its declaration. The meaning depends on the *syntactic context*, e.g., `C` is expected to be a `TypeName` in the **extends** clause of a class declaration, and an `ExpressionName` when being the right hand side of an assignment. There are also *contextually ambiguous* names where the set of visible declarations are required to resolve the name. For example, `A.B` can be the `PackageName` of the top level class `C`, or `A`, `B`, and `C` can all be nested `TypeNames`. Such ambiguities should be resolved by reclassification to `TypeNames` if there are visible type declarations and otherwise to `PackageNames`. The Java Language Specification [3] defines the specific rules for visible declarations at each point in a program and how to first classify context-free names according to their syntactic context and then refine them by reclassifying contextually ambiguous names.

JastAdd supports declarative attributes and context-dependent rewrites that enable the implementation to be modularized in the same way as the informal language specification. The individual rules in the specification transfer directly to equations in the implementation. The language specification contains a set of basic language concepts captured by a context-free grammar. There are, however, additional concepts that are context-dependent, e.g., `TypeNames`. Rewrites are used to refine the tree to use not only the basic concepts but also the context-dependent ones. We present a transformational technique to gradually define new concepts in terms of existing concepts, in the same way that they are defined in the informal language specification. This allows for decomposition of complex problems into simpler ones, and it also better supports separation of concerns.

We define a tiny subset of Java named *DemoJavaNames* which captures all the characteristic problems in resolving contextually ambiguous names that occur in full Java. A complete name analysis implementation for *DemoJavaNames* is presented and included in this paper. We have implemented a full Java 1.4 compiler based on the same technique to verify that the techniques scale to full languages. The system has been validated against the Jacks test-suite and passes more tests than the production quality compilers `javac` and `jikes` [1]. While not claiming superiority over either compiler we claim that our implementation is complete while being less than half the size of the handwritten `javac` compiler.

The rest of this paper is structured as follows. Section 2 introduces the features of JastAdd that are used in the implementation of *DemoJavaNames*. Section 3 describes the implementation of name lookup, syntactic classification, and reclassification of contextually ambiguous names. Section 4 compares our work to related work and Section 5 concludes the paper.

2 JastAdd Background

The JastAdd compiler construction system combines object-orientation and static aspect-oriented programming with declarative attributes and context-dependent rewrites to allow highly modular specifications. This section gives an introduction to the JastAdd system, needed to understand the source code listings in Section 3. The evaluation algorithm is described in [7,2] and the system is publically available [1].

2.1 Abstract grammar

The abstract grammar models an object-oriented class hierarchy from which classes are generated that are used as node types in the abstract syntax tree (AST). Consider the grammar in Listing 1.1. A class is generated for each production in the grammar, e.g., `Prog`, `CompUnit`, `ClassDecl`, and may inherit another production by adding a colon followed by the super production, e.g., `LocalVariableDecl : Stmt`.

The right hand side of a production is a list of elements. The default name of an element is the same as its type unless it is explicitly named by prefixing the element with a name and a colon, e.g., the `FieldDecl` has an element named `Type` which is of type `Name`. Elements enclosed in angle brackets are values, e.g., `<name:String>` in

FieldDecl, while other elements are tree nodes, e.g., Type:Name and Expr in FieldDecl. The tree node element may be suffixed by a star to specify a list of zero or more elements, e.g., ClassDecl* in CompUnit.

The system generates a constructor and accessor methods for value and tree elements. The accessor method for a value element has the same name as the element, e.g., String name(), while the tree element is prefixed by get, e.g., Name getType(). List elements have an index to select the appropriate node, e.g., getClassDecl(int index), and there is an accessor for the number of elements in the list, e.g., int getNumClassDecl().

2.2 Declarative attributes

Attribute Grammars [10] have proven useful when describing context-sensitive information for programming languages. Their declarativeness makes it easy to modularize grammars freely, and they integrate well with the object-oriented programming paradigm, in particular when augmented with *reference attributes*, allowing an attribute to be a reference to another tree node object [6]. This section gives a very brief introduction to *synthesized* and *inherited* declarative attributes.

