RC25201 (W1108-027) August 8, 2011
Computer Science

IBM Research Report

A Comprehensive Approach to Naming and Accessibility in Refactoring Java Programs

Max Schäfer
Oxford University

Andreas Thies, Friedrich Steimann
Fernuniversität in Hagen

Frank Tip
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich

LIMITED DISTRIBUTION NOTICE: This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties). Copies may be requested from IBM T. J. Watson Research Center , P. O. Box 218, Yorktown Heights, NY 10598 USA (email: reports@us.ibm.com). Some reports are available on the internet at http://domino.watson.ibm.com/library/CyberDig.nsf/home .

A Comprehensive Approach to Naming and Accessibility in Refactoring Java Programs

Max Schäfer
Oxford University
Dept. of Computer Science
max.schaefer@cs.ox.ac.uk

Andreas Thies Friedrich Steimann
Fernuniversität in Hagen
andreas.thies@fernuni-hagen.de
steimann@acm.org

Frank Tip
IBM T.J. Watson
Research Center
ftip@us.ibm.com

Abstract

Automated tool support for refactoring is now widely available for mainstream programming languages such as Java. However, current refactoring tools are still quite fragile in practice and often fail to preserve program behavior or compilability. This is mainly because analyzing and transforming source code requires consideration of many language features that complicate program analysis, in particular intricate name lookup and access control rules. This paper introduces J_L, a lookup-free, access control-free representation of Java programs. We present algorithms for translating Java programs into J_L and vice versa, thereby making it possible to formulate refactorings entirely at the level of J_L and to rely on the translations to take care of naming and accessibility issues. We demonstrate how complex refactorings become more robust and powerful when lifted to J_L. Our approach has been implemented using the JastAddJ compiler framework, and evaluated by systematically performing two commonly used refactorings on an extensive suite of real-world Java applications. The evaluation shows that our tool correctly handles many cases where current refactoring tools fail to handle the complex rules for name binding and accessibility in Java.

1 Introduction

Refactoring is the process of restructuring a program by means of behavior-preserving source code transformations, themselves called refactorings [5, 13]. Over the past decade, automated tool support for refactoring has become available for mainstream programming languages such as Java and C#, and popular IDEs such as Eclipse and VisualStudio support a growing number of refactorings. However, even state-of-the-art tools are still quite fragile, and often render refactored programs uncompilable or silently change program behavior. As a result, despite an evident need for refactoring in software development, acceptance of *refactoring tools* seems to be lagging [14].

An important cause for this lacking robustness is the fact that refactoring tools analyze and transform programs at the source level, which is significantly more challenging than working on some convenient intermediate representation, as compilers do. Source level programs contain many features such as nested classes, method overloading, and access modifiers that require great care when applying program transformations and that writers of compiler optimizations simply do not have to worry about.

In the context of Java, two particularly vexing problems are name lookup and access control. Java's rules for finding the declaration that a type or variable name refers to are quite intricate and context dependent. The combination of inheritance and lexical scoping, in particular, makes name lookup highly non-modular so that any change in the binding of names to declarations can have repercussions throughout the program. Determining whether a declaration is accessible at a given position in the program is a similarly knotty problem, and of course the two problems are intertwined, since accessibility can influence the result of name lookup.

However, naming and accessibility are omnipresent in refactoring: any refactoring that introduces, moves or

deletes a declaration runs the risk of upsetting the program's binding of names to declarations. Similarly, when a refactoring moves a reference to a declaration, great care has to be taken to ensure that it still binds to the same declaration after the move. Failure to do so may either lead to an uncompilable output program or, even worse, a program that still compiles but behaves differently due to changed name resolution. Examples from both categories are easy to find even with state-of-the-art refactoring tools [19] such as the refactoring engines of Eclipse JDT [12] and IntelliJ IDEA [9].

In this paper, we propose a comprehensive solution to these issues in the form of J_L, a lookup-free, access control-free representation of Java programs. In J_L, declarations are not identified by potentially ambiguous names but by unique labels, and are accessed by *locked bindings* that directly refer to a label without any lookup or access control rules; consequently, a transformation cannot accidentally change name bindings or introduce unbound names. We provide translations from Java to J_L and vice versa, allowing refactorings to be formulated directly at the level of J_L. This higher level of abstraction allows the implementer to concentrate on the essence of a refactoring, with the complexities of name binding and access control preservation being taken care of by the (refactoring-independent) translation to and from Java. Our translation from J_L to Java is based on two key techniques:

1. **Reference construction.** We observe that the problem of unlocking the locked bindings in J_L, i.e., replacing them with normal Java names, is easily solved using a *reference construction* algorithm that, given a declaration, constructs a (possibly qualified) name which binds to this declaration. We show that such an algorithm can be systematically derived from a suitable specification of name lookup.

2. **Accessibility constraints.** We observe that adjusting accessibilities requires updating declarations rather than names, and show that accessibility requirements can be captured using constraint rules relating the accessibilities of different declarations. A solution to these constraints indicates how declared accessibilities have to be adjusted to ensure that the resulting Java program adheres to the access control rules.

J_L and the translations to and from Java form the basis of the JRRT system [20], a prototype refactoring engine built on the JastAddJ Java compiler front end [3], which supports a growing number of popular refactorings [17]. We evaluate this implementation both on the internal test suite of the Eclipse refactoring engine and on a large suite of real-world applications, showing that it correctly handles many cases where existing state-of-the-art tools fail or produce incorrect results.

In summary, our work makes the following main contributions:

- We propose J_L, a lookup-free, access control-free representation of Java programs, and show how existing refactorings become simpler and more powerful when expressed on that representation.

- We show how an algorithm for constructing potentially qualified references referring to a given declaration from a given program point can be derived from a suitable specification of Java name lookup.

- We demonstrate how the access control rules of Java can be captured by constraint rules which can be applied to a program yielding a set of constraints that are used to constrain possible refactoring transformations to avoid generating uncompilable programs.

- We combine reference construction and accessibility constraints into an algorithm for translating from J_L to Java and report on an experimental evaluation of a refactoring tool built on this approach.

The remainder of this paper is organized as follows: Section 2 motivates the need for a systematic treatment of naming and accessibility by means of some examples. Section 3 surveys the name binding rules of Java and shows how to derive a reference construction algorithm from a suitable implementation of name lookup. Section 4 gives an overview of the access control rules of Java and demonstrates how they can be captured using constraint rules. These two techniques are then integrated in Section 5 to yield a translation from J_L to Java. An implementation of our approach is presented in Section 6 and evaluated in Section 7. Finally, Section 8 puts our work into the broader context of the literature, and Section 9 concludes.

```
1  class A {
2    int x;
3    A(int newX) {
4      x = newX;
5    }
6  }
```

(a)

```
7  class A {
8    int x;
9    A(int x) {
10     this.x = x;
11   }
12 }
```

(b)

Figure 1: A simple example of RENAME

```
13 class A {
14  long x;
15  A() {
16   x = 42;
17  }
18  A(long v) {
19   x = v+19;
20  }
21 }
22
23 class C {
24  A a1 = new A(),
25    a2 =
26      new A(23);
27 }
```

(a)

```
28 class A {
29  long x;
30  A(int x) {
31   this.x = x;
32  }
33  A(long v) {
34   x = v+19;
35  }
36 }
37
38 class C {
39  A a1 = new A(42),
40    a2 =
41      new A((long)23);
42 }
```

(b)

Figure 2: A simple example of INTRODUCE PARAMETER

2 Motivating Examples

In this section, we will demonstrate by means of examples that naming and accessibility are among the core problems in tool-supported refactoring for Java: they make their appearance in almost every refactoring imaginable, from simple structural refactorings such as RENAME to complex type-based ones such as EXTRACT INTERFACE, and they are handled quite poorly by current state-of-the-art refactoring tools. This often leads to uncompilable refactored programs, or, even more insidiously, to programs that still compile but behave differently.

Due to their ubiquity, a comprehensive treatment of these issues is called for. At the end of this section we give an overview of our approach, which employs a novel lookup-free, access control-free representation of Java programs that allows addressing naming and accessibility problems in a refactoring-independent way.

2.1 Basic Naming Problems

The paradigmatic example of a refactoring that needs to deal with naming issues is, of course, the RENAME refactoring which allows the programmer to change the name of a declared entity (such as a class, field or method) and consistently updates all references to this entity to use the new name while avoiding name capture.

A simple example of this refactoring is shown in Fig. 1. In the original program, shown on the left, the constructor of class `A` has a parameter `newX` that is used to initialize field `x`. Let us assume that the programmer wants to rename `newX` to have the same name as the field that it initializes. A refactoring tool should then produce the program on the right-hand side, where we have highlighted changes in gray: both the parameter declaration and its single use have been updated to use the new name, and the reference to field `x` has been qualified with `this` to ensure that the reference still binds to the field after the renaming operation and is not captured by the renamed parameter.

In general, a plausible correctness criterion for RENAME is that it should preserve the program's *binding structure*: names should bind to the same declaration in the refactored program as in the original program.[1] Due to the complex lookup rules of Java and the delicate interplay between inheritance and lexical scoping this is not always easy to ensure. Section 3 will introduce a systematic way of constructing names that bind to a given declaration, making binding preservation easy to guarantee.

The preservation of name bindings is also desirable in many other refactorings. For instance, the INTRODUCE PARAMETER refactoring turns a constant expression appearing inside a constructor or method body into an additional parameter and adjusts call sites accordingly. An example of this refactoring is shown in Fig. 2, again with the original program on the left and the refactored program on the right. This refactoring has to deal with two naming issues: first, the introduced parameter should not lead to any name capture; this is avoided in the exam-

[1] This does not quite imply behavior preservation in Java, since the names of classes, interfaces, fields and methods can be determined by reflection. We do not tackle this notoriously difficult problem here.

```
43  package a;
44  class A {
45    a.B b;
46  }
47
48  package a;
49  class B {}
```

(a)

```
50  package a;
51  class A {
52    b.B b;
53  }
54
55  package b;
56  public class B {}
```

(b)

Figure 3: A simple example of MOVE CLASS

ple by qualifying the reference to field `x` on line 31 as in the previous example. Second, the changed signature of the constructor leads to a change in overloading resolution for the `new` expression on line 26: whereas originally constructor `A(long)` was the most specific choice, the constructor `A(int)` would now be selected; to avoid this unwanted change in program behavior, we have inserted an upcast to `long` on the argument, thus enforcing the same choice as before.

Given these two adjustments, the refactoring is binding preserving. Similar precautions have to be taken for any refactoring that introduces, moves or deletes declarations. Even in cases where we do want name bindings to change, for instance with the ENCAPSULATE FIELD refactoring where field references are turned into calls to accessor methods, we generally want them to change in a controlled manner.

This argues for a more high-level approach to name binding, in which a refactoring does not directly manipulate raw Java names with their complex qualification and lookup rules, but instead specifies which names are to keep their binding, and which ones are supposed to bind to different declarations. A common naming framework then takes care of introducing qualifiers or upcasts where necessary to achieve the desired binding structure.

Current state-of-the-art refactoring tools fail to handle name bindings correctly in many cases. Eclipse correctly diagnoses the shadowing problem in Fig. 1, but simply emits an error message and refuses to perform the renaming, while IDEA inserts the desired qualification. Both mishandle the example in Fig. 2: Eclipse fails to recognize either of the naming issues, while IDEA notices the shadowing but fails to prevent the change in overloading resolution.

```
57  package a;
58  public class A {
59    void m()
60    {
61      /* ... */
62    }
63    void n() {
64    ((A)new a.B())
65        .m();
66    }
67  }
68
69  package a;
70  public class B
71     extends a.A {
72    void m()
73    {
74      /* ... */
75    }
76  }
```

(a)

```
77  package a;
78  public class A {
79    protected void m()
80    {
81      /* ... */
82    }
83    void n() {
84    ((A)new b.B())
85      .m();
86    }
87  }
88
89  package b;
90  public class B
91     extends a.A {
92    protected void m()
93    {
94      /* ... */
95    }
96  }
```

(b)

Figure 4: An example of MOVE CLASS involving dynamic binding

2.2 Basic Accessibility Problems

Like name bindings, accessibility concerns also are handled poorly by current tools. For instance, consider a situation where the MOVE CLASS refactoring is applied to move class `B` in the Java program in Fig. 3 into another package, `b`. In order for `B` to remain accessible in the declaration on line 45, its accessibility has to be raised to `public`, as shown on line 56. Eclipse fails to notice this problem and produces an uncompilable program; IDEA emits a warning, but does not attempt to fix the issue.

While this problem is detected by the compiler and easily responded to, failure to adjust accessibility can be more detrimental in presence of dynamic binding. For instance, moving class `B` in Fig. 4 (a) to package `b` leaves the code compilable, but changes the meaning of the program, because it changes the status of `A.m` from being overridden to not being overridden, so that calling `m()` on a receiver of static type `A` and dynamic type `B` will no longer dispatch to the implementation in `B`. In Eclipse , this change of meaning goes unnoticed; IDEA warns that class `A` contains a reference to class `B`, but this is not indicative of the problem. An accessibility-aware refactor-

ing tool could instead suggest increasing the accessibility of A.m, and with it that of B.m (required by [7, §8.4.8.1]), to protected, as shown in Fig. 4 (b).

Because of their simplicity, both of the above problems require only a local analysis to detect and fix insufficient access modifiers. In real programs, however, there may be ripple effects that are difficult to foresee, and also non-obvious side conditions that prevent necessary fixes. For instance, if class B has a subclass C

```
package c;
public class C extends a.B {
    public void m() { /* ... */ }
}
```

in the above example, then the MOVE CLASS refactoring must be rejected because raising the accessibility of B.m as required for maintaining the overriding in B would make C.m override B.m (which it previously did not), causing invocations of method m() on receivers of static type A or B and dynamic type C to dispatch to the (newly overriding) method C.m, potentially changing the meaning of the program.

2.3 Naming and Accessibility problems in EXTRACT INTERFACE

For a somewhat more involved example of the subtle interactions of naming and accessibility with other language features and each other, let us consider the EXTRACT INTERFACE refactoring. The purpose of this refactoring is to encourage loose coupling by creating a new interface I that declares some of the methods defined in a given class C, and then updating declarations throughout the program to refer to I instead of C wherever possible [5, 28, 27].

