Sound and Extensible Renaming for Java

Max Schäfer   Torbjörn Ekman   Oege de Moor
Programming Tools Group, University of Oxford, UK
{max.schaefer, torbjorn.ekman, oege.de.moor}@comlab.ox.ac.uk

Abstract

Descriptive names are crucial to understand code. However, good names are notoriously hard to choose and manually changing a globally visible name can be a maintenance nightmare. Hence, tool support for automated renaming is an essential aid for developers and widely supported by popular development environments.

This work improves on two limitations in current refactoring tools: too weak preconditions that lead to unsoundness where names do not bind to the correct declarations after renaming, and too strong preconditions that prevent renaming of certain programs. We identify two main reasons for unsoundness: complex name lookup rules make it hard to define sufficient preconditions, and new language features require additional preconditions. We alleviate both problems by presenting a novel extensible technique for creating symbolic names that are guaranteed to bind to a desired entity in a particular context by inverting lookup functions. The inverted lookup functions can then be tailored to create qualified names where otherwise a conflict would occur, allowing the refactoring to proceed and improve on the problem with too strong preconditions.

We have implemented renaming for Java as an extension to the JastAdd Extensible Java Compiler and integrated it in Eclipse. We show examples for which other refactoring engines have too weak preconditions, as well as examples where our approach succeeds in renaming entities by inserting qualifications. To validate the extensibility of the approach we have implemented renaming support for Java 5 and AspectJ like inter-type declarations as modular extensions to the initial Java 1.4 refactoring engine. The renaming engine is only a few thousand lines of code including extensions and performance is on par with industrial strength refactoring tools.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors

General Terms Languages

Keywords Refactoring, name analysis, renaming, extensible compilers

1. Introduction

Renaming is one of the most commonly used refactorings [OJ90]. Well chosen names for classes, methods, and fields, play a key role in software systems to complement documentation, but as systems evolve it is important to be able to change these names to reflect updated designs.

Most languages allow globally visible declarations, and renaming therefore requires a global analysis to find which source files to change. To avoid manual inspection of the entire source code, automated tool support for such name based refactorings provide an invaluable aid in everyday development. An important property of refactorings is that they are behaviour preserving [Opd92, Fow00]. This is particularly important for global refactorings, such as renaming, where it is not reasonable to manually inspect the performed transformations. Behaviour preservation is most often ensured by having a set of preconditions that must be met for a refactoring to be valid.

In this work we look at two limitations of a purely precondition based approach. Too weak preconditions lead to unsoundness where programs will not compile after the refactoring, or even worse, to programs where names refer to different declarations after the renaming than before the transformation. During this work we found examples that lead to the severe latter case, exposing bugs in refactoring engines in common IDEs such as Eclipse, NetBeans, IntelliJ, and JBuilder [Ecl07, Net07, Jet07, JBU07]. Moreover, languages like Java provide several ways to refer to the same entity using different symbolic names. For example, a field shadowed by a local variable can still be accessed by qualifying it with `this`, and a type in a distant package can often be accessed using a qualified name. We thus do not want too strong preconditions either, since they prevent renaming programs where some minor qualifications would enable the refactoring.

This work improves on too strong preconditions that prevent refactorings, by using more flexible renaming, and present a systematic approach to avoid too weak preconditions which lead to unsoundness. Rather than stating preconditions that guarantee behaviour preservation, we use the following correctness criterion: Rename refactorings should preserve the invariant that only names are affected by the refactoring, and that each name refers to the same declared entity before and after the transformation. This is slightly weaker than behaviour preservation, which is impossible to achieve in the presence of dynamic class loading and reflection, but on the other hand it provides us with an implementation strategy.

We present a novel technique for creating symbolic names that are guaranteed to bind to a desired entity in a particular context by inverting lookup functions. By applying this operation for each bound name that could possibly be affected by a renaming operation we guarantee that our correctness invariant described above is preserved. This also allows us to re-qualify names that would otherwise refer to different declarations before and after the refactoring. If no symbolic name can be computed for a desired context we abort the rename operation and revert all changes. We have previously showed how name lookup can be implemented in a modular syntax directed fashion using attribute grammars [EH06]. In this work we show how to systematically invert each lookup rule by providing a corresponding inverted rule to each lookup rule to which creates a symbolic name. The rules can be tailored for various levels of qualification which enable flexible renaming where the desired intrusiveness can be selected by the tool developer or even the tool user. The notion of inverted lookup aligns well with Opdyke's preconditions for renaming, where he states that the actual conditions are closely tied to name lookup for a particular language [Opd92].

A major challenge for analysis tools is how to handle changes and extensions to a language. Indeed, we found several examples of unsoundness in refactoring engines due to unsupported language extensions, e.g., static imports as introduced in Java 5. To address this problem we show how to extend renaming specifications in a modular fashion to support new language features. The close correspondence between lookup and renaming enables the compiler implementor to preserve the renaming invariant by adding a reverse rule each time she implements a language construct that affects name binding.

We have implemented a complete renaming engine for Java as a modular extension to the JastAdd Extensible Java Compiler (JastAddJ) [EH07] and integrated it as a plugin to Eclipse[1]. The engine has been validated against both automatically generated tests, and our own renaming test suite for benchmarking refactoring engines. The suite exposes several bugs due to too weak preconditions in other major refactoring engines which causes the binding invariant to be violated. Moreover, it includes numerous examples where our approach succeeds in renaming entities by inserting qualifications where other refactoring engines fail.

Extensibility is validated by adding renaming support for Java 5 and AspectJ like inter-type declarations as modular extensions to the Java 1.4 refactoring engine. Both extensions can be used individually with their corresponding compilers or together to support renaming for the combined set of language features. All in all the implementation is less than 3000 lines of code.

The renaming infrastructure can also be used as a building block when implementing more high-level refactorings. Refactorings that rearrange or produce new code need to ensure that the binding structure is preserved. A common framework handling these low-level details allows the implementor to focus on issues germane to the refactoring at hand. This could reduce the number of naming related bugs which we have found to be pervasive in all major refactoring engines.

In summary, we present a systematic approach to renaming Java based on inverted lookup rules. The approach is:

Sound when each lookup rule in the name analysis has a corresponding inversion rule.

Flexible in that the level of qualification in names introduced during renaming can be tailored as desired.

Modular where each language construct is specified in isolation using syntax directed rules.

Extensible since both lookup rules and inversion rules are implemented in a declarative modular fashion and automatically composed into a global solution.

The rest of this paper is structured as follows. In Section 2 we give an introduction to renaming and describe deficiencies in existing refactoring engines. We also give an introduction to the correctness criterion used throughout the paper. An introduction to name lookup using JastAdd is given in Section 3. Section 4 gives a detailed presentation of the reverse lookup strategy and shows how to tailor the desired level of name qualification. We show how to support renaming of an extended language in Section 5.The approach is evaluated in Section 6 with respect to its correctness, performance, and extensibility. Related work is discussed in Section 7 and we finally conclude and present future work in Section 8.

2. Example: Rename Variable

Throughout this paper, we will take the Rename Variable refactoring as our running example. In standard Java terminology [GJSB05], both local variables (including parameters) and fields (either static or instance) are referred to simply as *variables*. The aim of the Rename Variable refactoring

[1] All tools including source are available for download at `http://jastadd.org/refactoring-tools`.

```
class A {
 int x;
 A(int y) {
   x = y;
 }
}
```

Figure 1. A simple example program to refactor

```
class A {
  int x;
  A(int newX) {
    x = newX;
  }
}
```

Figure 2. A simple example program, refactored

is to change the name of a variable and make all necessary changes to the program such that all bindings are preserved.

In this section, we will give several example programs to refactor. We will use these examples to contrast our implementation with the refactoring engine of Eclipse, which is mature and widely used.

A first example program is shown in Figure 1. There are two variable declarations in this program, one for the field `x`, which is an instance member field of class `A`, and one for the parameter `y` of method `m`, which is itself an instance member method of class `A`.

The field `x` is referenced once in the body of `m` by the simple name `x`. We collectively refer to all expressions that refer to a field as (field) *accesses*, and likewise for other named entities such as types and methods. Beside the example here, there are many more possible accesses that could also refer to the field `x` from inside method `m`, for example **`this`**`.x`, `((A)`**`this`**`).x`, or even `A.`**`this`**`.x`.