A *synthesized* attribute is similar to a virtual method without side-effects which allows for efficient evaluation using caching. Consider the grammar in Listing 1.1 and the task to determine whether a Stmt node declares a local variable named *name* or not. This can be implemented through a synthesized attribute using the following JastAdd syntax:

```
syn boolean Stmt.isLocalVariableDecl(String name);
eq Stmt.isLocalVariableDecl(String name) { return false; }
eq LocalVariableDecl.isLocalVariableDecl(String name) =
    name().equals(name);
```

Notice that the equation for LocalVariableDecl overrides the default equation for its superclass Stmt. Notice also the functional styled short-hand for its right-hand side: it uses an expression rather than a block with a return statement. An additional shorthand is possible (but not shown): combining the attribute declaration and the first equation into a single clause by inserting the equation right-hand side before the semicolon in the declaration.

JastAdd supports inter-type declarations [9] where attributes can be added to an existing class in a modular fashion. The target class for each attribute and equation is specified by qualifying its name with the target class name, e.g Stmt and LocalVariableDecl above. The attribute is then woven into the class hierarchy generated from the abstract grammar.

An *inherited* attribute propagates the context downwards the AST. Consider the task to determine the enclosing Block for a Stmt node. A block can tell all its enclosed Stmts that it is the enclosing Block declaration. This can be implemented through an inherited attribute using the following syntax:

```
inh Block Stmt.enclosingBlock();
eq Block.getStmt().enclosingBlock() = this;
```

Equations for inherited attributes are broadcast to an entire subtree in a similar way as for the *including* construct in the Eli attribute grammar system [13]. This subtree is

explicitly selected using a child accessor (`getStmt()` in this case). The equation should thus be read as: *define the value for the enclosingBlock attribute in the entire subtree whose root is the node returned by getStmt() in a block node. The value should be this, i.e., a reference to the block node defining the equation.*

2.3 Context-dependent rewriting

JastAdd supports declarative context-dependent rewrites to dynamically change the AST. A node of type *S* is automatically rewritten to a node of type *T* when a certain condition is true using the syntax below:

```
rewrite S {
    when(condition())
    to T new T(...);
}
```

The rewrites are context-dependent in that the conditions may depend on synthesized and/or inherited attributes. The rewrites are declarative in that they are performed automatically by a rewrite evaluation engine. In the final tree, no rewrite conditions are true. There may be multiple when-to clauses in which case they are evaluated in lexical order. The evaluation engine is demand-driven and rewrites nodes when they are being visited, interleaved with attribute evaluation. The examples discuss the resulting transformation order for each rewrite as well as interaction with other rewrites and attribute evaluation. The evaluation algorithm is presented in [2].

3 Name analysis for DemoJavaNames

This section presents the implementation of name analysis for a tiny subset of Java that only includes compilation units, packages, nested classes with inheritance, fields, initializers, blocks, local variables, and names. We call this subset *DemoJavaNames* and, while being far from useful as a practical language, it captures all the characteristic problems in resolving contextually ambiguous names that occur in full Java.

The input of the name analysis is a context-free tree constructed by the parser. The result is an attributed tree where all names have been resolved to appropriate name kinds, and have reference attributes denoting the appropriate declaration node. The purpose of the paper is to show how ambiguities related to names of variables, types, and packages, can be solved in a declarative and modular manner, using JastAdd. We will show how each of the rules in the language maps to a specific equation in the attribute grammar.

DemoJavaNames keeps just enough language constructs to illustrate the following name related concepts: multiple kinds of nested scopes, object-oriented inheritance, qualified names, shadowing and hiding, and multiple kinds of variables. To simplify the example we removed all language concepts unrelated to names and we also removed language concepts that duplicate name analysis problems, e.g., we only use classes and not interfaces. For brevity, we also removed some language constructs that do affect name binding, i.e., imports of types and access control. While they are not included

```

ast Prog ::= CompUnit*;
ast CompUnit ::= <packageName:String> ClassDecl*;
ast ClassDecl ::= <name:String> Super:Name BodyDecl*;

ast abstract BodyDecl;
ast FieldDecl : BodyDecl ::= FieldType:Name <name:String> Expr;
ast MemberClassDecl : BodyDecl ::= ClassDecl;
ast Initializer : BodyDecl ::= Block;

ast abstract Stmt;
ast Block : Stmt ::= Stmt*;
ast LocalVariableDecl: Stmt ::= VarType:Name <name:String> Expr;

ast abstract Expr;
ast abstract Name : Expr ::= <name:String>;
ast Dot : Name ::= Left:Name Right:Name;
ast ExpressionName : Name;
ast PackageName : Name;
ast TypeName : Name;