While the essence of this refactoring is concerned with types, naming and accessibility issues also have to be handled. Consider, for instance, the example program of Fig. 5(a). For the purposes of this example, we will assume that the programmer wants to extract from class C an interface I that declares the method m.

Figure 5(b) shows the program after applying the refactoring. The new interface I appears on lines 117–119 of Fig. 5(b), and on line 121, type C was made to implement this new interface. We now explain the other changes.

Types The goal is to use the new interface wherever possible. However, some declarations that refer to type C cannot be changed to I.

For example, c2's type on line 131 cannot be changed because then the call to n on line 132 would not be type-correct as interface I does not declare a method n. On the other hand, o's type on line 131 and c1's type on line 130 can both be updated safely.

Accessibility Class C.B is declared private, meaning that it is not accessible outside class C. In particular, it is not accessible in the newly created interface I unless its accessibility is increased to at least package accessibility, as shown on line 122.

The newly created method I.m is implicitly public, hence method C.m, which overrides it, must be made public as well (line 123).

Names References to nested classes such as C.B must be qualified outside of their declaring class. Hence the signature of method I.m must use a qualified name (line 118).

A similar issue arises on line 135 where the type B of field f has become ambiguous as a result of increasing the accessibility of class C.B. This is resolved by using the qualified type name J.B.

Overloading Changing c1's type on line 130 to I renders the call to D's constructor on line 133 ambiguous because neither constructor is now more specific than the other. This ambiguity is resolved by inserting an upcast[2] on line 133.

While this example is arguably quite contrived, it shows that a complex interplay exists between typing, access control, and naming (including overloading resolution) that refactoring tools must be aware of. Neither Eclipse nor IDEA can carry out the example refactoring, since they require the extracted methods to be public already. If we change the example, making m public to begin with, both tools still fail to carry out some necessary adjustments, producing uncompilable output without a warning.

2.4 Naming and Accessibility Problems in PULL UP METHOD

Of course, EXTRACT INTERFACE is not the only refactoring that potentially faces such complications. Con-

[2] This cast only serves to ensure that the call is resolved to the correct declaration at compile-time and always succeeds at run-time.

```
97  class C {
98    private class B { }
99    void m(B b) { }
100   void n() { }
101 }
102
103
104
105
106
107 interface J { class B { } }
108
109 class D extends C implements J {
110   D(C c1, D d) { c1.m(null); }
111   D(C c2, C o) {
112     c2.n();
113     D d = new D(c2, null);
114   }
115   B f;
116 }
```

(a)

```
117 interface I {
118   void m(C.B b);                      // type name qualified
119 }
120
121 class C implements I {
122   class B { }                         // acc. increased
123   public void m(B b) { }              // acc. increased
124   void n() { }
125 }
126
127 interface J { class B { } }
128
129 class D extends C implements J {
130   D(I c1, D d) { c1.m(null); }        // type of c1 changed
131   D(C c2, I o) {                      // type of o changed
132     c2.n();
133     D d = new D((I)c2, null);         // cast inserted
134   }
135   J.B f;                              // type name qualified
136 }
```

(b)

Figure 5: Example application of the EXTRACT INTERFACE refactoring. Part (a) shows the original program; part (b) shows the program after the programmer has extracted from class `C` an interface `I` that declares method `m(C.B)`.

sider, for instance, the example program in Fig. 6(a), and assume the programmer wants to pull up method `C.foo(A)` into class `B` using the PULL UP METHOD refactoring. We observe the following about the refactored code in Fig. 6(b):

Accessibility Method `foo(A)` calls `C.baz`, a `private` method that is not accessible in `B`. This issue is resolved by increasing `baz`'s accessibility to package on line 164.

Names Accessing the `static` method `baz` outside of its declaring class requires explicit qualification of the method call on line 160.

Overloading Moving method `foo(A)` into class `B` makes the call `foo(null)` on line 169 ambiguous because neither of the methods `B.foo(A)` and `E.foo(String)` is more specific than the other. This is resolved by adding an upcast on line 169.

In general, the PULL UP METHOD refactoring also needs to preserve certain subtype relationships. For example, consider a scenario where a programmer attempts to pull up method `foo(A)` into class `A`. In this scenario, the refactoring cannot be applied because the type of the argument `this` in the method call `baz(this)` on line 147 would become `A`, causing the call to become type-incorrect.

In summary, PULL UP METHOD requires careful analysis to respect subtyping, accessibility constraints, name and overloading resolution. Again, the example exceeds the capabilities of Eclipse and IDEA, who reject it.

2.5 Our Solution

The examples in this section have demonstrated that naming and accessibility issues are pervasive in refactoring. However, their treatment is largely orthogonal to the purpose of the refactoring: the goal is to preserve name binding in most cases, rebind names where necessary, and adjust access modifiers to satisfy access control rules.

Ideally, refactorings should work on a language where name bindings are always preserved except when they are explicitly rebound, and where access modifiers are automatically adjusted as necessary. The J_L representation we introduce in this paper is just such a language. In J_L, normal Java names are abolished in favor of locked references, written $\uparrow l$, where l is a label uniquely identifying

```
141  class A {  }
142  class B extends A {
143  }
144  class C extends B {
145    private static void baz(B p) {  }
146    public void foo(A q) {
147      baz(this);
148    }
149  }
150  class E extends B {
151    private static void foo(String r)
152    { }
153    void bar() { foo(null); }
154  }
```

(a)

```
157  class A {  }
158  class B extends A {
159    public void foo(A q) {
160      C.baz(this);              // qualification added
161    }
162  }
163  class C extends B {
164    static void baz(B p) {  } // acc. increased
165  }
166  class E extends B {
167    private static void foo(String r)
168    { }
169    void bar() { foo((String)null); } // cast added
170  }
```

(b)

Figure 6: Example application of the PULL UP METHOD refactoring: pulling up method `C.foo(A)` into `B`.

a declaration. These references directly bind to the declaration they refer to without regard to normal lookup and access control rules.

Of course, J_L is only to be used as an intermediate representation that simplifies the specification and implementation of refactorings. To make a J_L-based approach practical, we need a way to translate from Java to J_L and back. In the following two sections we will develop the technical machinery needed for this translation by showing how to construct references that bind to a given declaration from a given position in the program, and how to capture Java's accessibility rules using constraint rules. Section 5 will then show how to integrate these two techniques, and how to upgrade a Java-based refactoring specification to work on J_L, revisiting some of the examples in this section.

3 Reference Construction

In this section, we consider the problem of how to construct a (possibly qualified) reference that binds to a given declaration from a given program point.

More precisely, assume name lookup is given as a partial function

$$\mathit{lookup}\colon \text{ProgramPoint} \times \text{Reference} \rightharpoonup \text{Declaration}$$

that determines the declaration $d = \mathit{lookup}(p, r)$ a reference r at point p binds to, if any.

We want to define a complementary *reference construction* function

$$\mathit{access}\colon \text{ProgramPoint} \times \text{Declaration} \rightharpoonup \text{Reference}$$

that constructs a reference $r = \mathit{access}(p, d)$ under which declaration d can be accessed from point p. The correctness of this function with respect to lookup is expressed by the condition

$$\forall p, d.\mathit{lookup}(p, \mathit{access}(p, d)) = d. \tag{1}$$

In other words, if function access produces a reference r under which to access declaration d from point p, then that reference really does bind to d at p: access is a (partial) right inverse to lookup.

Given access, we can eliminate locked bindings from a program by simply replacing every locked binding $\uparrow l$ occurring at some program point p with the reference $\mathit{access}(p, l)$. If $\mathit{access}(p, l)$ is undefined, indicating that an appropriate reference cannot be constructed, the refactoring is aborted.

A trivial implementation of access that is undefined everywhere vacuously satisfies Equation 1, but is not useful for eliminating locked bindings. In this section, we will show how a declarative specification of Java name lookup using the attribute grammar formalism of the JastAdd system [4] can be systematically (if not quite automatically) inverted, yielding a practical implementation of access.

3.1 Name Lookup in Java

The Java Language Specification (JLS) introduces eight kinds of *declared entities* [7, §6.1]: packages, class types (including enum types), interface types (including annotation types), type parameters, methods, fields, parameters, and local variables. An entity is introduced by a declaration and can be referred to using a simple or qualified name.

Like the JLS, we will use the term *reference type* to mean "class type, interface type or array type" and *variable* to mean "field, parameter or local variable".

Every declared entity e has a *scope* [7, §6.3], which encompasses all program points at which e can be referred to using a simple name, as long as it is *visible*. If, however, the scope of another entity e' of the same name is nested inside the scope of e, then e' is said to *shadow* e [7, §6.3.1]. Inside the scope of e' entity e is no longer visible, and it is not possible to refer to e by its simple name; a qualified name has to be used instead.

Shadowing is distinct from *hiding* [7, §8.3]: a field declaration in a reference type T hides any declaration of a field with the same name in superclasses or superinterfaces of T, subject to accessibility restrictions detailed in Section 4. Similarly, static method declarations hide methods with the same signature in superclasses or superinterfaces [7, §8.4.8.2].

Shadowing and hiding are both distinct from *obscuring* [7, §6.3.2]: in some syntactic contexts, it is not a priori clear whether a name refers to a package, a type or a variable. In this case, variables are given priority over types, and types over packages. This means that there may be program points p where it is impossible to refer to a type or package e_1 by its simple name even though it is visible, because p belongs to the scope of a variable or type e_2 of the same name; e_2 is then said to obscure e_1 at p.

We illustrate these concepts by means of an example in Fig. 7. This example program consists of a single compilation unit belonging to package `p`. The compilation unit declares five classes: `Super`, `Outer`, `Inner` and two classes named `A`. In addition, it uses the primitive type `int`. The classes `Super` and `Outer` are top level classes, while `Inner` is a member class of `Outer`.

Class `Super` declares an instance field `f`, a member class `A`, an instance method `m`, and a static field `length`; these are referred to as its *local members*. Likewise,

```
173 package p;
174
175 class Super {
176   int f; /*①*/
177   class A { }
178   int m(int i) { return 42; }
179   static int length = 56;
180 }
181
182 class Outer {
183   int f; /*②*/
184   int x;
185   class A { }
186   class Inner extends Super {
187     int f; /*③*/
188     int y;
189     int m(int f /*④*/) {
190       A a1;
191       Outer.A a2;
192       p.Outer.A a3;
193       int[] Super = {};
194       return x + y
195            + f                   // →④
196            + this.f              // →③
197            + super.f             // →①
198            + Inner.this.f        // →③
199            + Inner.super.f       // →①
200            + Outer.this.f        // →②
201            + ((Super)this).f     // →①
202            + Super.length;
203     }
204   }
205 }
```

Figure 7: Example for name lookup in Java

Outer declares fields f and x, and two classes A and Inner. The latter class itself declares two fields f and y, as well as a method m. In addition to its local members, Inner also inherits the member class A from Super; thus, the scope of the class A declared on line 177 includes the bodies of both Super and Inner.

Class Inner does not inherit field Super.f, since the locally declared field Inner.f hides it, and it does not inherit method Super.m, since the locally declared method Inner.m overrides it. Also note that the field f of class Outer is shadowed, and hence not visible, inside the body of Inner, even though that body is part of its scope.

Method m has a parameter f that shadows the field f of its host type Inner. The declarations of the local variables a1, a2, and a3 in method m demonstrate different kinds of type names. A type name can be a simple name, as in the declaration of a1, which refers to class A from Inner, not its shadowed namesake from Outer. To refer to the latter type, we have to qualify it with the name of its enclosing type (line 191), which may itself be qualified by the name of its package (line 192).

Lines 194–202 show examples of qualified variable and method references. Line 194 refers to variables x and y by their simple names, which is possible since they are visible. This would still work if y were declared in class Super, or x in a superclass of Outer, but not if y were declared in an enclosing class of Super. Parameter f of m is also visible, and can thus be accessed by its simple name at line 195, as indicated by the comment.

The following lines show different ways of qualified field access expressions. Field f of class Inner, which is shadowed by the parameter f, can be referred to by qualifying with this (line 196). The field f from Super, although hidden by the field f in Inner, is accessible through a qualification with super (line 197). We can access the same two fields through qualification with Inner.this (line 198) and Inner.super (line 199), although such qualified this accesses are more usually employed to access shadowed fields of enclosing classes, as with the reference Outer.this.f (line 200). Note that for fields, the access super.f is equivalent to ((Super)this).f (line 201), except that it has slightly more relaxed accessibility rules [7, §6.6.2].

Line 202 shows an example of obscuring: in the expression Super.length, name Super could either refer to a type or to a variable (though not to a package). Since this expression occurs within the scope of the local variable Super declared on line 193, the latter interpretation is chosen; at runtime, this expression evaluates to the length of the array referenced by Super, which is 0, and not to the value stored in the static field length of class Super. To refer to the latter field, we would have to use p.Super.length instead.

One feature of Java name lookup that we have not illustrated in this example is method *overloading* [7, §8.4.9]: at any given program point, several different candidate methods with the same name but different signatures may be in scope; to determine which method declaration an invocation refers to, the number and types of actual arguments are matched against the signatures of the candidate methods, and the closest match is chosen. If a unique closest match does not exist, the program is rejected with a compile-time error. The same process is also used to determine which constructor a class instance creation expression (i.e., a new expression) or explicit constructor invocation [7, §8.8.7.1] refers to.

In the following, we will use the (non-standard) term *reference* to cover package names, type names, field access expressions, expression names (i.e., names referring to variables), method invocations, class instance creation expressions, and explicit constructor invocations. It will also be convenient to consider constructors as declared entities, although the JLS does not do so.

3.2 Modular Specification of Name Lookup

Although the JLS defines name lookup in a global, static manner in terms of declaration scopes and their nesting, it is possible to give a more local, modular specification of name lookup that determines what declaration a reference binds to by considering its location within the program. For the purposes of inverting lookup to obtain a reference construction algorithm, this algorithmic style is more convenient. We will hence briefly outline its implementation in the JastAddJ Java compiler [3, 2].