The local variable `y` is also referenced once, by the simple name `y` (which is, of course, a local variable access). In contrast to fields, there is no way to qualify local variable accesses. A local variable is thus either directly visible or inaccessible.

Now our little example program is perhaps not in very good style, since the name of the constructor parameter `y` bears no relation to the name of the field it is used to initialise, which may be a bit confusing. We could, for example, rename the parameter to `newX`; this is easily accomplished, for example, in Eclipse, and will yield the refactored code in Figure 2, in which the parameter declaration and all references to it use the new name.

Observe that the refactored program has the same binding structure as the original, i.e., every access still refers to the same entity as in the original program; our refactoring was successful.

```
class A {
  int x;
  A(int x) {
    this.x = x;
  }
}
```

Figure 3. A simple example program, refactored again

Another commonly seen idiom is for the parameter to have the *same* name as the field it initialises. But trying to perform this refactoring will fail, for example, in Eclipse since the newly renamed parameter collides with the field.

The same refactoring will, however, be successfully performed if we first change the access to field `x` to **`this`**`.x` manually. Ideally, one would like the IDE to carry out this sort of adjustment automatically. Our implementation does this, and yields the code in Figure 3.

Again, the binding structure of the refactored program is the same, although the access `x` has changed to **`this`**`.x`. To achieve this, our implementation proceeds as follows: For every access occurring in the program, we know which declaration it *should* bind to (the same one as in the original program); if needed, we compute a new access that will in fact bind to that declaration in the refactored program, and replace the old access by the new one. This ensures that the binding structure of the refactored program is the same as that of the input program, i.e. the refactoring is correct.

The key observation about the access computation function is that it is a (partial) right inverse to the lookup function: We can think of the lookup function at a certain program position p as a partial function

$$\mathit{lookup}_p\colon \mathit{access} \rightharpoonup \mathit{decl}$$

that determines for a given access the declaration it refers to (if any). The access computation function at the same position should now be a partial function

$$\mathit{access}_p\colon \mathit{decl} \rightharpoonup \mathit{access}$$

that determines for a given declaration an access that refers to it (if there is one). Correctness of our access computation means that

$$\mathit{lookup}_p(\mathit{access}_p(d)) = d$$

for every declaration d and program position p, if $\mathit{access}_p(d)$ is defined.

We will not formally prove that our refactoring engine fulfills this condition, but it provides a basic guideline for implementation. Our access computation is directly modelled after the lookup machinery of the JastAddJ Java compiler with the code for access computation closely mirroring the code for lookup. This ensures that we will not forget one of the many border cases that exist in a complex language like Java.

```
class A {
  public static void main(String[] args) {
    final int y = 23;
    new Thread() {
      int x = 42;
      void run() {
        System.out.println(y);
      }
    }.start();
  }
}
```

Figure 4. A more complex example

```
class A {
  public static void main(String[] args) {
    final int x = 23;
    new Thread() {
      int x = 42;
      void run() {
        System.out.println(x);
      }
    }.start();
  }
}
```

Figure 5. A more complex example, wrongly refactored

Traditionally, many implementations of automatic refactorings have tried to avoid tricky cases by checking a set of preconditions before performing the refactoring. However, these preconditions are formulated from scratch and it is very hard to ensure that they really cover every nook and cranny.

When we tried to rename `newX` to `x` above, the preconditions were strong enough and warned about the name collision. In fact, we argued that they were too strong, since the refactoring can easily be carried out with only slightly more invasive changes to the user's code. However, in Figure 4 we have a not much more complex program where Eclipse's preconditions are not strong enough.

This little program constructs a thread and runs it; the thread will print `23`. Similar to before, we have a field `x` and a local variable `y`, but this time their scopes are nested in the opposite way. If we tell Eclipse to rename the local variable to `x`, it obliges, but creates the output program seen in Figure 5 whose binding structure is *not* the same as the input program's. Incidentally, its behaviour is different as well: the refactored program prints `42`.

Our implementation does not check preconditions in advance. Instead, when a rename is performed, it tries to adjust all accesses such that they resolve to the same declaration as before. In our last example, the refactoring engine tries to compute an access to the newly renamed local variable `x` from inside the thread class. There it will discover that there is in fact no such access, reject the refactoring, and undo all previous changes.

```
import static java.lang.Math.*;

class Indiana {
  static double myPI = 3.2;
  static double CircleArea(double r) {
    return PI*r*r;
  }
}
```

Figure 6. Example program using the static import feature

It may seem to be prohibitive to adjust *all* accesses occurring in the input program. Intuitively, it is clear that most of them will not be influenced by the refactoring. But which ones do need adjustment? For one, certainly every reference to the entity being renamed has to be updated. But as the example above shows there might be other entities which are "endangered" by the rename and will need to have some or all of their accesses adjusted. Towards the end of Section 4 we will discuss a very simple approximation to determine the endangered set that works well in practice.

Another major issue for refactoring engines is language evolution: Since its inception, the Java language has gone through seven major revisions (Java 1.0 to 1.4, Java 5, and Java 6) with an eighth on the horizon. While some of these revisions mostly concerned the standard library or minor issues of language syntax, the transition from Java 1.4 to Java 5, in particular, brought with it a wealth of new features and concepts, which refactoring engines are, of course, expected to support.

To pick just one example, let us look at the static import feature. It allows the programmer to import static fields and methods of classes so that they do not have to be explicitly qualified, which is very useful for frequently used constants like `Math.PI`.

But the introduction of static imports also had a significant impact on the way name lookup is performed: Previously, only types could be imported, so **`import`** statements never affected the visibility of variables. Static imports do, however, and what affects lookup also affects renaming.

Take, for example, the program in Figure 6. The class `Indiana` is located in a compilation unit that statically imports the field `Math.PI` so that it can be accessed using the simple name `PI`; at the same time, the class also defines a static field named `myPI`.

If we want to rename `myPI` to `PI`, we must also qualify the access to `Math.PI`, as our implementation does, yielding the program of Figure 7. Before the introduction of static imports, when renaming the field in this example it would have been enough to check for collisions with other variables in class `Indiana`, but now we also need to consider the import declarations of the surrounding compilation unit.

```
import static java.lang.Math.*;

class Indiana {
  static double PI = 3.2;
  static double CircleArea(double r) {
    return Math.PI*r*r;
  }
}
```

Figure 7. Example program with static import, refactored

Two of the most popular Java IDEs, Netbeans and IntelliJ, fail to take this into account and incorrectly refactor this example. Eclipse and JBuilder both detect the name clash in this case, but fail on a very similar example where we instead import a static method and shadow it locally [EESV08]. Thus none of surveyed IDEs offers complete support for static imports.

Our implementation covers this example as well as all other Java 5 features. The refactoring engine for Java 1.4 is implemented as an extension of JastAddJ's Java 1.4 frontend with its access computation modelled after the lookup rules. To support Java 5, we only need to implement a corresponding extension for the Java 1.5 module of the frontend. Since all the lookup rules for the new language features are carefully implemented there, we only need to equally carefully translate them into access computation rules.

3. Lookup

The approach we present to flexible renaming is closely related to name lookup and its implementation in the JastAddJ compiler. We therefore give an introduction to that approach as background before presenting the reverse lookup strategy in Section 4. A more detailed description and a thorough evaluation are given in [EH07, EH06].

The purpose of name lookup is to bind each symbolic name to a corresponding declaration. This is traditionally done by traversing the program's abstract syntax tree and populating a symbol table with declarations that are in scope at various locations in the tree. This often leads to scheduling problems of traversals due to dependencies between analyses such as name name lookup and type analysis [AET08]. We therefore take a different approach in JastAddJ and use the AST itself as a symbol table, by specifying name lookup as a function of the tree location for each symbolic name. Lookup rules are specified in a syntax directed fashion over the AST extended with additional graph structure such as name bindings, type hierarchies, etc. That technique gives us the following properties:

Modularity in that we can specify the name lookup for each language construct that affects scoping in isolation.