```

Listing 1.1. DemoJavaNames abstract grammar. A minimal subset of Java used to illustrate the problems in resolving contextually ambiguous names.

in the program listings we discuss how the implementation can be extended to handle these features as well.

Listing 1.1 presents the abstract grammar for DemoJavaNames. The Dot production that represents a qualified name requires further explanation. The parser is expected to build right recursive trees where the Left child is always a simple name while the Right child may be a Dot or a simple name. It is also worth noticing that the names in the grammar are context-sensitive, e.g., ExpressionName, TypeName, PackageName. We introduce context-free names and transformations into context-sensitive names in Section 3.3.

The type of names and variable declarations is needed to define qualified lookups and inherited members in later modules. We therefore define the type as an attribute of expressions and declarations. Listing 1.2 implements the type attribute as a reference to the appropriate declaration. To simplify equations in name binding modules we use a null object to represent unknown types. That way it is always possible to query an expression for members instead of handling the special case where the type is unknown.

The following sections present modules for name lookup and reclassification of ambiguous names followed by a discussion on how to extend the implementation to handle full Java.

3.1 Visible declarations

The most important contextual information used in name analysis is the set of visible declarations at each point in a program. Those declarations are then used to bind

```

syn ClassDecl Expr.type() = unknownType();
eq Dot.type() = getRight().type();
eq ExpressionName.type() = lookupVariable(name()) != null ?
    lookupVariable(name()).type() : unknownType();
eq TypeName.type() = lookupType(name()) != null ?
    lookupType(name()) : unknownType();
syn ClassDecl LocalVariableDecl.type() = getVarType().type();
syn ClassDecl FieldDecl.type() = getFieldType().type();

```

Listing 1.2. Type binding for DemoJavaNames where each expression and variable declaration is bound to a class declaration. A null object is used for unknown types to allow for a unified member lookup.

names in an actual context to their appropriate declarations. The name binding module in Listing 1.3 defines an attribute `inh` `VariableName.lookupVariable(String name)` that provides a binding through a reference to a named visible variable-declaration.

Language constructs that change the set of visible declarations, e.g., introduce new declarations or limit scope for an existing declaration, need to provide an equation for the lookup attribute. DemoJavaNames has two kinds of variables, `LocalVariableDeclarations` declared in `Blocks`, and `FieldDeclarations` declared in `ClassDecls`. The equations for lookup need thus be placed in the `Block` and `ClassDecl` types.

Nested scopes with shadowing The scope of a declaration is the region of the program in which the declaration can be referred to using a simple name. The scope of a declaration often involves nested language elements where declarations in one element are in scope in enclosed elements as well. A declaration may be shadowed in part of its scope by another declaration of the same name.

Both classes and blocks are allowed to be nested in DemoJavaNames and both implement shadowing as well. In Listing 1.3 the delegation to enclosing context, marked with ②, implements nested scopes. The eager return at first match, marked with ①, implements shadowing.

Declarations in a block have a *declare before use* policy. This is implemented by limiting the range of the block that is searched for declarations at ③. The equation is parameterized by the index of the `Stmt` in the element list and the search stops at the `Stmt` that encloses the name.

Inheritance The member fields of a class are not only the locally declared fields but also fields inherited from the superclass. A field is inherited if there is not a local field declaration that hides the field in the superclass. The eager return at ④ implements hiding and the delegation to the superclass at ⑤ implements inheritance.