JastAddJ is implemented in JastAdd [4], an extension of Java with attribute grammar features. Programs are represented by their abstract syntax trees (AST), and analyses are implemented as parameterized attributes on the nodes of the AST. Name lookup is mostly handled by a

```
210 eq TypeDecl.getBodyDecl(int i).
211     lookupVar(String name) {
212   Variable res = memberField(name);
213   if(res != null)
214     return res;
215   res = lookupVariable(name);
216   if(res != null)
217     if((inStaticContext() || isStatic())
218       && res.isInstanceVariable())
219       return null;
220   return res;
221 }
```

Figure 8: Variable lookup from inside a type declaration

trio of attributes for looking up types, variables and methods by their simple name, which are declared in JastAdd as follows:

```
inh TypeDecl ASTNode.lookupType(String n);
inh Variable ASTNode.lookupVar(String n);
inh Set<MethodDecl>
              ASTNode.lookupMeth(String n);
```

The first declaration declares an attribute `lookupType` that is defined on every AST node, as indicated by the receiver type `ASTNode`, and is parameterized by the name of the type to look up, which is a (Java) string. When evaluated on a node p with a name n as its argument, the attribute yields a `TypeDecl`, which is itself a node representing the declaration that type name n binds to at p.

Similarly, `lookupVar` is an attribute computing the variable declaration (which may declare either a field, a local variable or a parameter) that a given name refers to if interpreted as an expression name. Attribute `lookupMeth` returns not a single method, but a whole set of candidate methods that a method name may refer to, from which one is selected by overloading resolution.

The keyword `inh` appearing in all three declarations indicates that these are *inherited attributes*, meaning that they are defined by equations matching on the syntactic context of the node on which they are defined.

A typical example of an equation for `lookupVar`, slightly simplified for presentation purposes, is given in Fig. 8.

The equation is of the form

```
eq TypeDecl.getBodyDecl(int i).
     lookupVar(String name) { ... }
```

indicating that it defines the value of attribute `lookupVar` on any `BodyDecl` node that is the $\mathtt{i}^{\text{th}}$ child of a `TypeDecl` node: such a node represents a declaration or initializer block appearing in the body of a class or interface type declaration.[3]

The attribute computation itself is given as a regular Java method body, which is executed with `this` bound to the *parent* node, in this case the type declaration, and not the child node (i.e., the body declaration).

To determine the variable declaration that a simple expression name n refers to at the program point given by a body declaration node inside a type t, the following computation is performed (see Fig. 8):

- Attribute `memberField` is invoked on line 212 to look up n as a member field of t; if a member field named n is found, its declaration is returned (line 214).
- Otherwise, `lookupVar` is recursively invoked on t itself in line 215 to search enclosing scopes. This yields a lexical scoping discipline where inner classes can see member fields of enclosing classes. The test in line 213 prevents recursion if a member field of the same name exists, implementing shadowing.
- Finally, the result of the recursive invocation is filtered in line 217: if t is itself declared as static or occurs in a static context, instance variables cannot be accessed inside t [7, §6.5.6.1].

Other equations for `lookupVar` implement lookup of local variables and parameters inside methods, and of statically imported fields in a similar manner.

The most important auxiliary attribute used in the definition of `lookupVar` is `memberField`, whose implementation we show in Fig. 9. In contrast to `lookupVar`, `memberField` is a *synthesized attribute*, as indicated by the JastAdd keyword `syn`, meaning that the attribute is computed on the node itself as opposed to its parent node.

We show the definition of `memberField` for class types only, the definition for interface types is very similar: first, the given name is looked up among the locally declared fields using attribute `localField` (line 226),

[3] We refer to the literature for a more detailed discussion of the syntax of JastAdd attribute definitions [3].

```
224 syn Variable ClassDecl.memberField
225                                (String name) {
226   Variable f = localField(name);
227   if(f != null)
228     return field;
229   if(hasSuperclass()) {
230     f = superclass().memberField(name);
231     if(f != null)
232       if(f.isPrivate() ||
233            !f.accessibleFrom(this))
234         return null;
235       return f;
236   }
237   // search through interfaces omitted
238   return null;
239 }
240
241 syn Variable ClassDecl.localField
242                                (String name) {
243   for(BodyDecl bd : getBodyDecls())
244     if(bd instanceof FieldDecl) {
245       FieldDecl fd = (FieldDecl)bd;
246       if(name.equals(fd.getName()))
247         return fd;
248     }
249   return null;
250 }
```

Figure 9: Member field lookup

which simply iterates over all body declarations of the class looking for a field declaration with the appropriate name. If such a field is found, it is returned as the result of the lookup (line 228). Otherwise, `memberField` is invoked recursively on the superclass, if there is one,[4] (line 230) and on all superinterfaces (omitted from the figure). This implements inheritance, with line 233 filtering out members that lack sufficient accessibility. Hiding is implemented by aborting the search for inherited fields when a local field of the same name is found.

The defining equations for `lookupType` and `lookupMeth` are similar to what we have shown for `lookupVar`, using the same recursion patterns to implement lexical scoping with shadowing and inheritance with hiding, and additional filtering steps to account for accessibility rules and static members.

In JastAddJ there is no single attribute implementing a lookup function for resolving an arbitrary reference at an arbitrary program point. Instead, a solution based on AST rewriting is adopted, which is somewhat subtle and not well-suited for our purposes, since the algorithm is distributed over several attributes and rewrite rules; for details see [2].

In order to be able to reason at least informally about the correctness of the reference construction algorithm to be derived in the remainder of the section, we distill a composite algorithm for looking up arbitrary references that incorporates syntactic classification and disambiguation to handle obscuring.

A somewhat simplified version of this algorithm for resolving package, type and variable references (but not method or constructor invocations) is shown in Fig. 10. We assume that simple names are represented by AST nodes of type `SimpleName`, and qualified names (including field access expressions) by nodes of type `Dot`, and both types implement interface `Reference`. The attributes `SimpleName.lookupAt` and `Dot.lookupAt` look up, respectively, a simple name and a qualified name at a program point represented by a node `nd`.

Crucial to both is the attribute `nameKind` which determines what kind of name is expected at a given AST node. This can be `PACKAGE_NAME` (indicating that this node must be a package name), `TYPE_NAME`

[4]Note that only class `java.lang.Object` has no superclass.

```
251 syn Decl ASTNode.lookup(Reference r) {
252   return r.lookupAt(this);
253 }
254
255 syn Decl SimpleName.lookupAt(ASTNode nd)
256 {
257   String n = this.getName();
258   switch(nd.nameKind()) {
259   case EXPR_NAME:
260     return n.lookupVar(n);
261   case AMBIGUOUS_NAME:
262     Decl res = n.lookupVar(n);
263     if(res != null)
264       return res;
265     res = n.lookupType(n);
266     if(res != null)
267       return res;
268     return n.lookupPackage(n);
269   // other cases elided
270   }
271 }
272
273 syn Decl Dot.lookupAt(ASTNode nd) {
274   Expr l = getLeft();
275   String n = getRight().getName();
276   switch(nd.nameKind()) {
277   case EXPR_NAME:
278     return l.type().memberField(n);
279   case AMBIGUOUS_NAME:
280     Decl d = nd.lookup((Reference)l);
281     if(d instanceof PkgDecl) {
282       PkgDecl p = (PkgDecl)d;
283       Decl res = p.memberType(n);
284       if(res != null)
285         return res;
286       return p.subPackage(n);
287     } else if(d instanceof TypeDecl) {
288       TypeDecl t = (TypeDecl)d;
289       Decl res = t.memberField(n);
290       if(res != null)
291         return res;
292       return t.memberType(n);
293     } else {
294       return l.type().memberField(n);
295     }
296   // other cases elided
297   }
298 }
```

Figure 10: Lookup of general references (simplified)

(for type names), `EXPRESSION_NAME` (for a name referring to a variable), or one of the ambiguous kinds `PACKAGE_OR_TYPE_NAME` and `AMBIGUOUS_NAME`, the latter indicating that nothing at all can be said about the expected kind of name. We do not detail the implementation of this attribute as it is provided by JastAddJ and follows closely the rules described in the JLS [7, §6.5.1].

To resolve a simple name, we compute the name kind of the node at which it is looked up, and then dispatch to the appropriate simple lookup attribute; we only show the code for kind `EXPR_NAME` and for `AMBIGUOUS_NAME`, which is the most complicated case. For instance, the simple name `Super` on line 202 in Fig. 7 has name kind `AMBIGUOUS_NAME`, hence it is first looked up as a variable; since this lookup yields a result, no type or package lookup is attempted.

To resolve a qualified name, we first extract the qualifying expression `l`, which may either be another name or a more general expression such as a qualified `this` or `super`,[5] and the name `n` to the right of the dot.

Again the name kind is consulted to determine what kind of lookup to perform. If it is an expression name, the name is looked up as a field of the type of the left hand side expression. For simplicity, we have elided the definition of attribute `type`. If the name is ambiguous, the expression on the left hand side must itself be a name, so we look it up recursively. If the result is a package declaration, we first try to look up `n` as a type within that package; failing this, it must refer to a subpackage. If, on the other hand, `l` refers to a type, `n` is looked up as a member field of that type, or as a member type if there is no such field.

The full version of `lookupAt` also checks that accessibility rules are respected (see Section 4) and that non-static members are not accessed inside a static context, and performs overloading resolution for methods and constructors.

3.3 Inverting Variable Lookup

We now describe how name lookup rules such as the ones just presented can be inverted to yield reference construction rules.

[5] Note that JastAddJ considers `super` an expression; this is a simplification, but deviates from the JLS.

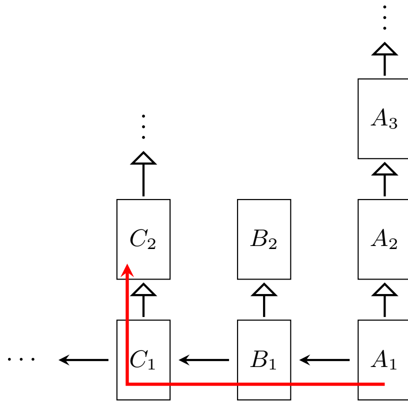

Figure 11: Schematic illustration of field lookup

Ideally, we would like to invert every attribute in isolation, for instance defining an attribute `accessVar` that is right inverse to `lookupVar` in the sense of Equation 1. But such an `accessVar` attribute would be a rather poor reference construction algorithm: since its return type would have to be `String`, it would return the name of the variable to refer to if that variable is visible, or `null` otherwise. In particular, a RENAME refactoring based on this algorithm would never be able to add a qualification to evade shadowing as shown in the example of Fig. 1.

Another possibility would be for `accessVar` to directly construct a `Reference`, possibly including qualifications. However, its correctness would then have to be argued for with respect to the general lookup function `lookup`, not only `lookupVar`, destroying the symmetry between lookup and reference construction.

Instead, we opt for a middle way: reference construction attributes such as `accessVar` construct an *abstract reference*, which contains enough information to build an actual reference, and we carefully formulate individual correctness properties relating these attributes to their corresponding lookup attributes. In a second step, the abstract references are converted into actual references, with the individual correctness properties ensuring that the constructed reference satisfies Equation 1.

To motivate the concept of an abstract reference, consider the lookup algorithm for fields presented above in Fig. 9. In general, this lookup proceeds in an "outward and upward" motion as illustrated in Fig. 11: starting from inside some class A_1, `memberField` first traverses A_1 and its superclasses A_2, A_3 and so on. If the field is not found anywhere, `lookupVar` is evaluated recursively on the class B_1 enclosing A_1, searching through the superclasses of B_1 in turn. The field is ultimately found in a type C_2, which is a supertype of a type C_1 enclosing A_1.

```
299 class AbstractVarRef {
300   String name;
301   boolean visible;
302   TypeDecl source, bend;
303   // standard constructor elided
304 }
```

Figure 12: Abstract references

The path from the point of reference to the declaration can be visualized as an outwards motion through enclosing classes until reaching a "bend" at C_1, and then proceeding upwards the inheritance hierarchy until reaching the "source" C_2. Consequently, we will refer to C_1 as the *bend* type and to C_2 as the *source* type of this field lookup. We do not require the target field to be a local member of C_2, it may just as well be inherited from its supertype C_3.

If the field is visible in A_1, i.e. there are no shadowing or hiding fields in $A_1, A_2, A_3, \ldots, B_1, B_2, C_1$, it can be referred to by its simple name, say `x`. However, even if it is not directly visible, it can still be referred to using the qualified field access `((`C_2`)`C_1`.this).x`. As discussed below, this access can be simplified depending on the inheritance and nesting relationship of C_1, C_2 and A_1.

This suggests that in order to construct a qualified reference to a target field f from some program point p, it suffices to know the source class, the bend class, the name of f and whether it is visible at p. These pieces of information are encapsulated into a class `AbstractVarRef` as shown shown in Fig. 12.

We will now show that the name lookup equations of the previous subsection can be systematically inverted to compute such abstract references.

We start by considering the counterpart to the lookup function `localField`, `accLocalField`, which is shown at the bottom of Fig. 13. Instead of iterating over the body declarations of a class looking for a field of a given name, it looks for the given field itself, and returns

```
305 eq TypeDecl.getBodyDecl(int i).
306     accessVar(Variable v)
307 {
308  AbstractVarRef r = accMemberField(v);
309  if(r != null)
310   return r;
311  r = accessVar(v);
312  if(r != null) {
313   if((inStaticContext() || isStatic())
314       && v.isInstanceVariable())
315    return null;
316   if(memberField(v.getName()) != null)
317    r.visible = false;
318  }
319  return r;
320 }
321
322 eq ClassDecl.accMemberField
323     (Variable v)
324 {
325  AbstractVarRef r = accLocalField(name);
326  if(r != null)
327   return r;
328  if(hasSuperclass()) {
329   r = superclass().accMemberField(name);
330   if(r != null) {
331    if(v.isPrivate() ||
332       !v.accessibleFrom(this))
333     return null;
334    if(r.visible &&
335       localField(name) == null)
336     r.source = this;
337    else
338     r.visible = false;
339    r.bend = this;
340    return r;
341   }
342  }
343  return null;
344 }
345
346 eq ClassDecl.accLocalField(Variable v)
347 {
348  for(BodyDecl bd : getBodyDecls())
349   if(bd == v)
350    return new AbstractVarRef(v.getName(),
351                          true, this, this);
352  return null;
353 }
```

Figure 13: Reference construction

an abstract reference, recording the name of the field; both source and bend are equal to the declaring class, and the field is directly visible.