Composability in that rules for individual language constructs can be combined to support the complete language automatically.

```
class X {
 int a;
}
class Y {
 int b;
 class Z extend X {
   int c;
   void m(boolean d) {
     int e;
     if(d) {
       int f;
       •
     }
   }
 }
}
```

Figure 8. Name lookup in Java with numerous nested scopes and inheritance. Declarations `a` to `f` are all accessible with their simple names from the position marked •

Declarativity in that we need not specify the order in which these rules are executed or combined.

Extensibility in that we can support lookup for language extensions by either adding new rules or refine existing rules.

Object-oriented languages with nested classes and inheritance provide many challenges from a name lookup point of view. Consider the Java example in Figure 8. Variable declarations a-f are all accessible with a simple name in the location marked •. To bind `f` we search the local block; `e` is visible since it is declared in an enclosing block; `d` is a parameter in an enclosing method; the field `c` is a local member field in the current class; `a` is an inherited member field from its superclass; `b` is visible since it is declared in an enclosing class. This strategy can be viewed as if lookup progresses outwards lexically with the exception that member lookup does a detour upwards the inheritance hierarchy.

We now present how Java name lookup with lexically nested structures and inheritance can be implemented using attribute grammars in JastAdd. First we present the static structure of the language and then show how to add name lookup on top of that structure using syntax directed rules.

3.1 Abstract Grammars

JastAdd uses an abstract grammar to model the structure of the AST. Consider the grammar in Figure 9 which models a subset of Java with nested blocks, nested classes, inheritance, and qualified access. It consists of a set of productions where each production describes an AST node type and a list of children, which can be either subtrees or terminals. For example, a `ClassDecl` has a string typed terminal child named `ID` to hold the name of the class, an optional child named `Super` which is an `Access`, and a list of `BodyDecl` children.

```
ClassDecl ::=
 <ID:String> [Super:Access] BodyDecl*;

abstract BodyDecl;
FieldDecl : BodyDecl ::=
 Type:Access <ID:String> [Init:Expr];
MethodDecl : BodyDecl ::=
 Type:Access <ID:String> ParameterDecl*
  [Block];
MemberClass : BodyDecl ::=
 ClassDecl;

ParameterDecl ::=
 Type:Access <ID:String>;

abstract Stmt;
Block : Stmt ::=
 Stmt*;
VariableDecl : Stmt ::=
 Type:Access <ID:String> [Init:Expr];

abstract Expr;
abstract Access : Expr;
VarAccess : Access ::= <ID:String>;
TypeAccess : Access ::= <ID:String>;
Dot : Access ::= Left:Expr Right:Access;
```

Figure 9. An abstract grammar for an object-oriented language with nested blocks, nested classes, and inheritance, used to introduce name lookup

A class will be generated for each production with getters and setters for accessing the children. For example, `ClassDecl` will have a getter `String getID()` for its terminal child. The optional child `Super` exists when the flag `boolean hasSuper()` is true, and can then be queried using `Access getSuper()`. The list of body declarations has getters for length, `int getNumBodyDecl()`, access to individual elements, `BodyDecl getBodyDecl(int i)`, and iteration by `Iterable<BodyDecl> getBodyDecls()`.

A production can inherit from another production, which translates into inheritance between the classes generated from them. For instance, `FieldDecl` inherits the abstract node type `BodyDecl`. All nodes implicitly inherit `ASTNode`, just like all Java classes extend `Object`. The type `ASTNode` holds generic node traversal code and other behaviour common to all tree nodes.

3.2 Lookup and type analysis external API

Lookup and type analysis are specified using declarative attribute grammars. Attributes can be seen as normal methods when invoked from imperative Java code, but an attribute evaluation engine derives a suitable evaluation order between dependent attributes. JastAdd allows attributes to be reference valued, i.e., refer to other nodes in the AST. Name lookup is then cast into the problem of binding a name to its corresponding declaration. Type analysis is similarly understood as binding each expression to a type declaration representing its static type.

```
// Lookup is used to find a visible
// declaration in a particular context
Variable ASTNode.lookupVariable(String name);
MethodDecl ASTNode.lookupMethod(String name);
TypeDecl ASTNode.lookupType(String name);

// The lookup framework is used to bind
// accesses to corresponding declarations
Declaration ASTNode.lookup(Access acc);

// Each expression has a type and binds
// to its corresponding declaration
TypeDecl Expr.type();
```

Figure 10. The API for name lookup and type analysis relevant to renaming. The node types are declared in Figure 9.

Throughout this presentation we will use the API in Figure 10. It is possible to lookup variables, methods, and types from all nodes in the AST. We define a common interface `Variable` for `VariableDecl`, `ParameterDecl`, and `FieldDecl` since they are treated the same during lookup. We have a similar interface `Declaration` to abstract over all kinds of declarations. Lookup is then used to bind each access to a declaration through the attribute `lookup()`. The type of an expression is similarly accessible through the `type()` attribute. The following sections will show how these attributes are implemented.

3.3 Nesting with shadowing

The purpose of name lookup is to find the declaration that corresponds to a symbolic name in a particular context. Attribute grammars provide inherited attributes which enable abstraction over the current context. An inherited attribute is declared in a node and any ancestral node may provide an equation defining the value for that attribute. That way the symbolic name need not be aware if it is, for example, nested in a block, method, or field initialiser.

Consider the code snippet in Figure 11. We declare an inherited attribute named `lookupVariable(String name)` in the `Access` type using an inter-type declaration. This will allow all access nodes to lookup variables visible from their particular contexts. Inter-type declarations are implemented outside their classes but otherwise act as if they were declared locally. This enables us to group attribute definitions by their concerns rather than by the type of their receiver.

Next we define an equation for that attribute provided by the `Block` node since it introduces a new scope. The rule gives an equation for the `lookupVariable(String name)` attribute in each subtree that can be reached from the `Block` node using the accessor `getStmt(int i)`. Such a subtree corresponds to the ith statement within the block.

To look up a variable from such a statement, we first search for local declarations which is done by the attribute `localVariable(String name)`. If there is no such declaration we call `lookupVariable(name)` from the `Block` to find a declaration in the enclosing context of the block. Since we only delegate to the enclosing context when no local declarations are found we effectively implement shadowing. Additional filters can be used to also check that variables are not used before they are defined, and to enforce accessibility restrictions.

Notice that the equation is only valid for the subtrees reachable through `getStmt(int i)` and not the `Block` itself. When we call `lookupVariable(name)` on the `Block` we therefore read the inherited attribute declared in `Block` which has an equation in an ancestor to the block, e.g., a nested block or method. This kind of chaining of inherited equations allow us to gradually progress outwards through the nested scopes when searching for declarations.

The local search is implemented using a synthesised attribute to iterate over local declarations to find a `Variable` in the `Block` itself. For this presentation they can be seen as virtual methods added as inter-type declarations. However, the implementation may not contain any externally visible side-effects, which enables caching of attribute values.

These kinds of attributes and equations are used for all language constructs introducing new scopes. A `MethodDecl` needs, for instance, an `eq lookupVariable(String name)` that is identical to the one in `Block`, a synthesised attribute `localVariable(String s)` that iterates over its formal parameters, and an equation `isVariable(String name)` in `ParameterDecl` identical to the one in `VariableDecl`.

The same technique is used to implement lookup for types and methods. Type lookup is for instance quite similar to variable lookup in that equations are needed to lookup local classes in a block, member classes that are inherited, and nested classes. There is also an outermost equation for `lookupType(String name)` in `CompilationUnit` that handles imports.

3.4 Member inheritance

A `ClassDecl` is similar to a `Block` in that it is a nested structure that introduces a scope for variables, in this case member fields, and therefore needs to provides equations for `lookupVariable(String name)`. It differs a bit in that not only local declarations should be considered but also member inherited from superclasses. Nested scope lookup is, so to speak, an outwards movement that searches enclosing scopes, while member lookup searches upwards in the inheritance hierarchy.