Canonical type lookup The lookup of visible class declarations is implemented in a similar fashion to variable lookup. The main difference is how the lookup is handled at the compilation unit level. If the type is not found in the current compilation unit then

```

// visible variable or null
inh Variable Name.lookupVariable(String name);

// local variables in blocks
eq Block.getStmt(int index).lookupVariable(String name) {
③   for(int i = 0; i < index; i++)
       if(getStmt(i).isLocalVariableDecl(name))
①     return (LocalVariableDecl)getStmt(i);
②   return lookupVariable(name);
}
syn boolean Stmt.isLocalVariableDecl(String name) = false;
eq LocalVariableDecl.isLocalVariableDecl(String name) =
    name().equals(name);
inh Variable Block.lookupVariable(String name);
// member fields in classes
eq ClassDecl.getBodyDecl().lookupVariable(String name) {
    if(memberField(name) != null)
①     return memberField(name);
②   return lookupVariable(name);
}
// members including inheritance
syn FieldDecl ClassDecl.memberField(String name) {
    for(int i = 0; i < getNumBodyDecl(); i++)
        if(getBodyDecl(i).isField(name))
④     return (FieldDecl)getBodyDecl(i);
⑤   if(getSuper().type().memberField(name) != null)
        return getSuper().type().memberField(name);
    return null;
}
syn boolean BodyDecl.isField(String name) = false;
eq FieldDecl.isField(String name) = name().equals(name);
inh Variable ClassDecl.lookupVariable(String name);
// no more nested declarations
eq Prog.getCompUnit().lookupVariable(String name) = null;

// abstraction for FieldDecl and LocalVariableDecl
interface Variable {
    String name();
    ClassDecl type();
}
FieldDecl implements Variable;
LocalVariableDecl implements Variable;

```

Listing 1.3. Variable binding for DemoJavaNames. Shadowing is implemented by eager return statements marked ①. Nesting is implemented using delegation marked ②. Declare before use is implemented by limiting variable search to the current node index in ③.

```

// visible type or null object
inh ClassDecl Name.lookupType(String name);

// top level types in compilation unit
eq CompUnit.getClassDecl().lookupType(String name) {
    if(topLevelType(name) != null)
        return topLevelType(name);
    // declarations in same package
① return lookupCanonical(packageName(), name);
}
syn ClassDecl CompUnit.topLevelType(String name) {
    for(int i = 0; i < getNumClassDecl(); i++)
        if(getClassDecl(i).name().equals(name))
            return getClassDecl(i);
    return null;
}
// lookup a type using its canonical name
inh ClassDecl Name.lookupCanonical(String pack, String type);
eq Prog.getCompUnit().lookupCanonical(String p, String t) {
    for(int i = 0; i < getNumCompUnit(); i++)
② if(getCompUnit(i).packageName().equals(p) &&
    getCompUnit(i).topLevelType(t) != null)
        return getCompUnit(i).topLevelType(t);
    return null;
}
// member classes in class declaration
// analogous to the member fields implementation
eq ClassDecl.getBodyDecl().lookupType(String name) { ... }
// no more nested declarations
eq Prog.getCompUnit().lookupType(String name) = null;

```

Listing 1.4. Type lookup for DemoJavaNames.

the top level types in compilation units belonging to the same package are considered. This is implemented in Listing 1.4 by delegation ① to a canonical lookup that takes both the package name and type name into account ②. Inheritance of member classes is implemented in the same way as for variables.

3.2 Qualified lookup

The set of visible declarations for a qualified name depends on the target of the resolved name to the left of the dot. A valid `ExpressionName` can be preceded by either a `TypeName` or an `ExpressionName`. Either way, the `ExpressionName` refers to a member field in the `ClassDecl` that represents the type of the preceding expression. Listing 1.5 extends the name binding module with qualified lookup. The equation at ① defines the variable lookup to search the `ClassDecl` (that the qualifier's type is bound to) for members.

The lookup attribute is an inherited attribute and thus defined by an equation in an ancestor node. The qualifier to the left of the dot in a qualified name should provide the equation for the name on the right hand side of the dot. This is done by the common ancestor `Dot` which propagates the value of the equation from left to right for variables at ① and types at ②, overriding the lookup defined by an ancestor further up in the AST.