The correctness of this function with respect to `localField` can be expressed by the following property, which is easily seen to hold (remembering that in Java a class cannot declare two fields of the same name):

Property 1. *For any class* `c` *and field declaration* `f`, *if* `c.accLocalField(f)` *evaluates to a reference* `r` *then* `r.bend` = `r.source` = `c`, `r.visible` *is true, and* `c.localField(r.name)` = `f`.

Attribute `accMemberField` shown in the same figure corresponds to `memberField`. Paralleling the control structure of the latter, it first invokes `accLocalField` to try and construct a reference to the target variable `v` as a locally declared field. If this fails, it recursively invokes itself on the superclass (and superinterfaces). Abstract references returned from these recursive invocations have to be adjusted to update information about visibility and the source and bend types as shown in lines 334–339.

These adjustments ensure that the following property holds (taking Property 1 above into account):

Property 2. *For any class* `c` *and field declaration* `f`, *if* `c.accMemberField(f)` *evaluates to reference* `r`, *then*

1. `c` = `r.bend` *is a subclass of* `r.source`; *if* `r.visible` *then* `r.bend` = `r.source`;
2. `r.source.memberField(r.name)` = `f`.

Finally, attribute `accessVar`, of which one equation is shown at the top of Fig. 13, iterates over enclosing classes in the same way as `localVar`, and is inverse to it in the following sense:

Property 3. *For any node* `nd` *and variable declaration* `v`, *if* `nd.accessVar(v)` *evaluates to a reference* `r`, *then*

1. `r.bend` *encloses* `nd`; *it is a subclass of* `r.source`;
2. *if* `r.visible` *then* `r.bend` = `r.source` *and* `nd.lookupVar(r.name)` = `v`.

The other equations of `lookupVar` can all be inverted in a similar fashion to yield corresponding reference construction attributes. It remains to discuss the algorithm for converting an abstract reference to an actual reference node, which is outlined in Fig. 14. The name kind of the node at which the reference node will eventually be inserted needs to be checked to ensure that a variable reference is allowed at this place. If this is the case and the abstract reference indicates that the variable is visible, a simple `SimpleName` node suffices: from Fig. 10 and Property 3 above it is easy to see that this reference will be resolved as intended.

Otherwise, a qualified field access has to be constructed. We only show two cases. In the simplest case, both source and bend are equal to the enclosing class `T`, i.e., the variable to refer to is a field of `T`; in this case, a `this`-qualified access should be constructed. Otherwise, we construct a fully qualified access explicitly referring to both source and bend using locked type bindings $\uparrow S$ and $\uparrow B$. Hence, eliminating one locked binding may introduce new locked bindings that have to be eliminated in turn.

There are several other opportunities for constructing simpler qualified accesses, which we have elided for simplicity. We have also omitted additional checks for accessibility and references to static members; these tests are precisely the same as those performed during lookup, and can hence be taken over directly from JastAddJ.

```
354 Reference ASTNode.mkRef(AbstractVarRef r)
355 {
356   if(nameKind() == EXPR_NAME ||
357      nameKind() == AMBIGUOUS_NAME) {
358     SimpleName n= new SimpleName(r.name);
359     if(r.visible)
360       return n;
361     TypeDecl T = enclosingType(),
362              S = r.source, B = r.bend;
363     if(S == B && B == T)
364       // return access this.n
365     else
366       // return access ((↑S)↑B.this).n
367   }
368   return null;
369 }
```

Figure 14: Skeleton of algorithm for constructing an actual reference from an abstract one

3.4 Inverting Type and Method Lookup

Since the lookup rules for types and methods are broadly similar to the variable lookup rules, corresponding reference construction rules can be obtained in the same way.

When constructing an actual type reference from an abstract type reference, care has to be taken to avoid obscuring: even if the abstract reference indicates that the type is visible, it must additionally be checked if an obscuring variable is in scope; if so, the type name must be qualified by either its package (for top level types) or its enclosing type (for member types). These checks can be carried out using the `lookupVar` and `nameKind` attributes of JastAddJ.

Abstract method references additionally track information about other methods with the same name as the target method. When constructing an actual reference, the overloading resolution machinery of JastAddJ is used to check whether any of these methods would take precedence over the target method; if so, additional type casts are inserted on the method arguments to ensure the desired method is selected.

Note that locked bindings do not by themselves prevent changes in dynamic dispatch behavior. For instance, Fig. 15 shows two programs that have the same (static) binding structure, yet different dynamic dispatch behavior: while in the program of Fig. 15(a) method `B.m` overrides method `A.m`, the renamed method `B.n` of Fig. 15(b) no longer does. Hence, the method invocation on line 376 returns `42`, while its counterpart on line 392 returns `23`, although both of them bind to method `A.m`.

In J_L, we treat method overriding by the mechanism of *explicit overriding* (see Section 5): every method is annotated with the set of methods it (directly) overrides; just as for locked names, this annotation does not change unless the refactoring explicitly alters it. When translating back to Java, we use accessibility adjustments, discussed in the next section, to enforce or prevent overriding where necessary, and abort the refactoring if this cannot be done.

3.5 Summary

To unlock locked bindings, it is necessary to construct a possibly qualified reference that binds to a given vari-

```
370 class A {
371  int m() {
372   return 23;
373  }
374  int f() {
375   A a = new B();
376   return a.m();
377  }
378 }
379
380 class B extends A
381 {
382  int m() {
383   return 42;
384  }
385 }
```

(a)

```
386 class A {
387  int m() {
388   return 23;
389  }
390  int f() {
391   A a = new B();
392   return a.m();
393  }
394 }
395
396 class B extends A
397 {
398  int n() {
399   return 42;
400  }
401 }
```

(b)

Figure 15: Example of a change in dynamic dispatch in spite of same binding structure

able, type or method from a particular program point. We have shown that it is possible to derive an implementation of such a reference construction algorithm from a name lookup algorithm. The two algorithms exhibit a very fine-grained correspondence, with every lookup rule paralleled by a reference construction rule, ensuring that no corner case of the lookup rules is overlooked when implementing reference construction. While we have only shown a handful of representative rules, the construction scales to the full Java language: in Section 6 below we will report on an implementation of reference construction that handles all name lookup features of Java 5.

Since lookup rules take access control into account, so does reference construction: if a declaration is not accessible at a program point, the algorithm will detect this and fail to produce a reference. In the next section, we will take a closer look at how to adjust accessibilities to ensure references are accessible wherever needed.

4 Accessibility Constraints

In this section, we consider the role of *access modifiers* in refactoring. In particular, we observe that access modifiers do not only serve to control access to declared entities, but also have an effect on inheritance, overriding, hiding, and subtyping. Because all these effects depend not only on access modifiers, but also on the relative locations of the involved declared entities and references, any refactoring that changes locations must consider access modifiers. As we will see, the locking of bindings as introduced in the previous section is insufficient to control the many forces on access modification; instead, a constraint-based approach will be needed.

4.1 Access Modifiers and Accessibility in Java

Access modifiers such as `private` or `public` let the programmer exert control over accessibility[6] of types and their members from different parts of a program. To determine which access modifier is sufficient to access an entity depends on the location of the declaration of the accessed entity in the program code, and on the location of the accessing reference. For instance, `public` accessibility is required to access a top level type, unless the type and the accessing reference reside in the same package, in which case `package` accessibility suffices.[7] The example of Fig. 3 showcased how this rule affects refactoring: Moving a class with `package` accessibility from one package to another renders it inaccessible for references from its former package, thereby necessitating an increase of declared accessibility to `public`.

For the access of type members the situation is slightly more differentiated:

- If the accessed member and the accessing reference reside in the same top level type, `private` is generally sufficient.
- Else, if the accessed entity and the reference reside in the same package, at least `package` accessibility is required.
- Else, if the accessing reference resides in a subclass

[6] *Accessibility* is not to be confused with *visibility*, introduced in the previous section.

[7] Note that Java has no keyword for `package` accessibility; instead, top level types and all members and constructors of classes are `package` accessible unless an explicit access modifier is specified. For this reason, `package` accessibility is often referred to as `default` accessibility, but this is misleading: interface members, for instance, are `public` by default, and enumeration constructors are `private`.

of the class in which the accessed entity is declared, at least `protected` accessibility is required.

- Else, `public` accessibility is required.

Figure 16 illustrates some of these accessibility rules. For instance, the private method `A.n` can be accessed from inside an inner class of `A` at line 412, while the package accessible method `A.p` cannot be accessed from a different package on line 425. However, `protected` accessibility suffices to access method `A.q` from within the body of `B`, which is a subtype of `A`, at line 429, even though this reference is in a different package and appears not in `B` itself but in an inner class.

The above rules are merely a short summary; the full rules are considerably more involved, and will be presented in detail in Section 4.5.

4.2 Other Effects of Access Modifiers in Java

Accessibility and inheritance In Java, access modifiers do not only govern access, they also contribute to inheritance:

- If a member is to be inherited at all, its accessibility must be greater than `private`.
- If a member is to be inherited by a subclasses declared in a different package, its accessibility must be greater than `package`.

Note that, in Java, members can only be inherited from immediate supertypes; if they are, they become members of the inheriting type, and can then be further inherited by immediate subtypes of *that* type and so on [7, §8.2]. This means that if type B is a subtype of type A in a different package, and type C is in turn a subtype of type B, then C does not inherit a `package` accessible member from A, even if C and A are in the same package.

Figure 16 has examples of this: the `package` accessible method `A.p` is not inherited by subtypes in different packages (line 424), and also not by subtypes in the same package (line 437), if there is an intervening type (here `B`) from a different package. The private method `A.n` is not inherited at all, not even by an inner type of its declaring type (line 411).

Accessibility and overriding Although one might expect the two notions to be closely connected, the rules for overriding in Java differ from those of inheritance in that it is possible for a type to override a method it would not inherit otherwise.

For instance, as shown in Fig. 16, the method `A.o` declared with `package` accessibility can be overridden in the same package (line 436) even though it would not be inherited otherwise (just as `A.p` is not inherited; cf. above). On the other hand, just like for inheritance, `A.o` is not overridden in line 423, which is located in a different package, and the private method `A.m` cannot be overridden anywhere, not even in the scope of the same type (line 410).

Altogether, the requirements for overriding are as follows:

- For a method to be overridden by another one declared in the same package, `package` accessibility suffices.
- For a method to be overridden by another one declared in a different package, `protected` accessibility suffices.
- Overriding is transitive, i.e., a method overriding another method also overrides all methods the other method overrides [7, §8.4.8.1].

The first two points mean that a method can override two instance methods none of which overrides the other. The last point means that a method m_1 can indirectly override a `package` accessible method m_2 in a different package, namely if an interjacent subtype in that package overrides it with `protected` accessibility.

Note that whether one method overrides another has semantic implications, since overriding is a prerequisite for dynamic binding. For this reason, the return type of the overriding method must be a subtype of that of the overridden method, and their `throws` clauses must be compatible. It is also not permitted to override a static method with an instance method [7, §8.4.8.1].

Accessibility and hiding Method hiding is primarily a problem of name resolution and therefore can be dealt with by locking bindings as shown in Section 3. However, as with overriding it is an error for a static method to hide an instance method [7, §8.4.8.2]. Since the definition of

```
402 package p;
403 public class A {
404   private void m() {}
405   private void n() {}
406   void o() {}
407   void p() {}
408   protected void q() {}
409   class C extends A {
410     @Override void m() {  // ✘ cannot override
411       this.n();           // ✘ not inherited
412       A.this.n();         // ✔ but accessible
413     }
414     @Override void o() {  // ✔ can override
415       this.p();           // ✔ inherited
416       A.this.p();         // ✔ accessible
417     }
418   }
419 }
420
421 package q;
422 abstract public class B extends p.A {
423   @Override void o() {    // ✘ cannot override
424     this.p();             // ✘ not inherited
425     ((p.A)this).p();      // ✘ not accessible
426   }
427   class C {
428     @Override q() {       // ✘ cannot override
429       B.this.q();         // ✔ but accessible
430     }
431   }
432 }
433
434 package p;
435 public class C extends q.B {
436   @Override void o() {    // ✔ can override
437     this.p();             // ✘ not inherited
438     ((A)this).p();        // ✔ but accessible
439   }
440 }
```

Figure 16: Meaning of access modifiers ("accessibility") for member access, inheritance, and overriding

hiding hinges on accessibility (the hidden member must be accessible from where the hiding occurs [7, §8.4.8.2]), care must be taken that an increase of accessibility of an instance method necessitated by some other condition does not lead to illegal hiding by a static method. For instance, in the simple program

```
class A { private void m() {} }
class B extends A { static void m() {} }
```

accessibility of `A.m` must not be increased, since otherwise the declaration of `B.m` causes a compile error.

Accessibility and subtyping Last but not least, accessibility interacts with typing: Because subtyping demands that instances of a subtype have accessible what is declared accessible by the supertype, accessibility of instance methods overridden in subtypes must not decrease.[8] For instance, in the example of Fig. 4, the increase of accessibility of `A.m` required to preserve the overriding of `B.m` had to be complemented by an increase of accessibility of `B.m`, but not to maintain overriding, but to respect subtyping. Interestingly, Java extends this rule to static methods, but not to fields.