Consider the implementation in Figure 12. Besides the usual `localVariable(String name)` in `ClassDecl` and `isVariable(String name)` in `FieldDecl`, we also introduce an attribute `memberFields(String name)` which is synthesised. This attribute includes local declarations as well as declarations inherited from superclasses. Notice that

```
inh Variable Access.
  lookupVariable(String name);
eq Block.getStmt(int i).
  lookupVariable(String name)
{
 // find local declarations
 Variable v = localVariable(name);
 if(v != null) return v;
 // otherwise delegate to enclosing context
 return lookupVariable(name);
}
// the block will need to delegate lookup
inh Variable Block.
  lookupVariable(String name);

syn Variable Block.localVariable(String name)
{
 // iterate over contained statements
 for(Stmt s : getStmts())
   if(s.isVariable(name))
     return (Variable)s;
 return null;
}

// most nodes are not variable declarations
syn boolean ASTNode.isVariable(String name)
 = false;
// only declarations with matching names
eq VariableDecl.isVariable(String name)
 = name.equals(getID());
```

Figure 11. Nested scopes variable lookup implementation.

we include superclasses transitively by recursively invoking `memberFields` on the type of the specified superclass name. This introduces a dependency between name lookup and type analysis, but fortunately the attribute evaluation scheme will schedule the computations automatically.

3.5 Qualified access

A `ClassDecl` plays two roles from a name lookup perspective. It provides an equation for its members when using `lookupVariable(String name)`. This includes its members as well as declarations from lexically enclosing constructs, e.g., nested classes, final local variables in enclosing methods for anonymous classes, statically imported fields, etc. However, during qualified lookup, e.g., accessing a field in an object, only members should be considered as illustrated in Figure 13.

The equation for `memberFields(String name)` is thus kept separate not only for understandability and separation of concerns but also to be reused during qualified lookup. Qualified lookup can then be implemented by looking up remote members similar to the way members are looked up in superclasses. A qualified name is modelled as two separate accesses that share a common `Dot` parent node.

```
syn Variable ClassDecl.
   localVariable(String name) {
 for(BodyDecl b : getBodyDecls())
   if(b.isVariable(name))
     return (Variable)b;
 return null;
}

eq FieldDecl.isVariable(String name)
 = name.equals(getID());

eq ClassDecl.getBodyDecl(int i).
             lookupVariable(String name) {
 // search for member fields
 Variable v = memberField(name);
 if(v != null) return v;
 // otherwise delegate to enclosing context
 return lookupVariable(name);
}

syn Variable ClassDecl.
  memberField(String name){
 // search local declarations
 Variable v = localVariable(name);
 if(v != null) return v;
 // search inherited members
 return hasSuper() ?
   getSuper().type().memberField(name)
 : null;
}
```

Figure 12. Inheritance variable lookup implementation.

```
class X {
 int a;
}
class Y {
 int b;
 class Z extend X {
   int c;
   Z z = new Z();
   // OK: local member 'c' in Z
   int i1 = z.c;
   // OK: member 'a' inherited from X
   int i2 = z.a;
   // ERROR: qualified access does not
   // search enclosing classes
   int i3 = z.b;
 }
}
```

Figure 13. Qualified field access

```
Access ASTNode.accessVariable(Variable v);
Access ASTNode.accessMethod(MethodDecl m);
Access ASTNode.accessType(TypeDecl t);
```

Figure 14. The API for access construction

We can then simply take the type of the left hand side of a qualified name, and search its member fields for a matching name.

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

The same technique is used for `memberTypes` when accessing a type by its fully qualified name, i.e., including its package name and possibly enclosing type names.

Finally, we can now outline the implementation of the `lookup` function itself: In an invocation of the form `p.lookup(acc)`, the node `p` indicates the context (i.e., the position in the syntax tree) from where the lookup of access `acc` is performed. If the access is qualified, we recursively look up the access to the right of the `Dot`. Otherwise it is either a `VarAccess`, a `MethodAccess`, a `TypeAccess`. All of these are just wrappers for simple names that tell us which lookup function to delegate to. In particular, if `acc` is a `VarAccess` with name `n`, we delegate to `p.lookupVariable(n)`.

4. Specifying Renaming

We will now give an executable specification of access computation as a right inverse of lookup: each lookup equation from the previous section is inverted, yielding a corresponding equation for access computation.

As a first step, we outline a very simple access computation that never produces qualified accesses. It can be used as the basis of a non-intrusive renaming implementation that will never add extra qualifications; in particular, this implementation cannot perform the refactoring from Figure 3. We then show how it can be extended to produce a possibly qualified access as its result.

Similar inverses are required for type lookup and method lookup, and can be constructed using the same approach. Together, they form the external API given in Figure 14, which is closely related to the lookup API in Figure 10. Our informal statement that access computation should be a partial right inverse of lookup can now be cast into a more concrete form: For every variable `v` and AST node `p`, if `p.accessVariable(v)` is not **`null`**, then we should have

```
p.lookup(p.accessVariable(v)) == v
```

and likewise for the other two access computation functions. It is worth noting that renaming should never affect visibility, and need therefore not deal with access control modifiers such as **`public`** and **`private`**.

```
syn Access Block.accessLocal(Variable v)
{
  //iterate over contained statements
  for(Stmt s : getStmts())
    if(s == v)
      //and search for a particular variable
      return new VarAccess(v.getID());
  return null;
}

syn Access ClassDecl.accessLocal(Variable v)
{
  for(BodyDecl b : getBodyDecls())
    if(b == v)
      return new VarAccess(v.getID());
  return null;
}
```

Figure 15. Inverting local lookup in blocks and classes

```
eq Block.getStmt(int i).
    accessVariable(Variable v) {
  Access acc = accessLocal(v);
  if(acc != null) return acc;
  return accessVariable(v);
}
```

Figure 16. First attempt at inverting a lookup rule

4.1 Non-Intrusive Renaming

The simple access computation we introduce here will only tell us whether a variable can be accessed through its simple name. Given a variable `v`, it will thus either return a `VarAccess` containing `v.getID()` if the variable is visible, or **`null`** if it is inaccessible.

The equations implementing the `lookupVariable` attribute in Figures 11 and 12 all follow a very simple pattern: First, we try perform a local lookup on the current node, and if that does not yield any result we recursively invoke ourselves on another node.

In the equation specifying lookup at a statement inside a block, for example, we first invoke `localVariable`; if that fails, `lookupVariable` is recursively invoked on the parent node, i.e. the block itself.

First we invert the attribute `Block.localVariable`, which is easily done. Since the code is very similar, we also invert the corresponding equation for `Class`es; both are given in Figure 15. Note that in a valid Java program there can be only one declaration with a given name in every block or class, hence the given code snippets really are inverses to the local lookup attributes given in the previous section.

Encouraged by our success we now would like to invert the equation for `lookupVariable` on a block statement. Our first attempt might look like the code in Figure 16.

```
eq Block.getStmt(int i).
    accessVariable(Variable v) {
  Access acc = accessLocal(v);
  if(acc != null) return acc;
  acc = accessVariable(v);
  // check for shadowing in block
  if(localVariable(acc) != null)
    return null;
  return acc;
}
```

Figure 17. Right inverse of a lookup rule

This, however, is not quite right. Consider, for example, the following program fragment:

```
class A {
  int x;
  void m() {
    int x;
    •
  }
}
```

Assume we want to construct an access to the field `x` in `A` from the position marked •. Since that position is the second statement within a block, the equation will be evaluated with parameter `i` set to `1` (indexing is zero-based).

The auxiliary method `accessLocal` will not find the declaration we are looking for: although there is a declaration for a local variable `x`, this declaration is not equal to the declaration we are trying to access. Hence we will invoke `accessVariable` again, one node higher up in the syntax tree. Eventually it will return the access `x`, which does in fact refer to the field if seen *outside* the body of method `m`. Inside that body, however, it does not, but will instead refer to the like named local variable of `m`.

Generally speaking, we can not always directly return the access computed recursively at a higher node in the syntax tree, but we might need to adjust it to make sure it is still valid. However, we are not dealing with qualifiers just yet, so for the moment all we do is to check for shadowing, and fail (i.e., return **`null`**) if that occurs. The resulting code is in Figure 17.

Tracing all possible execution paths of this equation, one sees that this equation is indeed inverse to the one in Figure 11, under the assumption that local access and local lookup are inverses (which is easily seen) and that the recursive invocation of `accessVariable` on the parent node is inverse to the recursive invocation of `lookupVariable` (which acts as a sort of induction hypothesis).