A valid `TypeName` can be preceded by either a `PackageName` or a `TypeName`. If the qualifier is a `PackageName` then the qualified name is the canonical name of the type. But if the qualifier is a `TypeName` then the name refers to a member type. There are thus different rules for the lookup depending on the kind of expression that precedes the name. The `Dot` therefore delegates the lookup to the expression at ② and searches for member types at ③ as the default strategy for expressions while the `PackageName` overrides the lookup at ④ to use canonical type names.

```

    eq Dot.getRight().lookupVariable(String name) =
①  getLeft().type().memberField(name);

    eq Dot.getRight().lookupType(String name) =
②  getLeft().qualifiedLookupType(name);

    syn ClassDecl Expr.qualifiedLookupType(String name) =
③  type().memberClass(name);
    eq PackageName.qualifiedLookupType(String typeName) =
④  lookupCanonical(name(), typeName);

```

Listing 1.5. Qualified lookup of types and fields.

3.3 Determine the meaning of names

The abstract syntax defined so far contains name nodes that are highly context sensitive and can thus not be built by a context-free parser. We now extend the abstract syntax with additional context-free name nodes that are used for gradually refining the names to reflect their semantic meaning.

The parser constructs unqualified name nodes only using the node type `ParseName`. These nodes are then refined by the name analysis to the resulting nodes listed in Listing 1.1. To simplify this computation, some of the refinements are done in intermediate steps, making use of two additional node types: `PackageOrTypeName` and `AmbiguousName`, see Listing 1.6.

Syntactic classification of names The first step in resolving names is to reclassify the `ParseName` nodes based on their immediate syntactic context. This way some nodes can be directly refined to their final class: `PackageName`, `TypeName`, or `ExpressionName`. However, for some names, the immediate syntactic context is not sufficient, in which

case the `ParseName` is refined to `PackageOrTypeName` (for names that must refer packages or types), or `AmbiguousName` (for names where the kind cannot yet be determined at all).

The Java language specification defines the classification process by describing a context and the expected name kind. For instance, a name is syntactically classified as a `TypeName` in the `extends` clause of a class declaration. We therefore introduce an inherited attribute `kind()` that describes the syntactic classification in a certain context by referring to an element in an enumeration of the above name kinds. Listing 1.6 shows the `kind()` attribute declaration at ②, the enumeration at ⑥, and the sample classification description at ③.

A qualifier in a qualified name may depend on the classification of the name it qualifies. For instance, a name is syntactically classified as a `PackageOrTypeName` to the left of the dot in a qualified `TypeName`. However, we still have the same requirement for equations in the ancestor as for qualified names. We therefore introduce another attribute `predKind()` which is delegated from right to left at ④ and the equation corresponding to the above example at ⑤.

The equations for `kind()` and `predKind()` complete the description of classification context and the transformation is almost trivial. The conditional rewrite at ① transforms a `ParseName` node into its syntactically classified counterpart. It is worth noticing that the dependences introduced by the `kind()` attribute equations in combination with demand driven rewriting causes qualified names to be classified from right to left.

Reclassification of contextually ambiguous names The next step is to reclassify contextually ambiguous names, i.e., `AmbiguousName` and `PackageOrTypeName`, in the context of visible declarations. An `AmbiguousName` is reclassified as an `ExpressionName` if there is a visible variable declaration with the same name. Otherwise, as a `TypeName` if there is a visible type declaration with the same name. Otherwise, as a `PackageName` if there is a visible package with the same name. The corresponding implementation is shown in Listing 1.7.

A contextually ambiguous name is resolved by binding it in the context of its qualifier. There is thus a dependence that the qualifier must be resolved before its right hand side can be resolved. We implement this dependence by making sure that all rewrite conditions in Listing 1.7 are false when the qualifier of a name is ambiguous. These conditions are false when there are no visible names. The type of an ambiguous name is `unknownType()` which has no visible member fields or types. To make the property hold we add an attribute `hasPackage(String name)` that is true when there is a visible package with that name and no ambiguous qualifiers. A qualified name `a.b.c` is thus first syntactically classified from right to left because of the dependences in the `kind()` attribute, and then reclassified from left to right.