4.3 Accessibility and Refactoring

It is obvious that due to its dependence on location, accessibility plays a central role in all refactorings that move program elements, including MOVE CLASS, MOVE MEMBER, PULL UP MEMBER, and PUSH DOWN MEMBER [5]. In addition, accessibility needs to be considered for type-related refactorings, including type generalization refactorings (such as GENERALIZE DECLARED TYPE, USE SUPERTYPE WHERE POSSIBLE, EXTRACT INTERFACE, and EXTRACT SUPERCLASS [28]), which require that the supertypes and their members are accessible to the clients of the generalized type), as well as refactorings changing the type hierarchy (such as and INFER TYPE [22] and REPLACE INHERITANCE WITH DELEGATION, the latter of which removes a subclass relationship and thus may render protected entities inaccessible from the former subclass [10]).

The unlocking algorithm of Section 3 also has to deal with accessibility in order to avoid constructing a qual-

[8] Note that this rule allows redefinition of a `package` accessible instance method as `private` in a subtype, if that subtype belongs to a different package.

ified reference that violates accessibility rules. Last but not least, changing accessibility may be the immediate purpose of a refactoring, for instance to achieve data encapsulation by making fields private (as is done by the ENCAPSULATE FIELD refactoring [5]). In all these refactorings, failure to adjust access modifiers, or incorrect adjustment of access modifiers, may lead to non-compiling programs or, worse still, to silent change of behavior. Like with the naming problems dealt with in the previous section, the solution is to record all relationships before the refactoring, and to compute additional changes required to make sure that they still hold afterwards. The difference is that for accessibility, the additional changes pertain to declarations (adaptation of access modifiers) rather than references.

As it turns out, this difference necessitates a wholly different approach. While adjusting references cannot interfere with any other part of the program, accessibilities are necessarily adjusted at the declaration site, and thus may cause new problems with other references and declarations. In particular, since access modifiers also influence inheritance and overriding and are further constrained by subtyping and hiding as outlined above, finding an access modifier that preserves all relationships involved in compilability and the bindings of a program (both static and dynamic) is basically a search problem.

4.4 Computation of Required Accessibility as a Constraint Satisfaction Problem

From a refactoring perspective, changing the accessibility of a declared entity is somewhat analogous to changing its type: like the new type, the new accessibility must not only suit all references to the entity, but must also harmonize with the accessibilities of other entities it is related to, which in turn must suit all of their references and so on. This analogy suggests viewing accessibility refactoring as a constraint satisfaction problem, as has been done before for type refactoring [28]. The main difference is that the variables in the constraint system represent access modifiers, rather than type annotations, of declared entities.

The constraints required for a constraint-based refactoring are usually generated by applying so-called *constraint rules* to the program to be refactored (see, e.g., [28, 25, 24]). Such a constraint rule is generally of the form

$$\frac{\textit{program query}}{\textit{constraints}}\ (\text{RULENAME})$$

where *program query* stands for an expression selecting those elements of the program to which the rule is to apply, while *constraints* represents a set of constraints expressing relationships between those properties of the selected program elements that are to be constrained by the rule.

For instance, the rule

$$\frac{\textit{interface-member}(m)}{\langle m \rangle = \texttt{public}}\ (\text{IMEMBER})$$

expresses that the accessibility of an interface member m, represented by the constraint variable $\langle m \rangle$, must be `public`. When applied to the program

```
interface I { void m(); }
class C implements I{ public void m(){} }
```

it generates the constraint $\langle \texttt{I.m} \rangle = \texttt{public}$, preventing any changes to the accessibility of `I.m`. Applying the subtyping rule

$$\frac{\textit{overrides}(m_2, m_1) \vee \textit{hides}(m_2, m_1)}{\langle m_2 \rangle \geq_A \langle m_1 \rangle}\ (\text{SUB})$$

to the same program generates the additional constraint $\langle \texttt{C.m} \rangle \geq_A \langle \texttt{I.m} \rangle$ expressing that the declared accessibility of `C.m` must be greater or equal ($\geq_A$) than that of `I.m`; together, the two constraints prevent any lowering of the accessibility of `C.m`.

Since both queries and constraints are relations, they can be exchanged for each other to a certain extent. In fact, as has been noted elsewhere [24, 23], the main difference between a query and a constraint is *when* it is evaluated: While queries are evaluated during constraint generation (and hence entirely based on the old program), constraints are evaluated during constraint solving, when some of the constraint variables have been given new values to reflect the intended changes, and when new values are being computed for others. Therefore, the (hypothetical) rule

Domains:

P	the packages of the program *Prog* to be refactored
T	the reference types of *Prog*
$T_A \subseteq T$	the access modifiable types in *Prog*, i.e., T excluding local and anonymous types
M	the (local) members and constructors of the types in T
$T_{top} := T \setminus M$	the top level types in *Prog*
$D := T_A \cup M$	the access modifiable declared entities in *Prog*
R	the references to members of D in *Prog*
A	the set of access modifiers of Java; $A = \{\texttt{private}, \texttt{package}, \texttt{protected}, \texttt{public}\}$

Orderings:

$\leq_T \subseteq T \times T$	the (reflexive, transitive) subtype relation of $Prog$
$\leq_N \subseteq T \times T$	the (reflexive, transitive) type nesting relation of $Prog$
$<_A \subseteq A \times A$	the total ordering of access modifiers: $\texttt{private} <_A \texttt{package} <_A \texttt{protected} <_A \texttt{public}$

Location functions:

$\pi : D \cup R \to P$	$\pi(e)$ is the package in which e is located
$\tau : D \cup R \rightharpoonup T$	for $m \in M$, $\tau(m)$ is type of which m is a local member; for $r \in R$, $\tau(r)$ is the innermost type enclosing r, if any
$\tau_{top} : D \cup R \rightharpoonup T_{top}$	$\tau_{top}(e)$ is the $t \in T_{top}$ with $\tau(e) \leq_N t$; only defined when $\tau(e)$ is defined

Accessibility functions:

$$\alpha : (R \cup M) \times D \to A \quad \alpha(e,d) := \begin{cases} \texttt{private} & \text{if } \tau_{top}(e) = \tau_{top}(d) \\ \texttt{package} & \text{else, if } \pi(e) = \pi(d) \\ \texttt{protected} & \text{else, if } \exists t \in T : \tau(e) \leq_N t <_T \tau(d) \\ \texttt{public} & \text{else} \end{cases}$$

$$\iota : T \times T \to A \quad \iota(t_i, t_d) := \begin{cases} \texttt{package} & \text{if } \forall t \in T, t_i \leq_T T <_T t_d : \pi(t) = \pi(t_d) \\ \texttt{protected} & \text{else} \end{cases}$$

$$\omega : M \times M \to A \quad \omega(m_2, m_1) := \begin{cases} \texttt{package} & \text{if } \pi(m_2) = \pi(m_1) \\ \texttt{protected} & \text{else} \end{cases}$$

Figure 17: Definition of the functions α, ι, and ω determining the minimum required accessibility for access, inheritance, and overriding, respectively

$$\frac{\langle m \rangle = \texttt{public}}{\langle m \rangle = \texttt{public}}$$

is neither circular nor tautological: it just expresses that what was declared `public` before the refactoring must be declared `public` after (for instance to preserve the API of a program).

4.5 The Constraint Rules of Accessibility

As elaborated above, to determine which access modifier a declaration requires is not only constrained by the accesses of the declaration present in the program, but also by the existing (and non-existing) inheritance, overriding, hiding, and subtyping relationships. However, the impact of access modifiers on compilability and meaning is seldom spelled out explicitly in the JLS.

In this subsection, we will discuss some representative constraint rules in detail; a full listing of all rules is given in Appendix A.

To formulate queries and constraints, we make use of several basic relations and functions defined in Fig. 17: relations $\leq_T$ and $\leq_N$ model the program's inheritance hierarchy and type nesting structure, respectively, while functions π, τ and τ_{top} determine the package, immediately enclosing type and top level type in which a declaration or reference is located. Both τ and τ_{top} are undefined for top level types, which by definition do not have enclosing types, and for references that occur outside the body of a top level type declaration, for instance in an `extends` or `implements` clause.

The functions α (for *accessibility*), ι (for *inheritance*) and ω (for *overriding*) determine minimum accessibilities needed for access, inheritance and overriding, respectively. For a reference r and a declaration d, $\alpha(r, d)$ is the minimum accessibility needed for d to be accessible for r; its definition is basically a transcription of §6.6.1 in the JLS [7]. The first argument of α can also be a member m, in which case $\alpha(m, d)$ gives the minimum accessibility necessary for d to be accessible from where m is declared; this is needed to correctly model the definition of hiding.

For two types t_i and t_d, $\iota(t_i, t_d)$ is the minimum accessibility a member of t_d needs to have in order to be inherited by t_i: as discussed above, this is `package` if all types between t_d and t_i in the subtype hierarchy are in the same package, and `protected` otherwise. Finally, $\omega(m_2, m_1)$ is the minimum accessibility a method m_1 needs to have to be *directly* overridden by method m_2, which is `package` if m_1 and m_2 belong to the same package, and `protected` otherwise. Strictly speaking, this predicate should only be defined if the enclosing type of m_2 is a subtype of the one of m_1, but for convenience we define it for all methods.

To streamline the formulation of program queries, we will use additional query predicates, such as predicates *overrides* and *hides* introduced above. For now we informally explain the predicates when we use them; full definitions are given in Appendix A.

The first, and most fundamental, accessibility constraint rule for Java is the (ACC-1) rule:

$$\frac{\textit{binds}(r, d)}{\langle d \rangle \geq_A \alpha(r, d)} \text{ (ACC-1)}$$

Using the *binds* predicate to query the binding structure of the program, it states that whenever a reference r binds to a declared entity d, the accessibility of d must be no less than the minimum accessibility needed for d to be accessible at the position of r. As an illustration of this rule, consider the example of Fig. 3. On the original program, taking d to be the declaration of class `B` and r the reference in line 45, we see that $\alpha(r, d) = \texttt{package}$, so the constraint $\langle d \rangle \geq_A \texttt{package}$ is generated. On the refactored program, we have $\alpha(r, d) = \texttt{public}$, so the constraint is now $\langle d \rangle \geq_A \texttt{public}$, indicating that `B` must be declared `public`. Constraints generated by this rule also explain the compile error on line 425 of Fig. 16, whereas the constraints for lines 412, 416, 429 and 438 are satisfied.

A second, somewhat related constraint rule addresses the access of inherited members:

$$\frac{\textit{binds}(r, m) \quad \textit{receiver-type}(r, t) \quad t <_T \tau(m)}{\langle m \rangle \geq_A \iota(t, \tau(m))} \text{ (INHACC)}$$

The program query matches any reference r binding to a member m such that m is not locally declared in the receiver type t of r: such a member must be inherited, so its declared accessibility must be no less than the minimum accessibility required for t to actually inherit m, as computed by ι. This rule explains the accesses on lines 411, 415, 424, and 437 in Fig. 16.

A third constraint rule, preventing the loss of overriding exemplified in Fig. 4, is (OVRPRES):

$$\frac{\textit{overrides}(m_2, m_1)}{\langle m_1 \rangle \geq_A \omega(m_2, m_1)} \text{ (OVRPRES)}$$

Here, we use the predicate *overrides* to find all pairs of methods (m_2, m_1) such that m_2 directly overrides m_1. The generated constraint requires that m_1 has at least the minimum accessibility needed for the overriding to take place as computed by ω. While this rule only applies to direct overriding relationships, its comprehensive application to all methods in the program ensures that indirect (transitive) overriding is preserved as well.

As shown in the example at the end of Section 4.2, a static method m_2 may not hide an instance method m_1. Similarly, the return type of m_2 must be a subtype of the return type of m_1, and their `throws` clauses must be compatible [7, §8.4.8.3]. We define a predicate *may-hide* to check these conditions, and use it to define a constraint rule (HID) that lowers the accessibility of m_1 if necessary to prevent invalid hiding:

$$\frac{\begin{array}{c}\tau(m_2) <_T \tau(m_1) \quad \textit{static}(m_2) \\ \textit{override-equiv}(m_2, m_1) \quad \neg\textit{may-hide}(m_2, m_1)\end{array}}{\langle m_1 \rangle <_A \alpha(m_2, m_1)} \text{ (HID)}$$

Note that although this rule is about hiding, it does not use the query predicate *hides*. This is necessary since we are looking for a pair (m_2, m_1) of methods such that m_2 would hide m_1, were it not for the low accessibility of m_1. Using $\neg hides(m_2, m_1)$ as a query instead would produce all pairs of methods (m_2, m_1) such that m_2 does not hide m_1: this is true for many pairs of completely unrelated methods, for which this accessibility constraint would be unjustified. We will see this pattern frequently in the full listing of all constraint rules, as given in the Appendix A.

Also note that the definition of hiding, although relying on accessibility as expressed by α, is independent of any concrete reference, and thus uses the hiding method in place of a reference as argument; to cover this, the domain of the first argument of α in Fig. 17 is extended to include M, allowing it to address hypothetical accessibility as required by the definition of hiding [7, §8.4.8.2].

The constraint rule (SUB) ensuring the conditions of subtyping as required by the extension of the example of Fig. 4 in Section 2.2 has already been given in Section 4.4; for the case of hiding (rather than overriding) members, it is implicitly restricted to (static) methods, i.e., the rule does not apply to fields.

4.6 Summary

We have shown how the rules for accessibility in Java can be encoded as constraint rules. Based on the syntactic structure, type hierarchy, name bindings and overriding relationships of a program, these rules generate a set of constraints on the accessibilities of declarations that have to be satisfied in order to avoid compile errors and maintain dynamic dispatch behavior.

In the next section, we show how these constraints can be integrated with the binding unlocking algorithm of the previous section, yielding a comprehensive framework for maintaining and updating bindings.

5 J_L and Java

In this section, we give a more detailed presentation of J_L, our lookup-free, access control-free representation of Java programs, and present algorithms for converting between Java programs and their J_L representations.

5.1 Lookup-free, Access Control-free Representation of Java Programs

A J_L program is, syntactically speaking, almost a Java program, except for three differences:

1. Every declaration is annotated with a globally unique label. In example J_L code we write the label as a superscript on the declaration.

2. There are no simple names, instead there are locked bindings that directly refer to a declared entity by its label. In example J_L code we write a locked binding referring to a declaration labelled l as $\uparrow l$.