The process of inverting is not entirely straightforward to automate, mainly owing to the fact that the equations can contain arbitrary Java code (without side effects). It is certainly systematic, though, and can be performed manually for every lookup rule without much difficulty.

```
class A {
  int x6;
}
class B extends A {
  int x5;
}
class C extends B {
  int x4;
  class D extends F {
    int x1;
    •
  }
}
class E {
  int x3;
}
class F extends E {
  int x2;
}
```

Figure 18. Qualified accesses in Java

Field name	Source	Bend	Safely qualified access
x1	D	D	**this**.x1
x2	F	D	**super**.x2
x3	E	D	((E)**this**).x3
x4	C	C	C.**this**.x4
x5	B	C	C.**super**.x5
x6	A	C	((A)C.**this**).x6

Table 1. Safely qualified accesses (cf. Figure 18)

4.2 Adding Qualifiers

Now we will show how to extend the above approach to add qualifications. Remember that field lookup in Java proceeds in an "outwards and upwards" motion: We first look among the member fields of the lexically enclosing class, then among all its supertypes, then among the next lexically nested class, then among that class' supertypes, and so on. When the field is finally found, it will be located in an ancestor class A of some class B lexically surrounding the point of lookup. We call the class A the *source* and the class B the *bend*. To illustrate all possible cases, Table 1 indicates for every field in the program in Figure 18 what its source and bend are when looked up from the position •. The table also gives a *safely qualified* access for each field, i.e. an access that would refer to the field even if it were shadowed in some way.

We can see, for example, that a field can always be safely accessed by qualifying with **this** if its source and bend class are both equal to the class lexically surrounding the point of lookup. Similar rules exist for the other qualifications, so in order to determine how a field can be accessed it is enough to compute its source and bend.

```
eq ClassDecl.getBodyDecl(int i).
    accessVariable(Variable v) {
  VarAccessInfo acc = accessMember(v);
  if(acc != null) return acc;
  acc = accessVariable(v);
  if(acc != null)
    return acc.moveInto(this);
  return null;
}

syn VarAccessInfo ClassDecl.
    accessMember(Variable v) {
  VarAccessInfo acc = accessLocal(v);
  if(acc != null) return v;
  if(hasSuper()) {
    acc = getSuper().type().accessMember(v);
    if(acc != null)
      return acc.moveDownTo(this);
  }
  return null;
}
```

Figure 19. Right inverse of lookup rule with requalification

Our improved version of accessVariable does no longer simply return a String, but a VarAccessInfo, which stores the source and bend of the field to be accessed (the *target*) as well as a flag indicating whether the field is shadowed or hidden (by a local variable or another field). In a second step, this information will be used to create an actual Access. We will at first omit this step for simplicity.

Corresponding to the two directions of movement during lookup, there are two basic situations where the access information has to be updated: Whenever we return from a lookup at a parent node we need to move the information *into* the inner scope (e.g., a class or a block), noting whether any shadowing could take place; upon returning from a lookup in a parent type we need to move the information *down to* the child type, again noting possible hiding. These two movements are achieved by methods moveInto and moveDownTo in VarAccessInfo.

For example, let us look at the inverses of memberField and lookupVariable when invoked on a body declaration inside a class, which are given in Figure 19. In the first case, after having computed an access to variable v from outside the class (for example, from its enclosing class), we need to move this access "into" the class: We check whether there is a member field of the same name that would shadow v (which is the target of our access computation), and if so set a flag to indicate that a qualifier will have to be added when we create an Access to v (see Figure 20).

Similarly, after having computed an access to a variable in the super class, we need to move this access "down to"

```
public VarAccessInfo moveInto(ClassDecl td)
{
  if(td.memberField(target.getID())!=null)
    needsQualifier = true;
  return this;
}

public VarAccessInfo moveDownTo(ClassDecl td)
{
  if(td.localVariable(target.getID())!=null)
    needsQualifier = true;
  return this;
}
```

Figure 20. Moving an access into and down to a type declaration

```
Access toAccess() {
  VarAccess va=new VarAccess(target.getID());
  if(needsQualifier) {
    if(bend == enclosingType()) {
    if(source==bend)
      return new Dot(new ThisAccess(), va);
    else if(source==bend.getSuper().type())
      return new Dot(new SuperAccess(), va);
    }
    return null;
  } else {
    return va;
  }
}
```

Figure 21. Generating accesses

the current class, checking whether there are any local fields that could hide the variable being accessed.

Having the moving methods return a `VarAccessInfo` leaves open the possibility that such a movement may fail and return **`null`**. This allows us to handle local variable accesses in the same way as field accesses: They are represented by objects of type `LocalVarAccessInfo` which extends `VarAccessInfo`. However, since local variable accesses can not be qualified, its `moveDownTo` and `moveInto` methods will always return **`null`** if any shadowing is detected.

Once all the required information is collected, we need to produce a Java expression to actually access the target field. This is handled by a method `toAccess` in class `VarAccessInfo`. Figure 21 shows a simplified implementation of the access generation, which can generate **`this`**- or **`super`**-qualified accesses:

The actual implementation goes further and can, in fact, generate all of the access forms in Table 1 including casts.

```
class A {
  int x;
}
class B extends A {
  int y;
}
class C {
  int m(B b) {
    return b.x;
  }
}
```

Figure 22. The need for merging accesses

In addition, it also checks a number of additional conditions to ensure that only correct accesses are generated[2].

It is certainly debatable if a refactoring engine should introduce complex qualifications like `((A)B.`**`this`**`).x` into a program, but we found it hard to draw an *a priori* boundary between "reasonable" and "unreasonable" qualifications. The degree of intrusiveness can be adjusted by changing the implementation of the `toAccess` methods, and it is certainly conceivable to make this configurable by the user. If no qualifications are ever added, we again obtain an Eclipse-style maximally unobtrusive refactoring.

Correctness of this enhanced implementation is no longer as easy to see as for the simple implementation, but it still follows the implementation of lookup equation by equation.

4.3 Access Merging

One final subtlety which we have ignored so far has to do with qualifications. Consider the example program in Figure 22.

In method `C.m`, we access the field `x` of an object of type `B`. Now assume we want to rename that field to `y`. Obviously, we also need to adjust its (only) reference and compute a new access to put in its place. Our access computation suggests **`super`**`.y`, which is indeed a correct way of accessing the field from inside `B`. We cannot, however, simply insert this access in place of the reference, as that would result `b.`**`super`**`.y`, which is not valid Java. Instead we want to *merge* the access **`super`**`.y` with its qualifier `b`, yielding the access `((A)b).y`.

Intuitively, this merging step can be understood as substituting the qualifier for **`this`**: Since **`super`**`.y` in the above example is actually just a shorthand for `((A)`**`this`**`).y`, replacing **`this`** by `b` yields the desired result. For our purposes, the merging of a qualifier q and a field access a can

[2] For example, an access such as **`this`**`.x` is invalid inside a static method or a static initialiser, and hence should not be generated.

be described by three rewrite rules:

$q \oplus n$	$\rightarrow$	$q.n$	for any name n
$q \oplus \mathbf{this}.n$	$\rightarrow$	$q.n$	
$q \oplus \mathbf{super}.n$	$\rightarrow$	$((A)q).n$	where A is the superclass of q's type

In the last case, we will need to construct a type access referring to type `A`. This is done by an attribute `accessType`, whose implementation is modelled after `lookupType`. Computing an access to a type has to handle very similar problems like computing a variable access: in particular, types can also be shadowed or hidden by other types, and sometimes even be obscured by fields of the same name. Overall, its implementation is very similar to `accessVariable`.

4.4 Determining Endangered Declarations

We have seen above how to adjust accesses to make sure that they refer to a given declaration. But how do we decide which accesses need adjustment?

In general, it is clear that when renaming an entity `x` to `y`, the only declarations that can possible be endangered are those which themselves declare entities called either `x` and `y`. So a very straightforward approach would be to sweep the entire program for all simple names `x` and `y` and treat all of them as potentially endangered accesses.