3.4 Extensions to handle full Java

The `DemoJavaNames` language lacks some important name-related language constructs available in Java. This section describes the needed changes to the implementation to support full Java.

```

ast ParseName : Name;
ast PackageOrTypeName : Name;
ast AmbiguousName : Name;

① rewrite ParseName {
    when(kind() == Kind.PACKAGE_NAME)
    to Name new PackageName(name());
    when(kind() == Kind.TYPE_NAME)
    to Name new TypeName(name());
    when(kind() == Kind.EXPRESSION_NAME)
    to Name new ExpressionName(name());
    when(kind() == Kind.PACKAGE_OR_TYPE_NAME)
    to Name new PackageOrTypeName(name());
    when(kind() == Kind.AMBIGUOUS_NAME)
    to Name new AmbiguousName(name());
}

② inh Kind ParseName.kind();
eq Prog.getCompUnit().kind() = Kind.AMBIGUOUS_NAME;
③ eq ClassDecl.getSuper().kind() = Kind.PACKAGE_NAME;
eq FieldDecl.getFieldType().kind() = Kind.TYPE_NAME;
eq FieldDecl.getExpr().kind() = Kind.EXPRESSION_NAME;
eq LocalVariableDecl.getVarType().kind() = Kind.TYPE_NAME;
eq LocalVariableDecl.getExpr().kind() = Kind.EXPRESSION_NAME;

// propagate information from right to left
④ eq Dot.getLeft().kind() = getRight().predKind();
syn Kind Name.predKind() = Kind.AMBIGUOUS_NAME;
eq Dot.predKind() = getLeft().predKind();

eq PackageName.predKind() = Kind.PACKAGE_NAME;
⑤ eq TypeName.predKind() = Kind.PACKAGE_OR_TYPE_NAME;
eq ExpressionName.predKind() = Kind.AMBIGUOUS_NAME;
eq PackageOrTypeName.predKind() = Kind.PACKAGE_OR_TYPE_NAME;
eq AmbiguousName.predKind() = Kind.AMBIGUOUS_NAME;

⑥ class Kind {
    static Kind PACKAGE_NAME = new Kind();
    static Kind TYPE_NAME = new Kind();
    static Kind EXPRESSION_NAME = new Kind();
    static Kind PACKAGE_OR_TYPE_NAME = new Kind();
    static Kind AMBIGUOUS_NAME = new Kind();
}

```

Listing 1.6. Syntactic classification of names depending on their context. The context-free ParseName names are classified and rewritten to any of the five name kinds defined in Kind.

```

rewrite AmbiguousName {
  when(lookupVariable(name()) != null)
  to Name new ExpressionName(name());
  when(lookupType(name()) != null)
  to Name new TypeName(name());
  when(hasPackage(name()))
  to Name new PackageName(name());
}
rewrite PackageOrTypeName {
  when(lookupType(name()) != null)
  to Name new TypeName(name());
  when(hasPackage(name()))
  to Name new PackageName(name());
}
inh boolean Name.hasPackage(String name);
eq Program.getCompUnit().hasPackage(String name) {
  for(int i = 0; i < getNumCompUnit(); i++)
    if(getCompUnit(i).packageName().equals(name))
      return true;
  return false;
}
eq Dot.getRight().hasPackage(String name) =
  getLeft().qualifiedHasPackage(name);
syn boolean Expr.qualifiedHasPackage(String name) = false;
eq PackageName.qualifiedHasPackage(String name) =
  hasPackage(name() + '.' + name);

```

Listing 1.7. Reclassification of Contextually Ambiguous Names.

The implementation can be extended with more nested scopes by providing a new equation for the lookup attribute in each new scope. The various nested scopes are totally decoupled from each other using inherited attributes with parameters. The only constraint is that a scope is nested in another scope if they are on the same path to the root node. A `ForStmt` may for instance provide an equation (very similar to the equation for `Block` in Listing 1.3) that searches for `LocalVariableDeclarations` in its `init`-clause. Type imports extend the scope of type declarations and can be implemented by inserting a search for matching imports at ① in Listing 1.4. Java 5 [4] constructs such as *static imports* and the *enhanced for statement* can be supported using the same technique by adding a search for imported fields in the `CompUnit` node type and a lookup equation for local variable declarations in the enhanced for statement AST node.