   While simple names are replaced by locked bindings, J_L programs can still contain qualified names as well as field access expressions and method invocation expressions, but instead of simple names they are composed of locked bindings.

3. Every instance method declaration in the program has an explicit overriding annotation of the form

$$\textbf{overrides}\ m_1, \ldots, m_n$$

where the m_i are locked bindings enumerating all the methods this method directly overrides.

We say that a J_L program P' *represents* a valid Java program P if the following three conditions are met:

1. P and P' are syntactically the same, except that overriding annotations are removed in P, declared accessibility levels of declarations may differ, locked bindings in P' are replaced with normal references in P, and method invocations in P may have additional upcasts.

2. P and P' have the same name binding structure, i.e., for every locked binding $\uparrow l$ in P' the corresponding reference in P resolves to the declaration labelled l in P' by the lookup rules of Java.

3. P and P' have the same overriding structure, i.e., method m_1 directly overrides method m_2 in P iff the `overrides` clause of m_1 in P' contains $\uparrow m_2$.

As an example, Fig. 18 shows the J_L version of the program in Fig. 5(a). We omit the `overrides` clauses since they are all empty. Although the names and declared accessibilities of declarations are unimportant in J_L, we retain them to allow reconstructing a Java program from its J_L encoding.

We will now present algorithms for translating Java programs to corresponding J_L programs and back.

```
446 class C^t1 {
447   private class B^t2 { }
448   void m^m1(↑t2 b^v1) { }
449   void n^m2() { }
450 }
451
452 interface J^t3 { class B^t4 { } }
453
454 class D^t5 extends ↑t1 implements ↑t3 {
455   D^c1(↑t1 c1^v2, ↑t5 d^v3) { ↑v2.↑m1(null); }
456   D^c2(↑t1 c2^v4, ↑t1 o^v5) {
457     ↑v4.↑m2();
458     ↑t5 d^v6 = new ↑c1(↑v4, null);
459   }
460   ↑t4 f^v7;
461 }
```

Figure 18: The J_L version of the program in Fig. 5(a)

5.2 Translating from Java to J_L and Back

Finding a J_L program to represent a given valid Java program is easy: assign unique labels to every declaration, resolve simple names by the standard lookup rules and replace them with locked bindings, and finally determine which methods every instance method (directly) overrides and add an `overrides` clause to its declaration.

Translating in the other direction is not quite as easy, but still fairly straightforward given the technical groundwork presented in the previous sections: accessibilities are adjusted to make declarations accessible anywhere they are referenced and to enforce or prevent overriding, and locked bindings are replaced by (possibly qualified) references. The explicit overriding declarations can then simply be removed.

Note, however, that binding unlocking and accessibility constraint solving influence each other, and hence have to be interleaved instead of being performed in sequence.

Binding unlocking may introduce qualifiers and upcasts that refer to inaccessible types. Therefore, an iterative approach is required: First, we generate and solve accessibility constraints to make sure that at every locked binding $\uparrow l$ the declaration labelled by l is indeed accessible. Then we unlock all locked bindings, which may generate new locked bindings, for which we again generate and solve accessibility constraints before unlocking them in turn, continuing until all locked bindings are gone.

This means, however, that during the translation the program may contain both locked bindings and normal references. Care must be taken when changing the accessibility of a declaration d in such a program, since this change might change the binding of already unlocked bindings. Clearly, such binding changes can only occur for references to a declaration with the same name as d. To avoid this issue, it is hence enough to additionally lock all bindings to declarations with the same name as d anywhere in the program before changing its accessibility.

Figure 19 shows our algorithm for translating from J_L to Java. While there are still locked bindings to eliminate,

1: **procedure** `Translate to Java` (Program p):
2: **while** p contains locked names **do**
3: $C \leftarrow$ accessibility constraints for p
4: **if** C is unsolvable **then**
5: **abort**
6: $S \leftarrow$ solution of C
7: **for all** $(d, a) \in S$ **do**
8: **for all** declarations d' with same name as d **do**
9: lock all references to d'
10: set accessibility of d to a
11: unlock all bindings in p
12: remove all explicit overriding declarations

Figure 19: Algorithm for translating from J_L to Java

the algorithm collects accessibility constraints and solves them, aborting if this is not possible. A solution consists of a set of pairs (d, a), where d is a declaration and a an accessibility, indicating that the declared accessibility of d has to be changed to a in order to satisfy the constraint system. When changing the accessibility, we lock potentially endangered names, and then unlock all names in the program.

Termination of the algorithm follows from the fact that starting from the second iteration of the loop only locked type and package names remain, since name unlocking never introduces other kinds of locked names. When unlocking these remaining names, locked bindings referring to their enclosing types or packages may be introduced. Thus, the number of loop iterations is bounded by the maximum depth of type and package nesting in the program.

As an example, consider the program of Fig. 20(b), which arises as the result of a J_L-level application of EXTRACT INTERFACE as detailed below. When computing accessibility constraints for this program, we find two unsatisfied constraints. First, since the member type t_2 is referenced on line 483 outside its declaring type, rule (ACC-1) generates the constraint $\langle t_2 \rangle \geq$ `package`, requiring t_2 to have at least package accessibility. With t_2 being `private`, this constraint is not fulfilled. Second, rule (SUB) generates the constraint $\langle m_1 \rangle \geq \langle m_3 \rangle$ requiring method m_1 to have at least the same level of accessibility as m_3, the method it overrides, which is not the case.

We can solve both constraints by removing the `private` qualifier from t_2's declaration, and making m_1 `public`. This change provides a solution to the whole system of accessibility constraints (not just the two shown here), and makes it possible to eliminate locked bindings. The name unlocking algorithm takes care of inserting necessary qualifications and casts, yielding the program previously shown in Fig. 5(b).

This small example shows that accessibility constraints make name unlocking more powerful. Additionally, adjusting accessibilities allows us to enforce or prevent overriding: if, in the J_L version of the program, some method m is supposed to override another method m', rule (OVRPRES) will create a constraint ensuring that m' is accessible at the point where m is declared. Conversely, if m is *not* supposed to override m' but would override it according to Java's overriding rules, (OVRPREV) will constrain the accessibility of m' to prevent the overriding after all.

On the other hand, locked bindings also enable more flexible accessibility constraints: In the original formulation of accessibility constraints given by Steimann and Thies [25], there is a constraint rule that would, in the above example, bar us from raising the accessibility of t_2. The rationale for this rule is to prevent a class like `D` from inheriting two types of the same name from both a superclass and an interface, since these types can then not be accessed by their simple names. We can dispense with this rule, since name unlocking will insert qualifiers if necessary. Two other rules for preventing name capture due to hiding and changed overloading resolution are likewise rendered obsolete.

5.3 Refactoring on J_L

Many refactorings become simpler and more powerful if they are formulated at the level of J_L rather than on plain Java. The most striking illustration of the benefits of J_L is provided by type-related refactorings such as EXTRACT INTERFACE, which was briefly introduced in Sec. 2.

Tip *et al.*'s work on type-related refactorings [28] presents 38 type constraint rules for Java 1.4 in detail and gives an algorithm for determining updatable declarations for EXTRACT INTERFACE based on the generated constraint system. Apart from these genuinely type correctness-related constraints, the authors also briefly sketch some additional constraints that are not needed for

```
462  interface I^{t6} {
463    void m^{m3}(↑t_2 b^{v8});
464  }
465
466  class C^{t1} implements ↑t_6 {
467    private class B^{t2} { }
468    void m^{m1}(↑t_2 b^{v1}) overrides ↑m_3 { }
469    void n^{m2}() { }
470  }
471
472  interface J^{t3} { class B^{t4} { } }
473
474  class D^{t5} extends ↑t_1 implements ↑t_3 {
475    D^{c1}(↑t_1 c1^{v2}, ↑t_5 d^{v3}) { ↑v_2.↑m_1(null); }
476    D^{c2}(↑t_1 c2^{v4}, ↑t_1 o^{v5}) {
477      ↑v_4.↑m_2();
478      ↑t_5 d^{v6} = new ↑c_1(↑v_4, null);
479    }
480    ↑t_4 f^{v7};
481  }
```

(a)

```
482  interface I^{t6} {
483    void m^{m3}(↑t_2 b^{v8});
484  }
485
486  class C^{t1} implements ↑t_6 {
487    private class B^{t2} { }
488    void m^{m1}(↑t_2 b^{v1}) overrides ↑m_3 { }
489    void n^{m2}() { }
490  }
491
492  interface J^{t3} { class B^{t4} { } }
493
494  class D^{t5} extends ↑t_1 implements ↑t_3 {
495    D^{c1}(↑t_6 c1^{v2}, ↑t_5 d^{v3}) { ↑v_2.↑m_3(null); }
496    D^{c2}(↑t_1 c2^{v4}, ↑t_6 o^{v5}) {
497      ↑v_4.↑m_2();
498      ↑t_5 d^{v6} = new ↑c_1(↑v_4, null);
499    }
500    ↑t_4 f^{v7};
501  }
```

(b)

Figure 20: An application of EXTRACT INTERFACE on a J_L program

type correctness, but rather to prevent inadvertent changes to name binding and overloading resolution.

These additional constraints are not discussed in great detail, and in particular the correctness proof presented for EXTRACT INTERFACE does not consider them at all and simply assumes that such binding changes cannot happen. It is furthermore tacitly assumed that the necessary type changes do not fail due to insufficient accessibilities.

If we reformulate refactorings such as EXTRACT INTERFACE at the level of J_L, these assumptions are automatically satisfied. We can concentrate on the type-related issues germane to the refactoring, and leave it to the translation from J_L to Java to address naming and access control by making the necessary changes. This not only simplifies the specification of refactorings, but also makes them more powerful: inadvertent binding changes can often be fixed by adapting names and access modifiers, while a purely type constraint-based approach would have to reject the refactoring out of hand.

5.4 An Example of a J_L Refactoring

Let us take another look at the example from Sec. 2, and see how this application of EXTRACT INTERFACE plays out in J_L. The J_L version of the input program of Fig. 5(a) was shown in Fig. 18. Recall that we want to extract from class `C`, or t_1 in J_L, an interface `I` with a method `m(C.B)` for method m_1 to implement.

The first step of this refactoring is easy: create the new interface (we assign it the fresh label t_6), insert a declaration of the method we want to extract (labeled m_3), and have t_1 implement t_6, as shown in Fig. 20(a). In Java, even such a simple transformation would be fraught with peril: introducing the new interface might upset existing name bindings; references to types in the parameter lists of the extracted methods may need to be adjusted; and sometimes, as in this case, one of these types may not even be accessible. By formulating the refactoring on J_L instead, we can rely on the translation to Java to take care of all these issues.

Since the new interface is, as yet, not mentioned anywhere in the program (except in the `implements` clause of t_1), this step does not change the program's external behavior. It does, however, change overriding slightly, since m_1 now overrides its extracted counterpart m_3. In J_L, this has to be made explicit by inserting an `overrides` declaration as shown. More generally, for every method m to which EXTRACT INTERFACE creates a counterpart m' in

the new interface, m' has to be added to m's `overrides` clause.

The more interesting part of the refactoring is the second part of the transformation: now that we have the new interface t_6, we want to take advantage of it, and change as many variables as possible from type t_1 to t_6. For this, we rely on the algorithm for computing updatable declarations that was presented in [28]. We note that all constraint rules presented for Java make sense for J_L as well, except for the ones aiming at preventing changes in name binding or overloading resolution, which become unnecessary.

For the above example program, the algorithm determines that the types of the parameters v_2 and v_5 can be updated from $\uparrow t_1$ to $\uparrow t_6$. We also have to update the call to m_1 on line 475 to bind to m_3 instead as shown in Fig. 20(b). This will, of course, not affect dynamic dispatch at runtime.

In general, to determine which calls have to be updated we need to know the set E of expressions whose type is updated, and the type they are updated to (here always I); the algorithm in [28], for example, already computes this set. For a virtual call $e. \uparrow m(\ldots)$ with $e \in E$, we determine the method m' in I that m overrides, and replace it with $e. \uparrow m'(\ldots)$. This step, like the updating of overriding relationships above, is left implicit in formulations of EXTRACT INTERFACE for Java, where the binding will change silently, but is made explicit in J_L.

5.5 Porting Type-related Refactorings to J_L

The informal description of EXTRACT INTERFACE for J_L in the previous subsection is easily turned into a pseudocode algorithm, shown in Fig. 21. Note that every step of the refactoring except for line 7, which updates overriding declarations, and line 13, which adjusts virtual calls, would also occur in a Java-level specification.

To save space, we have not elaborated the preconditions the refactoring needs to check in line 2: an actual implementation should check, among others, that C is not a library class, and that none of the methods to be extracted is static. Crucially, however, these preconditions can all be taken be taken directly from a Java-based specification of the refactoring. We can, of course, omit any preconditions designed to prevent name binding changes or accessibility problems, since these issues are handled instead in the translation from J_L to Java.

Porting other type-related Java refactoring to J_L is analogous: take the Java specification, remove unnecessary preconditions, and make changes to overriding and call targets explicit. PULL UP METHOD, for instance, needs to update the `overrides` clauses of any methods that should override the pulled-up method after the refactoring.[9]

The procedure `Adjust Virtual Calls` can be reused by other refactorings: when using PULL UP METHOD to pull up a method m from a class A to a class B, all (unqualified) `this` accesses within m change their type from A to B, so virtual method calls on these accesses have to be adjusted.

In this way, existing refactorings can easily be lifted from Java to J_L by making changes in overriding explicit and using `Adjust Virtual Calls` to rectify the targets of virtual calls.

6 Implementation

We have implemented our approach in the JRRT refactoring tool [20]. JRRT is based on the JastAddJ front end, which it uses for parsing and to provide syntax trees. We have worked out specifications and implementations of many commonly used refactorings, which have been evaluated and compared to other implementations in previous work [17]. In this section, we will briefly highlight some of the salient points of our implementation of J_L and the transformations from and to Java.

We implement locked bindings by introducing new node types in the AST together with special lookup rules that implement direct binding. Similarly, nodes corresponding to method declarations are extended with an additional field to record explicit overriding.