This sounds somewhat expensive, however, so a more refined approach could try to make use of the language's lookup rules to narrow down the set of endangered accesses. Surprisingly, it turns out that this is not needed. As our evaluation in Section 6 shows, the naïve approach works quite well in practice, and has the additional advantage of being largely language independent.

5. Extending Renaming

One of the major features of JastAddJ is its extensibility which allows modular specification of language features. The name lookup presented in Section 3 can be modularly extended to handle new language features, such as the enhanced **`for`** statement and static imports in Java 5. Since our implementation of access computation has a direct correspondence to the name lookup implementation (and the computation of endangered accesses is language independent) we enjoy the same kind of modularity.

JastAddJ is actually implemented as a Java 1.4 compiler with Java 5 and AspectJ-like inter-type declarations implemented as modular extensions. In the same way, we have extended our Java 1.4 refactoring engine with support for Java 5 and inter-type declarations.

5.1 Java 5

There are quite a few language features in Java 5 that affect name binding. For instance, static imports of fields and methods make imported entities visible in the current compilation unit with their simple names, and generic types introduce named type variables. Specifying renaming for most of these features is fortunately quite simple in our framework.

```
eq CompilationUnit.getTypeDecl(int i)
   .lookupVariable(String name) {
 for(ImportDecl i : getImportDecl())
   if(i.importsField(name) != null)
     return i.importsField(name);
 for(ImportDecl i : getImportDecl())
   if(i.importsFieldOnDemand(name) != null)
     return i.importsFieldOnDemand(name);
 return lookupVariable(name);
}
// match name then return member
eq StaticImportDecl.importsField(String name)
 = name().equals(name) ?
   type().memberField(name) : null;
// on demand matches all members
eq StaticImportDeclOnDemand
   .importsFieldOnDemand(String name)
 = type().memberField(name);
```

Figure 23. Import static fields lookup implementation

```
eq CompilationUnit.getTypeDecl()
   .accessVariable(Variable decl) {
 for(ImportDecl i : getImportDecls())
   if(i.importsField(decl.name()) == decl)
     return new VarAccessInfo(decl);
 for(ImportDecl i : getImportDecls())
   if(i.importsFieldsOnDemand(decl.name())
      == decl)
     return new VarAccessInfo(decl);
 return accessVariable(decl);
}
```

Figure 24. Access computation for static import fields

Consider the task of supporting renaming for static imports. Import clauses may import static member fields and static member methods which are then made visible using their simple names in the current compilation unit. Like normal imports, static import clauses are either *named*, i.e. they only import a single field or method, or *on-demand*, which means that they import all static fields and methods of a given class.

The Java 5 extension adds attributes and equations related to variable lookup as shown in Figure 23. We first search named static imports, and then proceed to on-demand static imports if no match is found. Finally we delegate to the enclosing context if neither kind of import match. The code is slightly abbreviated in that a check that the imported field is accessible from the current compilation unit is left out.

These rules follow the well-known "try local lookup, then delegate" pattern, and hence are easily inverted (see Figure 24). Providing inverted rules is equally straight forward for most extensions. New type lookup rules are for instance

```
aspect X {
 static int x;
 int B.m() {
   return x+y;
 }
}

class B {
 int y;
}
```

Figure 25. A simple program using inter-type declarations

```
eq IntertypeMethodDecl.getBody().
   lookupVariable(String name) {
 Variable v = parameterDeclaration(name);
 if(v != null) return v;
 v = introducedType().memberFields(name);
 if(v != null) return v;
 return lookupVariable(name);
}
```

Figure 26. Variable lookup on inter-type methods

needed to account for type variables, but again they are easily inverted, with corresponding changes to the `moveInto` method to make sure that shadowing by type variables does not go unnoticed.

5.2 Inter-Type Declarations

Inter-type declarations are a powerful concept to separate concerns that cross-cut the static class hierarchy. They are part of the AspectJ language, enabling addition of new members to an already declared class [Tea].

Take for example the program in Figure 25. It contains both an aspect `X` and a class `B` with the aspect declaring an inter-type method `m` on `B`; to differentiate, we refer to `X` as `m`'s *host aspect*, while `B` is its *introduced type*. The method `m` behaves like a regular member method of `B` but the lookup rules for inter-type methods are slightly augmented. This allows them to access members of both the introduced type and the host aspect using simple names. The method `m` can thus access both members of `B`, like `y` in the example, and static members of `X`, like `x`.

The implementation of inter-type declarations introduces new node types to represent aspects, inter-type methods, and inter-type fields. The introduced members are similar to their non inter-type counterparts but have an additional attribute `introducedType()` that refers to the declaration they are introduced into. This allows variable lookup for inter-type methods to be easily implemented as shown in Figure 26: Parameters are looked up first, followed by members of the introduced type; then we delegate to the parent node, which is the host aspect.

```
eq IntertypeMethodDecl.getBody().
   accessVariable(Variable decl) {
 VarAccessInfo acc
   = accessParameterDeclaration(decl);
 if(acc != null) return acc;
 acc = introducedType().
   accessMemberField(decl);
 if(acc != null)
   return acc.moveInto((MethodDecl)this);
 acc = accessVariable(decl);
 if(acc != null)
   return acc.moveInto(this);
 return null;
}
```

Figure 27. Access computation on inter-type methods

Again, this lookup rule follows our basic pattern, so we can invert it to obtain the code in Figure 27. Observe that for the first `moveInto` we reuse already existing code that checks whether a type access would be shadowed by parameters; the second `moveInto` also needs to take member fields of the introduced type into account, which is done by a new method `moveInto(IntertypeMethodDecl)`.

Since the inter-type declarations extension is mostly concerned with names and follows the name lookup strategy from Section 3, no further extensions beyond implementing the new access rules are necessary.

6. Evaluation

6.1 Correctness

Along with our implementation we have developed a suite of several hundred test cases, about fifty of which come from Eclipse's refactoring test suite. These tests systematically explore both common and exotic cases for all the refactorings we have implemented and have been very helpful not only for validating our own implementation, but also for finding bugs in other IDEs. A commented list of examples that we found to be refactored incorrectly by the most recent version of Eclipse's refactoring engine can be found online [EESV08].

Of the around three hundred test cases, 10% require additional qualification to avoid shadowing and hence cannot be refactored by Eclipse. Some of these cases can be handled by IntelliJ, which can automatically add **this** qualifiers, but others, which would require more sophisticated qualification, cannot. The part of the test suite that deals with inter-type declarations can not directly be compared against other tools: Although AspectJ plugins are available for some of the major IDEs (notably AJDT for Eclipse [AJD]), none of them offers any refactoring support.

In order to convince ourselves that our refactorings work as expected on simple inputs, we have used the ASTGen syntax tree generator library [DDGM07], which has been used

Refactoring	Generator	TGI	CI	Succ: Ecl	Succ: JA	Diff
Rename Class	AllRelationships	108	88	88	88	0
Rename Method	SingleClassMethodReference	9450	9450	9450	9450	0
Rename Field	SingleClassFieldReference	5280	2824	2824	2824	0
	DualClassFieldReference	23760	7947	7947	7947	0

Table 2. Results of the automatically generated tests; Ecl = Eclipse, JA = our implementation; TGI = Total Generated Inputs; CI = Compilable Inputs; Succ = successfully refactored inputs, Diff = different outputs

before to detect bugs in refactoring software. This library provides a number of generators that can be used to generate a large number of input programs and it provides support for automatically performing refactorings on them using the Eclipse refactoring engine. We have written a simple driver program that in addition performs the same refactorings using our own implementation, and compares the output to Eclipse's.

Table 2 summarises the results of this experiment: The first column gives the name of the refactoring tested, the second column lists the test generator used (for details about the individual generators please refer to [DDGM07]); the following columns give information about the total number of generated test cases, the number of compilable test cases (refactoring uncompilable programs is possible, but the results would be hard to assess), and the number of inputs successfully refactored by Eclipse resp. our implementation.

The generated test programs are generally of a very simple structure (and the original paper did, indeed, not find any bugs in Eclipse's renaming support), so it is not surprising that both Eclipse and our own implementation are able to refactor all compilable inputs. Our implementation gives the exact same results on all programs (as seen in the last column), in particular it never produces uncompilable output programs.