Java supports access control where modifiers impose visibility constraints on names. Access control limits inheritance in that only non private accessible members are inherited from the superclass. This is easily implemented by adding a filter at ⑤ in Listing 1.3 that removes private non accessible fields. Access control also affects qualified lookups. The type of a qualifier must for instance be accessible and there are also additional constraints when the qualifier is an `ExpressionName`. Such behavior can be implemented

by filters at ① and ② in Listing 1.5. The specialized rules for `ExpressionName` may require the qualified lookup for fields to be extended to the variant used for types. The filter can then be placed on the `ExpressionName` qualifier.

`DemoJavaNames` supports inheritance from classes only while Java also supports interfaces. Interfaces complicate name analysis somewhat in that multiple inheritance may cause several fields with the same name to be inherited. This is only an error if a name refers to the ambiguous fields and the error detection can thus not occur in the `ClassDecl` directly but needs to be deferred to a `Name` node. This can be implemented by turning the lookup attribute into a set of references instead of a single reference. This does not affect the described modularization, but a few equations need to be changed to handle sets. Lookup equations defined to reference a single declaration are changed to a set of declarations, e.g., `eq Block.getStmt(int index).lookupVariable(String name)` in Listing 1.3 should return a set with a single reference to a variable declaration. Equations that expect a single reference need to ensure that the queried set contains a single reference and then extract that reference, e.g., `eq ExpressionName.type()` in Listing 1.2 should extract a single type declaration reference or return `unknownType()`. If a name binds to more than one element the name is ambiguous and a compile-time error is reported.

4 Related work

Transformation technology is commonly used in compiler construction to refine the AST to include context-sensitive information for later passes. Our approach differs from similar techniques in the use of context-dependent rewrites interleaved with attribute computations. Rewrites allow us to gradually define new concepts in terms of existing concepts, in the same way commonly used in informal language definitions. The fine-grained interaction between attribute computation and rewriting enables the immediate use of these concepts in equations without the need of defining separate passes. This is a key mechanism that allows complex analysis problems like Java name resolution to be broken down into small simple steps. To our knowledge, there are no other systems supporting similar mechanisms. Higher-order attribute grammars [17,12] allow the AST to be used as the only data structure, and combined with forwarding [15] it may be possible to use in a similar fashion, but as far as we know, forwarding has only been implemented in prototypes built on top of Haskell, and it is unclear how the practical performance would scale to full languages like Java.

The basic idea of name analysis for object-oriented languages based on explicit name bindings was used by ourselves earlier for simpler object-oriented languages [5], [6], and by Vorthmann in his visibility graph technique [18]. Vorthmann also uses a filtering technique to take care of constructs that limit declaration visibility. However, these approaches did not use context-dependent node types, which contribute substantially to making the approach modular. There is some other work aiming at separating the name analysis from other phases of a compiler, most notably the work on Kastens and Waite on an abstract data type for symbol tables [8]. The current version of Eli [13] contains an extensible library of modules for a large variety of scope rules, e.g., single inheritance, multiple inheritance, declare before use.

JastAdd lets context-dependent computations drive the transformations but it is interesting to compare to the opposite approach: letting transformations drive contextual computations commonly used in transformation systems such as ASF+SDF [14] and Stratego [16]. An important difference is that transformation systems typically handle contextual information by using an external database that is updated during the transformations. This requires the user to explicitly associate database updates with particular transformation rules or phases. The traversal order must thus take contextual dependences, which can be highly nonlocal, into account. In contrast, JastAdd uses the contextual dependences to derive a suitable traversal strategy. The Stratego system has a mechanism for dependent dynamic transformation rules [11], supporting certain context-dependent transformations, but it is not clear how this could be used for implementing name binding and similar problems in object-oriented languages.

