A Comprehensive Approach to Naming and Accessibility in Refactoring Java Programs

Max Schäfer, Andreas Thies, Friedrich Steimann, and Frank Tip

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.

Index Terms—Restructuring, reverse engineering, and reengineering, object-oriented languages, Java

1 Introduction

Refactoring is the process of restructuring a program by means of behavior-preserving source code transformations, themselves called refactorings [1], [2]. Over the past decade, automated tool support for refactoring has become available in popular IDEs such as Eclipse and VisualStudio. However, even state-of-the-art tools are still quite fragile, and often render refactored programs uncompilable or silently change program behavior.

An important cause for this lack of 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 nonmodular so that changes 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.

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 changes in name resolution or overriding.

Examples from both categories are easy to find even with state-of-the-art refactoring tools [3] such as the refactoring engines of Eclipse JDT [4] and IntelliJ IDEA [5].[1]

In this paper, we propose a comprehensive solution to these issues in the form of J_L, a lookup-free and 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. Unless explicitly rebound to a new declaration, locked bindings continue to refer to the same target declaration even if that declaration is renamed or moved; 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

- *M. Schäfer is with the IBM T.J. Watson Research Center, 19 Skyline Drive, Hawthorne, NY 10532. E-mail: mschaefer@us.ibm.com.*
- *A. Thies and F. Steimann are with the Fernuniversität in Hagen, Universitätsstraße 1, D-58097 Hagen, Germany. E-mail: andreas.thies@fernuni-hagen.de, steimann@acm.org.*
- *F. Tip is with the the David R. Cheriton School of Computer Science, University of Waterloo, 200 University Avenue West, Waterloo, ON N2L 3G1, Canada. E-mail: ftip@uwaterloo.ca.*

Manuscript received 29 July 2011; revised 10 Jan. 2012; accepted 21 Jan. 2012; published online 13 Feb. 2012.
Recommended for acceptance by M. Robillard.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-2011-07-0228.
Digital Object Identifier no. 10.1109/TSE.2012.13.

1. Throughout this paper, whenever we refer to Eclipse JDT we mean version 3.6, and version 10.5 for IntelliJ IDEA, unless stated otherwise.

independent) translation to and from Java. Our translation from J_L to Java is based on two key techniques:

1. **Reference construction.** Unlocking the locked bindings of J_L, i.e., replacing them with normal Java names, is easy if we have a *reference construction* algorithm that, given a declaration and a location in the program, 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.** Java's access control rules 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 program adheres to the access control rules.

J_L and the translations to and from Java form the basis of the JRRT system [6], a prototype refactoring engine built on the JastAddJ Java compiler front end [7], which supports a growing number of popular refactorings [8]. We evaluate this implementation both on the internal test suite of the Eclipse refactoring engine and on a large suite of real-world applications, demonstrating that it handles many situations where existing state-of-the-art tools just give up.

In summary, our work makes the following main contributions:

- We propose J_L, a lookup-free and access control-free representation of Java programs and show how existing refactorings become simpler and more widely applicable when expressed on that representation.
- We show how an algorithm for constructing potentially qualified references that refer 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 that can be applied to a program. These rules yield 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)

Fig. 1. A simple example of RENAME.

2 Motivating Examples

We start by giving some examples to show that naming and accessibility are pervasive problems that have to be dealt with by many refactorings. We then outline our proposed solution, which employs a novel lookup-free, access control-free representation of Java programs to address these 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 the RENAME refactoring, which changes the name of a declared entity (such as a class, field, or method) and consistently updates all references 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.

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. 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 example 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

```
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)

Fig. 2. A simple example of INTRODUCE PARAMETER.

behavior, we have inserted an upcast to `long` on the argument, thus enforcing the same choice as before.

Similar precautions have to be taken for any refactoring that introduces, moves, or deletes declarations [8]. 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, for each name in the original program, which declaration it is supposed to bind to in the refactored program. A common naming framework then takes care of introducing qualifiers or upcasts where necessary to achieve the desired binding structure.

Current industrial-strength 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.

A notoriously difficult and, in current refactoring tool implementations, unsolved problem in binding preservation arises from Java's reflection API, which allows accessing classes, interfaces, fields, and methods by computed names. We do not tackle this issue here. There is current research on frameworks enabling sound static analysis in the presence of reflection [9] which, together with our approach, may help to enable reflection support for refactoring tools in the future.

2.2 Basic Accessibility Problems

Like naming, accessibility is also handled poorly by current tools. For instance, consider the scenario of Fig. 3, where the MOVE CLASS refactoring is applied to move class `B` from package `a` to package `b`. To remain accessible in the declaration on line 45, `B` has to be made `public` as shown on line 56. Eclipse fails to notice this problem and produces

```
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)

Fig. 3. A simple example of MOVE CLASS.

an uncompilable program; IDEA emits a warning, but does not attempt to fix the issue.