To speed up translation to and from J_L, our implementation does not usually lock all bindings in the entire program, instead it is up to individual refactorings to determine which names are in danger of changing their binding and to replace them with locked bindings.

[9] Just as for a Java-level implementation, additional analysis is needed to ensure that the changed overriding does not affect virtual method dispatch.

```
1:  procedure EXTRACT INTERFACE_{J_L} (ClassDecl C,
                                   String n, Set⟨Method⟩ M)
2:  check Java-level preconditions
3:  I ← new interface with qualified name n
4:  for all m ∈ M do
5:     m' ← new method with same return type,
              parameters and thrown exceptions as m
6:     insert m' into I
7:     add ↑m' to overrides clause of m
8:  add ↑I to implements clause of C
9:  U_d ← updatable declarations
10: for all u ∈ U_d do
11:    change type of u to ↑I
12: U_e ← updatable expressions
13: Adjust Virtual Calls(U_e, I)

14: procedure Adjust Virtual Calls
                            (Set⟨Expr⟩ U_e, Type T)
15: for all virtual calls c = e_0.↑m(e_1, ..., e_n) do
16:    if e_0 ∈ U_e then
17:       m' ← resolve c on T
18:       replace c with e_0.↑m'(e_1, ..., e_n)
```

Figure 21: EXTRACT INTERFACE on J_L

For the common case of refactorings that do not alter the type hierarchy, a conservative over-approximation of the set of endangered names can be determined as follows.

We consider a method or constructor *affected* by the refactoring if its signature or location changes; similarly, a class, interface, field or other variable is affected if its name or location changes.

A constructor or method is considered *potentially affected* if it has the same name and arity as an affected method/constructor, and likewise for types and variables. A reference is potentially affected if it either refers to a potentially affected declaration, it is itself moved by the refactoring, or the type of its qualifying expression or one of its arguments (for method invocation expressions) changes.

A refactoring then only needs to lock potentially affected names, and only needs to introduce explicit overriding for potentially affected methods.

Refactorings like EXTRACT INTERFACE that do change the hierarchy have to perform additional locking.

To implement name unlocking, we have implemented the algorithm introduced in Sec. 3, taking all lookup rules of Java 5 into account. Due to the concise syntax and well-developed infrastructure of JastAddJ this can be achieved in about 1400 lines of code, which is roughly the same as the corresponding lookup rules.[10]

For accessibility handling, we have implemented a module to collect and solve accessibility constraints. Because most queries contained in the constraint rule's preconditions (such as *binds*, *overrides* and *receiver-type*) are directly provided by the JastAddJ front end, the implementation of the accessibility constraint generator could be achieved with less than 650 lines of code and thus turns out to be quite concise. The accessibility constraints are then translated into constraints over integers and solved using the Cream constraint solving library [26].

7 Evaluation

We will now present an evaluation of our approach and its implementation with respect to correctness and scalability.

[10]These and all following code size measurements were generated using David A. Wheeler's 'SLOCCount' [31].

Renamed Entity	Total	Inapp-licable	Missing Feature	Rejected by Eclipse	Same Result
Package	35	17	4	0	14
Type	247	16	35	68	128
Method	235	26	18	15	176
Variable	215	30	18	33	134

Table 1: Evaluation of RENAME on Eclipse's test suite

7.1 Correctness of Reference Construction

Ideally, we would like to formally verify that our naming framework always constructs references that bind to the intended declaration, and that the accessibility constraints faithfully capture Java's access control rules. However, while there has been some work on the formalization of access control [21], the name binding rules have, to our knowledge, never been formally specified.

For this reason, we have chosen a more empirical approach to convince ourselves of the correctness of our naming framework. We have implemented the RENAME refactoring for packages, types, methods and variables and tested them on the publicly available test suite for Eclipse. All test cases consist of an input program, a description of the renaming to perform, and an expected output program (none in the case of tests where the refactoring is expected to be rejected). We adapted the test suite to use our refactoring tool for performing the refactoring instead of the JDT.

The results of evaluating our implementation of the RENAME refactorings in this way are shown in Table 1. It lists the four considered refactorings in the first column. For every refactoring, the column labelled "Total" indicates how many test cases are provided by Eclipse. The remaining four columns classify these test cases into four disjoint categories.

Category "Inapplicable" comprises those test cases that we could not run through our implementation, most of them because the input program does not compile: a side effect of basing our implementation on a compiler front end is that it cannot handle uncompilable programs, for which no consistent syntax trees are generated by the front end. While it would be nice to support refactoring of invalid code from a usability perspective, this is not a well-defined problem since the input program has no behavior that the refactoring could preserve. Also included in the category of inapplicable tests are some test cases which exercise details of Eclipse's precondition checking algorithm that have no counterpart in our approach.

Test cases in category "Missing Feature" test minor features we have not implemented yet; notably, Eclipse can rename similarly named elements along with the main element being renamed, or update what looks like names contained in string literals. Again, these are heuristic features that are not amenable to rigorous comparison.

Category "Rejected by Eclipse" encompasses test cases that are supposed to be rejected by the Eclipse implementation, but which can be handled by our implementation. This includes test cases where names have to be qualified to avoid capture, which Eclipse does not attempt to do.

The final category, "Same Result", are those test cases on which both implementations produce the same result.

In summary, our implementation does quite well: while we do not implement all the additional features that Eclipse provides, our naming framework handles all test cases correctly, and can indeed be used to perform renamings on which Eclipse has to give up.

7.2 Scalability and Performance

To investigate issues of scalability, we performed an experiment in which we systematically applied the two refactorings EXTRACT INTERFACE and PULL UP METHOD on a large collection of real-world Java applications, shown in Table 2. We were particularly interested in determining how often adjustment of accessibilities and name qualification arises on real code, as these are situations that current refactoring tools are ill-equipped to handle. The subject programs are publicly available and include frequently used frameworks such as JUnit and Tomcat, comprising more than one million lines of source code in total.

PULL UP METHOD We used our tool to move each method in each class, along with all fields, methods and member types of the same class used by that method, to the immediate superclass, except in cases where the superclass is a library class.

Table 2 shows, under the heading PULL UP METHOD, the total number of methods to be moved for each subject program (column "total"), next to the number of methods for which the tool detected a potential problem that could

Name	Size	Pull Up Method					Extract Interface				
		total	rejected	successful			total	rejected	successful		
				total	acc.	names			total	acc.	names
Apache Tomcat 6.0.x	169	4398	4396	2	1	0	1381	1261	1208	287	37
HSQLDB 2.0.0	144	280	226	54	14	0	577	84	493	138	37
Xalan-Java 2.4	110	3872	2921	951	567	58	839	118	721	271	10
W3C Jigsaw 2.2.5	101	3002	1875	1127	427	446	970	112	858	243	0
Lucene 3.0.1	84	4376	4136	240	136	34	962	77	885	374	34
JHotDraw 7.5.1	76	2522	2061	461	249	55	689	88	601	93	45
ServingXML 1.1.2	65	940	827	113	96	6	1257	34	1223	246	5
JGroups 2.10	62	1934	1880	54	27	3	516	76	440	160	23
Hadoop Core 0.22	49	1179	1007	172	139	55	622	126	496	194	21
JMeter 1.9	41	1556	967	589	287	340	404	26	378	39	0
Clojure 1.1.0	29	1167	983	184	51	53	252	29	223	98	83
Draw2D 3.4.2	23	1396	1021	375	209	42	277	22	255	89	2
HTMLParser 1.6	22	520	359	161	25	7	148	10	138	12	0
Jaxen 1.1.1	12	442	402	40	28	2	167	15	152	40	0
Commons IO 1.4	5	99	91	8	7	0	74	6	68	4	0
JUnit 4.5	5	241	204	37	20	3	108	21	87	6	5
JUnit 3.8.1	4	183	106	77	46	6	52	5	47	15	0
JUnit 3.8.2	4	149	99	50	29	6	48	5	43	13	0
Commons Codec 1.3	2	20	14	6	4	0	19	2	17	2	0
Jester 1.37b	2	5	2	3	0	0	30	4	26	0	0
total	1009	28281	23577	4704	2362	1116	9392	1033	8359	2324	302

Table 2: Quantitative evaluation; sizes in thousands of lines of source code

render the output program uncompilable or change its semantics and hence rejected the refactoring (column "rejected").

The following three columns list the total number of successful refactoring applications (sub-column "total" of column "successful"), the number of cases that required at least one accessibility adjustment (sub-column "acc."), and the number of cases where a name adjustment was made (sub-column "names").[11]

Given the large number of refactoring applications, we cannot give a detailed analysis of the situations where the refactoring was rejected. To take the most extreme example, of the 4398 possible refactoring applications on Tomcat 4396 were rejected, the most common reasons being changes in the evaluation order of field initializers (2360 cases) and possible changes in dynamic method resolution (1602 cases). In other cases the numbers are not quite so drastic, leading to such a large number of successful refactorings that it was impractical to manually check preservation of program behavior in each case. Instead, we checked that the output program was still compilable.

Averaging over all programs, accessibilities were adjusted in about 50.2% of all successful applications, names in 23.7%. The need for accessibility adjustment arose often when moving methods between packages, with declarations referenced in the moved members becoming inaccessible. Naming issues occurred frequently when moved methods held references to static members or member types which needed further qualifications afterwards.

Extract Interface The final column group of Table 2 shows the result of using Extract Interface to extract an interface from each (non-anonymous) class in each subject program, containing all the public non-static methods of the class, to a new package where it was to serve as a published interface. Accessibility adjustments were needed in 27.8% and name adjustments in 3.6% of all successful applications.

Given its prototype nature, our current implementation is not ready for a full-fledged performance evaluation. Performance measurements taken while collecting the data for Table 2 show that a single application (whether successful or not) of any of the three refactor-

[11] This category includes insertion of upcasts to rectify overloading resolution, and of qualifiers to avoid name capture; we do not count the fairly trivial case of qualifying types with their package name.

Name	cases	ECLIPSE succ.	ECLIPSE reject	ECLIPSE error	INTELLIJ IDEA succ.	INTELLIJ IDEA reject	INTELLIJ IDEA error
Xalan-Java 2.4	49	0	49	0	8	40	1
Lucene 3.0.1	30	2	26	2	7	22	1
Clojure 1.1.0	13	1	8	4	7	6	0
Jaxen 1.1.1	2	0	2	0	0	2	0
JUnit 3.8.1	6	0	6	0	0	6	0
total	100	3	91	6	22	76	2

Table 3: 100 PULL UP METHOD refactorings with accessibility and naming issues in Eclipse 3.6.0 and IntelliJ IDEA 9.0.4

ings completes, on average, within 46 seconds or less, even on the largest of our benchmarks.[12] 90% of all applications complete within 53 seconds, with occasional outliers measured to take several minutes. This seems to be due to external factors: when re-run in isolation, such cases finished in well under a minute.

The strategy for approximating the set of names to be locked during translation outlined in above is very effective in practice: we found that PULL UP METHOD, e.g., never locks more than 7% of all names and usually much less than that, which never becomes a bottleneck. These numbers show that while there is room for improvement, our approach is practically feasible.

7.3 Comparison with Other Refactoring Tools

To see how our implementation compares with other refactoring tools we chose 100 cases where our PULL UP METHOD had adjusted both accessibility and naming, and manually performed these refactorings in Eclipse and IntelliJ IDEA, with the results shown in Table 3.

The first column corresponds to the one in Table 2, the second column gives the number of PULL UP METHOD refactorings involving naming and accessibility issues. While our tool succeeded (i.e., performed the refactoring and produced compilable output) in all cases, Eclipse and IntelliJ showed a rather different picture. Eclipse could only successfully refactor in three cases, while IntelliJ succeeded in 22 cases. In the majority of cases—91 for Eclipse and 76 for IntelliJ—both tools rejected the refactoring with an error message. In six and two cases, respectively, the tools failed completely and performed changes leading to uncompilable code. We could not perform a similar comparison for EXTRACT INTERFACE, as Eclipse does not support extraction into a different package.

[12] Timings obtained on a 2GHz Intel Centrino Duo running a Sun HotSpot JVM version 1.6.0_21 on a 1.2 GB heap under Microsoft Windows 7 [6.1.7600].

7.4 Discussion

In our quantitative evaluation, we applied the refactorings indiscriminately all over every subject program. The fraction of cases where it would make sense to apply the refactoring in order to improve a program's design is, of course, likely to be very small, but as our results show our tool is robust enough to handle a wide variety of situations, including those where other tools fail.

Since we treat accessibility adjustment as a global constraint problem, our approach may sometimes end up suggesting a large number of changes to many different parts of the program. It may be doubtful whether a refactoring that requires extensive changes in order to go through is actually worth performing. However, we believe that this is not for the refactoring tool to decide. Instead, this issue is probably best handled in the user interface by providing a preview of the proposed changes to the programmer, who can still choose to abort the refactoring if it is too invasive.

8 Related Work

Almost two decades after inception of the discipline as marked by the theses of Griswold [8] and Opdyke [15], refactoring is still a hard problem. This is evidenced by a steadily growing body of literature on the subject, still dealing with the same basic problem as the inaugural works: how to construct refactoring tools that are as reliable as other programming tools like compilers and debuggers.

Starting with the work by Opdyke [15] and Roberts [16], most previous refactoring research has relied on pre- and post-conditions to ensure that program behavior is preserved. However, the presence of many language constructs such as nested classes, overloading, and access modifiers results in extremely complex preconditions that are hard to get right. This is what

motivated Schäfer's approach of formulating refactorings in terms of dependency preservation [18, 17], of which binding preservation is one instance.

Other work considered various programming language features such as class hierarchies [28, 10], generics [1, 30, 6], design patterns [29, 11], and access modifiers [25]. Most of these works focus on a single language feature in isolation, and do not consider the complex interactions between different features that have to be addressed to make refactoring tools robust.

In recent work, Steimann *et al.* [24] explore the use of conditional, quantified constraints as a unifying framework for specifying and implementing refactorings and show promising first results. An advantage of their purely constraint-based framework over a combined approach like ours is that there is no need to explicitly schedule different adjustments as we do with binding unlocking and accessibility constraint solving. However, a constraint-based specification is in general less well-suited to describe refactorings that introduce or delete program elements (as opposed to just moving them or manipulating their attributes) since constraint solvers generally presuppose a fixed domain of elements to work on.