6.2 Code Size

Next, we want to assess the code complexity of our implementation, both in absolute terms and in comparison to the JastAddJ compiler frontend.

Table 3 shows the code size of the JastAddJ frontend[3]. Its basic module, which implements the full Java 1.4 language, comprises about ten thousand lines of code. Both the extension module for Java 5 and the extension to handle inter-type declarations are significantly smaller. Either of them can separately be used together with the Java 1.4 frontend, or all three can be combined to yield a compiler for Java 5 with inter-type declarations (for this, an extra 76 lines of code are needed).

Our refactoring engine is partitioned in the same way as the frontend and also consists of three modules that can be combined with the corresponding modules from the frontend. Their code sizes are summarised in Table 4.

Module	**Total**
Java 1.4 Frontend	9990
Java 1.5 Frontend	6234
Inter-type Declarations	1824
ITD + Java 1.5 Integration	76

Table 3. Code size of the JastAddJ frontend

Module	**Total**	**Access**	**Framework**
Java 1.4 Refactoring	1902	1014	888
Java 1.5 Refactoring	538	284	254
Inter-type Declarations	211	211	0

Table 4. Code size of the refactoring engine

For every module, we indicate its total code size, how much code is devoted to access computation, and how much of "framework" code we need. For the Java 1.4 component, we notice that the size of the access computation code tallies well with the lookup code from the frontend, which is implemented in about 1100 lines. Framework code for this module includes driver programs, the computation of endangered accesses, and undo functionality.

For the Java 1.5 refactoring module, we see that only very little code needs to be added to the access computation. It also includes a modest amount of framework code that handles JastAddJ's internal representation of generic types and their instantiations and harmonises it with the rest of the refactoring code.

Perhaps surprisingly, the module implementing refactoring for inter-type declarations is even smaller than the Java 1.5 one. This is because inter-type declarations mainly affect lookup and so are handled very neatly by our approach without requiring any additional framework code.

Overall, these numbers show that by integrating the refactoring code tightly with the compiler and exploiting JastAdd's modularity and extensibility we are able to obtain a very concise implementation. The implementation of renaming refactorings in Eclipse, for instance, is at least three times as big.

[3] These and all the following source code line counts were obtained using David A. Wheeler's SLOCCount utility [Whe06].

6.3 Performance

Another important question to evaluate, of course, is if our implementation is practical on large input programs. To address this, we performed a number of renamings on the source code of the Jigsaw webserver [w3c06], which consists of about 100K lines of Java 1.4 code

The results of our experiments (as measured on an AMD Athlon 64 X2 machine running Linux 2.6.22) are put together in Table 5. We will briefly explain the data relating to type renaming. Interesting types to rename for evaluation purposes are on the one hand those which are referenced a lot, and on the other hand those which are rarely referenced. Hence we chose three of the most heavily used types (the classes `Attribute` and `Request` as well as the interface `HTTP`) and some rarely used classes to rename; the number of types referencing each of them is given in the third column[4].

For every refactoring to be performed, there is quite a significant startup time during which the program is loaded into memory and checked for errors (around 18 seconds for Jigsaw). In an IDE this step would normally already have been performed before the user initiates a refactoring, so we have not included it in our evaluation.

Once the program is loaded, the refactoring needs to determine the set of potentially endangered accesses, and then proceeds to rename the type and perform any other adjustments. The third column of the table gives the total number of endangered accesses, the fourth the time needed to find these accesses, and the last the total time for performing the refactoring. The time for adjusting accesses was well below 0.1s in every case. The bulk of the time was spent flushing internal attribute caches which the renaming invalidates, in turn triggering the garbage collector.

Nevertheless, we can observe that the overall time it takes to rename a type is around two seconds (regardless of their frequency of use), which is comparable to Eclipse's performance on the same tasks. Determining endangered accesses is quite fast, and although our approach is very coarse grained we still end up with manageable numbers of accesses to adjust.

An earlier version of our implementation had a more sophisticated algorithm that tried to avoid too many false positives, and indeed managed to give about 20% less endangered accesses than the current one. However, the extra effort needed more than outweighed the gain in precision, and lead to much slower refactorings.

The situation for field and local variable renaming is not much different. For the latter, we have chosen a variable with perhaps the most common name of all. Here, the discrepancy between actual references and endangered accesses as determined by our algorithm is very marked, but still not big enough to cause a substantial decrease in performance.

[4] These numbers were obtained using SemmleCode [Sem08].

In conclusion, the numbers show that our implementation compares very favourably to an industrial strength refactoring engine like Eclipse's. It can perform more sophisticated renamings, and handles corner cases on which Eclipse fails.

7. Related Work

7.1 Correctness of Refactorings

Correctness of refactorings has long been a primary concern in the literature. Opdyke [Opd92] and Roberts [Rob99] champion a precondition based approach. They specify global conditions which a program has to meet for the refactoring to be correct. The scope nesting rules of the languages treated in these theses (a subset of C++ and Smalltalk, respectively) are fairly simple. In particular, there are no nested classes, so the preconditions for renamings are very lightweight and not much attention is paid to them.

Later work [Cor04, Ett07] has focused on giving a semantics of the underlying language in order to rigorously prove that the given preconditions are sufficient to preserve program semantics. To keep the formal development manageable, however, the language has to be quite simple.

One possible way to overcome this limitation may be to formalise both the underlying language and the refactorings to be performed in an interactive theorem prover, which will keep track of all the details to be proved. This approach, taken, for example, by Sultana and Thompson [ST08], suffers from a lack of automation; even seemingly trivial statements have to be proven in excruciating detail, again restricting the scope of the work.

Much work has also gone into the precise definition of type-based refactorings, i.e. refactorings that change the type structure of a program [Tip03, BTF05, vDD04, DKTE04]. Such refactorings, however, have to deal with quite different issues, mostly type constraints which are orthogonal to renaming.

7.2 Specification of Refactoring

The high-level declarative specification of refactorings has seen quite some interest in the research community. Mens *et al.* [MDJ02] specify refactorings as transformations on a largely language-independent graph representation of programs that concentrates on those aspects of the source code to be preserved by the refactoring. Their definition of preservation properties is very close to the access preservation invariant we use here. However, their graph representation does not seem suitable for specifying renamings, since name lookup is not explicitly represented in the program graph.

Another approach is introduced in [GM06]. Based on a formalisation of Java in rewriting logic, they give executable specifications of several refactorings in Maude. While their implementations are quite concise, it seems that this approach mainly excels at local refactorings (such as the Rename Temporary example they give) and would perhaps

Renamed Entity	Name	References	Number of EA	Time to find EA	Total Time
Toplevel Type	`Attribute`	177	1464	0.4s	2.2s
	`Request`	132	887	0.3s	2.0s
	`HTTP`	106	1100	0.3s	2.2s
	`FileEditor`	1	0	0.3s	1.4s
	`Main`	0	2	0.2s	1.9s
Nested Type	`Alert`	1	0	0.1s	1.8s
	`Openner`	1	2	0.2s	1.9s
Field	`EDITABLE`	86	538	0.3s	3.3s
	`OK`	74	185	0.3s	3.3s
	`INTERNAL_SERVER_ERROR`	46	142	0.2s	2.6s
	`DEFAULT_SSL_ENABLED`	1	0	0.1s	3.1s
	`img`	0	17	0.1s	3.2s
Local Variable	`i`	4	3055	0.2s	1.5s

Table 5. Refactoring Jigsaw: Some Performance Measurements (averaged over 20 runs); EA = endangered accesses

be more cumbersome to use for refactorings that affect the whole program.

A quite different and more flexible approach is taken by the JunGL language [VEdM06], which is a domain specific language especially suited for implementing refactorings. The refactorings can manipulate an extensible graph representation of the program with user-definable edges to capture program properties of interest. An important feature is path queries, which can be used to express, for example, name lookup in an elegant short form. So far, however, JunGL has only been used to implement refactorings on subsets of languages while we support full Java 5.

7.3 Aspect-Oriented Refactoring

Our implementation supports refactoring in the presence of inter-type declarations, one of the major features of aspect oriented programming. The interplay between aspect oriented programming and refactoring has been explored by others.