5 Conclusions

We have presented a technique to implement name analysis for the Java programming language. The main contribution of the paper is to show how complex problems in name analysis including ambiguities related to names of variables, types, and packages can be solved in a declarative and modular way. The use of declarative attributes and contextual rewrites allow the implementation to be modularized in the same way as the language specification. Context-free as well as context dependent concepts in the language can be used directly in attributes and equations. It is worth noticing that the implementation can be freely modularized according to different criteria. A language extender may for instance choose to define a module with all attributes and equations related to a new language construct. The granularity of what can be modularized is a single attribute or equation, thereby providing excellent support for separation of concerns.

We have defined a small subset of Java that captures all the characteristic problems in resolving contextually ambiguous names. The implementation using JastAdd is less than 200 lines of code, most of it included in the paper. The source code and the JastAdd tool are available for download at [1]. The technique has been used to implement a full Java 1.4 compiler to verify that the technique scales to the full language. The system has been validated against the Jacks test-suite and passes more tests than the production quality compilers `javac` and `jikes` [1] while being roughly half the size of the handwritten `javac` compiler.

Acknowledgements

We are grateful to Calle Lejdfors and the anonymous reviewers for valuable feedback and helpful comments.

References

1. T. Ekman and G. Hedin. The JastAdd II compiler compiler system. <http://jastadd.cs.lth.se>.
2. T. Ekman and G. Hedin. Rewritable Reference Attributed Grammars. In *Proceedings of ECOOP 2004*, volume 3086 of *LNCS*. Springer-Verlag, 2004.

3. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification Second Edition*. Addison-Wesley, Boston, Mass., 2000.
4. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification Third Edition*. Addison-Wesley, Boston, Mass., 2005.
5. G. Hedin. An overview of door attribute grammars. In *Proceedings of Compiler Construction 1994*, volume 786 of *LNCS*, pages 31–51. Springer-Verlag, 1994.
6. G. Hedin. Reference attribute grammars. In *Informatica (Slovenia)*, 24(3), 2000.
7. G. Hedin and E. Magnusson. JastAdd: an aspect-oriented compiler construction system. *Science of Computer Programming*, 47(1):37–58, 2003.
8. U. Kastens and W. M. Waite. An abstract data type for name analysis. *Acta Informatica*, 28(6):539–558, 1991.
9. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *ECOOP 2001*, volume 2072 of *LNCS*, pages 327–355. Springer-Verlag, 2001.
10. D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, June 1968. Correction: *Mathematical Systems Theory* 5, 1, pp. 95-96 (March 1971).
11. K. Olmos and E. Visser. Composing source-to-source data-flow transformations with rewriting strategies and dependent dynamic rewrite rules. In *Proceedings of Compiler Construction 2005*, volume 3443 of *LNCS*. Springer-Verlag, 2005.
12. J. Saraiva. *Purely functional implementation of attribute grammars*. PhD thesis, Utrecht University, The Netherlands, 1999.
13. A. Sloane, W. M. Waite, and U. Kastens. Eli - translator construction made easy. <http://eli-project.sourceforge.net/>.
14. M. van den Brand and P. Klint. The ASF+SDF MetaEnvironment. <http://www.cwi.nl/htbin/sen1/twiki/bin/view/SEN1/MetaEnvironment>.
15. E. Van Wyk, O. d. Moor, K. Backhouse, and P. Kwiatkowski. Forwarding in attribute grammars for modular language design. In *Proceedings of Compiler Construction 2002*, volume 2304 of *LNCS*, pages 128–142. Springer-Verlag, 2002.
16. E. Visser, M. Bravenboer, and R. Vermaas. Stratego: Strategies for Program Transformation. <http://www.program-transformation.org/Stratego/WebHome>.
17. H. H. Vogt, S. D. Swierstra, and M. F. Kuiper. Higher order attribute grammars. In *Proceedings of the SIGPLAN '89 Conference on Programming language design and implementation*, pages 131–145. ACM Press, 1989.
18. S. A. Vorthmann. Modelling and specifying name visibility and binding semantics. Technical Report CMU//CS-93-158, 1993.