9 Conclusions

Implementing behavior-preserving program transformations is difficult, particularly at the source level. Modern mainstream programming languages such as Java provide many convenient idioms and syntactic sugar that make it very hard not only to ensure that the transformed program has the same behavior as the input program, but even that it compiles in the first place. One particularly complex, yet very fundamental problem is how to deal with name binding, which is governed by a sophisticated set of lookup and access control rules.

In this paper, we have introduced J_L, a representation of Java programs that abstracts away from the details of name lookup and access control, instead providing a view of the program in which references to declared entities appear locked: they only change when explicitly rebound by the refactoring, and otherwise keep their original binding. We have shown that refactorings become much more robust and powerful when formulated at the level of J_L.

In order for J_L to be usable, we need translations from Java to J_L and vice versa. We have shown how such a translation can be achieved with the help of a reference construction function and accessibility constraints: the former constructs references binding to a target declaration, the latter determine how declared accessibilities have to be adjusted to satisfy access control rules. We have implemented these translations and put them to work by implementing several refactorings on top of them. To evaluate our implementation, we have systematically applied two of these refactorings to a large body of real-world Java applications, showing that our tool is able to perform transformations that are beyond the scope of current state-of-the-art refactoring engines.

While our current work specifically addresses Java, we believe that the basic approach applies to other languages as well. Languages such as C#, Scala or Eiffel have similar name binding and access control concepts as Java, although details differ between languages. Refactorings for these languages would almost certainly also benefit from a lookup-free, access control-free program representation such as J_L.

References

[1] Alan Donovan, Adam Kieżun, Matthew Tschantz, and Michael Ernst. Converting Java Programs to Use Generic Libraries. In *Object-Oriented Programming, Systems and Languages (OOPSLA)*, pages 15–34, 2004.

[2] Torbjörn Ekman and Görel Hedin. Modular Name Analysis for Java Using JastAdd. In *Generative and Transformational Techniques in Software Engineering (GTTSE)*, pages 422–436. Springer-Verlag, 2006.

[3] Torbjörn Ekman and Görel Hedin. The JastAdd Extensible Java Compiler. In *Object-Oriented Programming, Systems and Languages (OOPSLA)*, pages 1–18. ACM Press, 2007.

[4] Torbjörn Ekman and Görel Hedin. The JastAdd system—modular extensible compiler construction. *Science of Computer Programming*, 69(1–3):14–26, 2007.

[5] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.

[6] Robert M. Fuhrer, Frank Tip, Adam Kieżun, Julian Dolby, and Markus Keller. Efficiently Refactoring Java Applications to Use Generic Libraries. In *European Conference on Object-Oriented Programming (ECOOP)*, 2005.

[7] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison Wesley, 3rd edition, 2005.

[8] William G. Griswold. *Program Restructuring as an Aid to Software Maintenance*. Ph.D. thesis, University of Washington, 1991.

[9] JetBrains. IntelliJ IDEA 10.5. `http://www.jetbrains.com/idea`, 2011.

[10] Hannes Kegel and Friedrich Steimann. Systematically Refactoring Inheritance to Delegation in Java. In *International Conference on Software Engineering (ICSE)*, pages 431–440. ACM Press, 2008.

[11] Joshua Kerievsky. *Refactoring to Patterns*. Addison Wesley, 2005.

[12] Eclipse Foundation. Eclipse 3.6 JDT. `http://www.eclipse.org/jdt`, 2011.

[13] Torm Mens and Tom Tourwé. A Survey of Software Refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, February 2004.

[14] Emerson R. Murphy-Hill, Chris Parnin, and Andrew P. Black. How We Refactor, and How We Know It. In *ICSE*, 2009.

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

[16] Donald B. Roberts. *Practical Analysis for Refactoring*. Ph.D. thesis, University of Illinois at Urbana-Champaign, 1999.

[17] Max Schäfer and Oege de Moor. Specifying and Implementing Refactorings. In Martin Rinard, editor, *Object-Oriented Programming, Systems and Languages (OOPSLA)*. ACM Press, 2010.

[18] Max Schäfer, Torbjörn Ekman, and Oege de Moor. Sound and Extensible Renaming for Java. In Gregor Kiczales, editor, *Object-Oriented Programming, Systems and Languages (OOPSLA)*, pages 227–294. ACM Press, 2008.

[19] Max Schäfer, Torbjörn Ekman, Ran Ettinger, and Mathieu Verbaere. Refactoring bugs. `http://code.google.com/p/jrrt/wiki/RefactoringBugs`, 2011.

[20] Max Schäfer, Torbjörn Ekman, and Andreas Thies. JRRT—JastAdd Refactoring Tools. `http://code.google.com/p/jrrt`, 2011.

[21] Norbert Schirmer. Analysing the Java package/access concepts in Isabelle/HOL. *Concurrency - Practice and Experience*, 16(7):689–706, 2004.

[22] Friedrich Steimann. The Infer Type Refactoring and its Use for Interface-Based Programming. *Journal of Object Technology*, 6(2):99–120, 2007.

[23] Friedrich Steimann. Constraint-Based Model Refactoring. In Jon Whittle, Clark, and Thomas Kühne, editors, *Model Driven Engineering Languages and Systems (MoDELS)*. Springer-Verlag, 2011.

[24] Friedrich Steimann, Christian Kollee, and Jens von Pilgrim. A Refactoring Constraint Language and its Application. In *European Conference on Object-Oriented Programming (ECOOP)*, 2011.

[25] Friedrich Steimann and Andreas Thies. From Public to Private to Absent: Refactoring Java Programs under Constrained Accessibility. In Sophia Drossopoulou, editor, *European Conference on Object-Oriented Programming (ECOOP)*, pages 419–443. Springer-Verlag, 2009.

[26] Naoyuki Tamura. Cream: Class Library for Constraint Programming in Java. `http://bach.istc.kobe-u.ac.jp/cream`, 2009.

[27] Frank Tip, Robert M. Fuhrer, Adam Kieżun, Michael D. Ernst, Ittai Balaban, and Bjorn De Sutter. Refactoring using Type Constraints. *ACM Trans. Program. Lang. Syst.*, 33(3):9, 2011.

[28] Frank Tip, Adam Kieżun, and Dirk Bäumer. Refactoring for Generalization using Type Constraints. In *Object-Oriented Programming, Systems and Languages (OOPSLA)*, pages 13–26. ACM Press, 2003.

[29] Lance Tokuda and Don Batory. Evolving Object-Oriented Designs with Refactorings. *Automated Software Engineering*, 8(1):89–120, January 2001.

[30] Daniel von Dincklage and Amer Diwan. Converting Java Classes to Use Generics. In *Object-Oriented Programming, Systems and Languages (OOPSLA)*, pages 1–14, 2004.

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

A Accessibility Constraint Rules

The constraint rules governing access modifiers are defined as shown in Fig. 23; they rely on the program queries listed in Fig. 17. The rules are explained as follows:

- (ACC-1): This is the basic rule for type and member access [7, §6.6.1].
- (ACC-2), (CTORACC): If a member or constructor of an object is accessed from outside the package in which it is declared by code that is not responsible for the implementation of that object, its accessibility must be `public` [7, §6.2.2].
- (ACC-3): For a type member to be accessible, its owning type must also be accessible, even if it is not explicitly referenced [7, §6.6.1].
- (INHACC): This rule makes sure that members accessed through a type that inherits them are still accessible in the refactored program.
- (OVRPRES), (OVRPREV): These two rules make sure that existing overriding relationships between methods are preserved, and no new ones are introduced.
- (HID): This rule prevents erroneous hiding.
- (SUB): Method hiding or overriding cannot decrease accessibility, which is ensured by this rule.
- (INH-1), (INH-2), (INH-3): These rules cover several subtle cases arising from multiple inheritance of a method from both a superclass and a superinterface, also known as interface inheritance [7, §8.4.8.4].
- The remaining rules ensure various other accessibility requirements found in the language specification.

abstract $\subseteq D$	*abstract*(d) holds if d is declared `abstract`
binds $\subseteq R \times D$	*binds*(r, d) holds if reference r binds to declaration d
constructor $\subseteq M$	*constructor*(m) holds if m is a constructor
enumeration $\subseteq T$	*enumeration*(t) holds if t is an enumeration type
head-of-cu $\subseteq T$	*head-of-cu*(t) holds if t has the same name as its compilation unit
hides $\subseteq M \times M$	*hides*(m_1, m_2) holds if method m_1 hides method m_2 [7, §8.4.8]
inherits $\subseteq T \times M$	*inherits*(t, m) holds if type t inherits member m
interface $\subseteq T$	*interface*(t) holds if t is an interface type
main-method $\subseteq M$	*main-method*(m) holds if m is a main method
may-hide $\subseteq M \times M$	*may-hide*(m_1, m_2) holds for static methods m_1, m_2 if the return type of m_1 is return-type substitutable for that of m_2 and the `throws` clause of m_1 does not conflict with that of m_2 [7, §8.4.8.3]
method $\subseteq M$	*method*(m) holds if m is a method
override-equiv $\subseteq M \times M$	*override-equiv*(m_1, m_2) holds if methods m_1 and m_2 have override equivalent signatures [7, §8.4.2]
overrides $\subseteq M \times M$	*overrides*(m_1, m_2) holds if method m_1 directly overrides method m_2; that means m_1 overrides m_2 (as specified in [7, §8.4.8.1]) and there exists no other method m_3 that overrides m_1 and that is overridden by m_2
receiver-type $\subseteq R \times T$	for a field access or method invocation r, *receiver-type*(r, t) holds if t is the type on which the field is accessed or the method invoked
same-decl $\subseteq M \times M$	*same-decl*(m_1, m_2) holds if m_1 and m_2 are declared in the same declaration statement
static $\subseteq M$	*static*(m) holds if m is declared `static`
super-qualified $\subseteq R$	*super-qualified*(r) holds if r is a `super` field access or method invocation

Figure 22: Program queries used in the constraint rules of Fig. 23

$$\frac{\textit{binds}(r,d)}{\langle d\rangle \geq_A \alpha(r,d)}\ (\textsc{Acc-1})$$

$$\frac{\textit{binds}(r,d) \quad \textit{receiver-type}(r,t) \quad \pi(r) \neq \pi(d) \quad t \not\leq_T \tau(r) \quad \neg\textit{super-qualified}(r) \quad \neg\textit{static}(d)}{\langle d\rangle = \texttt{public}}\ (\textsc{Acc-2})$$

$$\frac{\textit{receiver-type}(r,t)}{\langle t\rangle \geq_A \alpha(r,t)}\ (\textsc{Acc-3})$$

$$\frac{\textit{binds}(r,d) \quad r = \texttt{new}\, C(\ldots) \quad \pi(r) \neq \pi(d)}{\langle d\rangle = \texttt{public}}\ (\textsc{CtorAcc})$$

$$\frac{\textit{binds}(r,d) \quad \textit{receiver-type}(r,t) \quad t <_T \tau(d)}{\langle d\rangle \geq_A \iota(t,\tau(d))}\ (\textsc{InhAcc})$$

$$\frac{\textit{overrides}(m_2,m_1)}{\langle m_1\rangle \geq \omega(m_2,m_1)}\ (\textsc{OvrPres})$$

$$\frac{\tau(m_2) <_T \tau(m_1) \quad \textit{override-equiv}(m_2,m_1) \quad \neg\textit{static}(m_2) \quad \neg\textit{overrides}(m_2,m_1)}{\langle m_1\rangle <_A \omega(m_2,m_1)}\ (\textsc{OvrPrev})$$

$$\frac{\tau(m_2) <_T \tau(m_1) \quad \textit{override-equiv}(m_2,m_1) \quad \textit{static}(m_2) \quad \neg\textit{may-hide}(m_2,m_1)}{\langle m_1\rangle <_A \alpha(m_2,m_1)}\ (\textsc{Hid})$$

$$\frac{\textit{overrides}(m_2,m_1) \vee \textit{hides}(m_2,m_1)}{\langle m_2\rangle \geq_A \langle m_1\rangle}\ (\textsc{Sub})$$

$$\frac{\tau(m_1) = i \quad interface(i) \quad inherits(c,m_1) \quad inherits(c,m_2) \quad \neg\textit{abstract}(m_2) \quad \textit{override-equiv}(m_2,m_1)}{\langle m_2\rangle = \textit{public} \vee \langle m_2\rangle < \iota(c,\tau(m_2))}\ (\textsc{Inh-1})$$

$$\frac{\tau(m_1) = i \quad interface(i) \quad inherits(c,m_1) \quad inherits(c,m_2) \quad \neg\textit{abstract}(m_2) \quad \neg\textit{abstract}(c) \quad \textit{override-equiv}(m_2,m_1)}{\langle m_2\rangle = \textit{public}}\ (\textsc{Inh-2})$$

$$\frac{\tau(m_1) = i \quad interface(i) \quad c <_T i \quad c <_T \tau(m_2) \quad \textit{static}(m_2) \quad \textit{override-equiv}(m_2,m_1)}{\langle m_2\rangle <_A \iota(c,\tau(m_2))}\ (\textsc{Inh-3})$$

$$\frac{t[\,] \in T}{\langle t[\,]\rangle = \langle t\rangle}\ (\textsc{Array}) \qquad \frac{\textit{same-decl}(m_1,m_2)}{\langle m_1\rangle = \langle m_2\rangle}\ (\textsc{SameDecl}) \qquad \frac{\textit{method}(m) \quad \textit{abstract}(m)}{\langle m\rangle > \texttt{private}}\ (\textsc{AbsMeth})$$

$$\frac{t \in T_{top}}{\langle t\rangle \in \{\texttt{package}, \texttt{public}\}}\ (\textsc{TLType-1}) \qquad \frac{t \in T_{top} \wedge \neg\textit{head-of-cu}(t)}{\langle t\rangle <_A \texttt{public}}\ (\textsc{TLType-2})$$