For example, Cole and Borba [CB05] give some preconditions for refactoring AspectJ code, while Hanenberg *et al.* [HOU03] show how to make well-known refactorings such as Rename Method aspect-aware. Both papers, however, concentrate almost exclusively on how to handle pointcuts and advice and do not consider inter-type declarations. It would be interesting to see how our approach can be extended to take these more dynamic features into account.

8. Conclusions

In this paper we have investigated two problems related to preconditions for renaming refactorings. Too weak preconditions results in unsound renamings where symbolic names refer to different declared entities before and after the refactoring, and too strong preconditions prevent certain renamings from being carried out.

We have presented an approach to specifying renaming refactorings that is closely coupled to the way name lookup is implemented in a compiler. Inverted lookup rules are used to create accesses that are guaranteed to bind a symbolic name to a specific declared entity. By applying this operation to each name possibly affected by the refactoring, we guarantee that the binding structure is preserved and avoid too weak preconditions. The accesses created during inverted lookup can be tailored to introduce qualifications which enables renamings that would otherwise be prohibited by too strong preconditions.

Another challenge is to update refactorings as the language evolves. In our experiments we have not found a single refactoring engine for Java that handles static imports properly. The direct correspondence between lookup rules and access rules help us avoid many pitfalls that went unnoticed in other refactoring engines. The presented approach effectively improves on state-of-the-art, both in correctness and flexibility in allowed renamings.

We have implemented renaming, including pluggable extensions for Java 5 and AspectJ like inter-type declarations, as modular extensions to the JastAdd Extensible Java Compiler. The tool is available as a plugin for Eclipse, with an implementation size of less than 3000 lines of code, and it achieves similar performance to mature refactoring engines. We have created a renaming test suite that our implementation passes but which exposes numerous bugs and limitations in industrial strength refactoring tools.

We believe that this renaming approach is general and can benefit both other refactorings and other languages beside Java. High-level refactorings can use it as a building block to preserve bindings when rearranging code, a common source of bugs in current tools. We have previously shown that name lookup for the Modelica language can be implemented in the same way as shown here [ÅEH08], and it would be interesting to apply the renaming approach to that implementation as well.

In order to give a formal justification of the soundness of our approach, it is of central importance to prove the inversion property of access computation and name lookup. We have developed a general framework for formalising reference attribute grammars in the theorem prover Coq, and are now working on a mechanised proof of the inversion property. The goal of that work is to provide a theoretical foundation for the practical implementation techniques presented in this paper.

It has been argued that a refactoring engine cannot be both thorough and fast. For example, the authors of the Refactoring Browser [BR99] write in [Fow00]: "Computer scientists tend to focus on all of the boundary cases that a particular approach will not handle. The fact is that most programs are not boundary cases (...)" We believe that our approach shows that at least for a statically typed language it is possible to achieve reasonable speed and also handle boundary cases.

Acknowledgments

We would like to thank Mathieu Verbaere for sharing his experience on refactorings and their implementation with us, and for his many helpful and encouraging comments on every stage of this work. Special thanks also go to Damien Sereni for help with proof reading.

References

[ÅEH08] Johan Åkesson, Torbjörn Ekman, and Görel Hedin. Development of a Modelica Compiler Using JastAdd. *Electronic Notes in Theoretical Computer Science*, 203(2):117–131, 2008.

[AET08] Pavel Avgustinov, Torbjörn Ekman, and Julian Tibble. Modularity First: A Case for Mixing AOP and Attribute Grammars. In *Aspect-Oriented Software Development (AOSD)*. ACM Press, 2008.

[AJD] AspectJ Development Tools 1.5.1. `http://www.eclipse.org/ajdt`.

[BR99] John Brant and Don Roberts. The Smalltalk Refactoring Browser. `http://st-www.cs.uiuc.edu/users/brant/Refactory/`, 1999.

[BTF05] Ittai Balaban, Frank Tip, and Robert Fuhrer. Refactoring support for class library migration. In *Proceedings of the 20th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 265–279, 2005.

[CB05] Leonardo Cole and Paulo Borba. Deriving Refactorings for AspectJ. In *Aspect-Oriented Software Development (AOSD)*. ACM Press, 2005.

[Cor04] Márcio Lopes Cornélio. *Refactorings as Formal Refinements*. Ph.D. thesis, Universidade Federal de Pernambuco, 2004.

[DDGM07] Brett Daniel, Danny Dig, Kely Garcia, and Darko Marinov. Automated Testing of Refactoring Engines. In *Proceedings of ESEC/FSE'07*. ACM Press, 2007.

[DKTE04] Alan Donovan, Adam Kiezun, Matthew S. Tschantz, and Michael D. Ernst. Converting Java Programs to use Generic Libraries. In *Object-Oriented Programming, Systems and Languages*, pages 15–34, 2004.

[Ecl07] Eclipse 3.3.1. `http://www.eclipse.org`, 2007.

[EESV08] Torbjörn Ekman, Ran Ettinger, Max Schäfer, and Mathieu Verbaere. Refactoring bugs in Eclipse, IDEA and Visual Studio, 2008. `http://progtools.comlab.ox.ac.uk/refactoring/bugreports`.

[EH06] Torbjörn Ekman and Görel Hedin. Modular name analysis for Java using JastAdd. In *Generative and Transformational Techniques in Software Engineering, International Summer School, GTTSE 2005*, volume 4143 of *LNCS*. Springer, 2006.

[EH07] Torbjörn Ekman and Görel Hedin. The JastAdd Extensible Java Compiler. In Richard P. Gabriel, editor, *ACM Conference on Object-Oriented Programming, Systems and Languages (OOPSLA)*. ACM Press, 2007.

[Ett07] Ran Ettinger. *Refactoring via Program Slicing and Sliding*. D.Phil. thesis, Computing Laboratory, Oxford, UK, 2007.

[Fow00] Martin Fowler. *Refactoring: improving the design of existing code*. Addison Wesley, 2000.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Prentice Hall, 3rd edition, 2005.

[GM06] Alejandra Garrido and José Meseguer. Formal Specification and Verification of Java Refactorings. In *Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM)*, 2006.

[HOU03] Stefan Hanenberg, Christian Oberschulte, and Rainer Unland. Refactoring of Aspect-Oriented Software. In *Net.ObjectDays*, 2003.

[JBU07] JBuilder 2007. `http://www.codegear.com/products/jbuilder`, 2007.

[Jet07] IntelliJ IDEA 7.0.1. `http://www.jetbrains.com`, 2007.

[MDJ02] Tom Mens, Serge DeMeyer, and Dirk Janssens. Formalising behaviour preserving program transformations. In *Graph Transformation*, volume 2505 of *Lecture Notes in Computer Science*, pages 286–301, 2002.

[Net07] Netbeans 6.0. `http://www.netbeans.com`, 2007.

[OJ90] William F. Opdyke and Ralph E. Johnson. Refactoring: An aid in designing application frameworks and evolving object-oriented systems. In *Proceedings of Symposium on Object-Oriented Programming Emphasizing Practical Applications (SOOPPA)*, September 1990.

[Opd92] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[Rob99] Donald F. Roberts. *Practical Analysis for Refactoring*. PhD thesis, University of Illinois at Urbana-Champaign, 1999.

[Sem08] Semmle. SemmleCode. `http://semmle.com`, 2008.

[ST08] Nik Sultana and Simon Thompson. Mechanical Verification of Refactorings. In *Workshop on Partial Evaluation and Program Manipulation*. ACM SIGPLAN, January 2008.

[Tea] The AspectJ Team. The AspectJ Programming Guide.

[Tip03] Frank Tip. Refactoring for generalization using type constraints. In *Proceedings of the 18th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 13–26, 2003.

[vDD04] Daniel von Dincklage and Amer Diwan. Converting Java classes to use generics. In *Proceedings of the 19th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 1–14, 2004.

[VEdM06] Mathieu Verbaere, Ran Ettinger, and Oege de Moor. JunGL: a Scripting Language for Refactoring. In *International Conference on Software Engineering (ICSE'06)*, 2006.

[w3c06] w3c. Jigsaw. `http://www.w3.org/Jigsaw/`, 2006.

[Whe06] David Wheeler. SLOCCount. `http://www.dwheeler.com/sloccount/`, 2006.