While this problem is detected by the compiler, failure to adjust accessibility can be more detrimental in presence of dynamic binding. For instance, moving class `B` in Fig. 4a 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 refactoring tool could instead suggest increasing the accessibility of `A.m`, and with it that of `B.m` (required by [10, Section 8.4.8.1]), to `protected`, as shown in Fig. 4b.

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 [1], [11], [12].

```
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)

Fig. 4. An example of MOVE CLASS involving dynamic binding.

```
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)

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

```
137 class A {  }
138 class B extends A {
139 }
140 class C extends B {
141   private static void baz(B p) {  }
142   public void foo(A q) {
143     baz(this);
144   }
145 }
146 class E extends B {
147   private static void foo(String r)
148   { }
149   void bar() { foo(null); }
150 }
```

(a)

```
151 class A {  }
152 class B extends A {
153   public void foo(A q) {
154     C.baz(this);                    // qualification added
155   }
156 }
157 class C extends B {
158   static void baz(B p) {  } // acc. increased
159 }
160 class E extends B {
161   private static void foo(String r)
162   { }
163   void bar() { foo((String)null); } // cast added
164 }
```

(b)

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

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. 5a. 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`.

Fig. 5b shows the program after applying the refactoring. The new interface `I` appears on lines 117-119 of Fig. 5b, 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. This cast always succeeds at runtime and only serves to ensure that the call is resolved to the correct declaration at compile time.

2.4 Naming and Accessibility Problems in PULL UP METHOD

Of course, EXTRACT INTERFACE is not the only refactoring that potentially faces such complications. Consider, for instance, the example program in Fig. 6a, 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. 6b.

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 158.

Names. Accessing the `static` method `baz` outside of its declaring class requires explicit qualification of the method call on line 154.

Overloading. Moving method `foo(A)` into class `B` makes the call `foo(null)` on line 163 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 163.

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 143. 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, which reject it.

2.5 Our Solution

The examples in this section suggest that naming and accessibility issues are pervasive in refactoring. More evidence of this can be found at the JRRT website [3], where we maintain a list of bugs found in refactorings as implemented by several industrial-strength refactoring engines, many of which concern naming and accessibility.

However, the treatment of these issues is largely orthogonal to the purpose of a specific refactoring. 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. This is the goal of the J_L representation we introduce in this paper. In J_L, normal Java names are abolished in favor of locked references of the form $\uparrow l$, where l is a label uniquely identifying a declaration, that directly bind to their declaration 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, so we need translations from Java to J_L and back. The following two sections develop the technical machinery needed for these translations; Section 5 will then show 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,

$$lookup : \text{ProgramPoint} \times \text{Reference} \rightharpoonup \text{Declaration},$$

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

We want to define a complementary *reference construction* function,

$$access : \text{ProgramPoint} \times \text{Declaration} \rightharpoonup \text{Reference},$$

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

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

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: $access(p, -)$ is a (partial) right inverse to $lookup(p, -)$.

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 $access(p, l)$. If $access(p, l)$ 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 (1), but is not useful for eliminating locked bindings. We show in this section how a suitable specification of name lookup can be systematically inverted to yield a practical implementation of $access$.

3.1 Name Lookup in Java

The Java Language Specification (JLS) introduces eight kinds of *declared entities* [10, Section 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* [10, Section 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 [10, Section 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* [10, Section 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

```
165 package p;
166
167 class Super {
168   int f; /*①*/
169   class A { }
170   int m(int i) { return 42; }
171   static int length = 56;
172 }
173
174 class Outer {
175   int f; /*②*/
176   int x;
177   class A { }
178   class Inner extends Super {
179     int f; /*③*/
180     int y;
181     int m(int f /*④*/) {
182       A a1;
183       Outer.A a2;
184       p.Outer.A a3;
185       int[] Super = {};
186       return x + y
187            + f                    // → ④
188            + this.f               // → ③
189            + super.f              // → ①
190            + Inner.this.f         // → ③
191            + Inner.super.f        // → ①
192            + Outer.this.f         // → ②
193            + ((Super)this).f      // → ①
194            + Super.length;
195     }
196   }
197 }
```

Fig. 7. Example for name lookup in Java.

methods with the same signature in superclasses or superinterfaces [10, Section 8.4.8.2].

Shadowing and hiding are both distinct from *obscuring* [10, Section 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, `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 169 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 183), which may itself be qualified by the name of its package (line 184).

Lines 186-194 show examples of qualified variable and method references. Line 186 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 187, as indicated by the comment.

The following lines show different forms 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 188). The field `f` from `Super`, although hidden by the field `f` in `Inner`, is accessible through a qualification with `super` (line 189). We can access the same two fields through qualification with `Inner.this` (line 190) and `Inner.super` (line 191), although such qualified `this` accesses are more usually employed to access shadowed fields of enclosing classes, as with the reference `Outer.this.f` (line 192). Note that for fields, the access `super.f` is equivalent to `((Super) this).f` (line 193), except that it has slightly more relaxed accessibility rules [10, Section 6.6.2].

Line 194 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 185, 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* [10, Section 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 [10, Section 8.8.7.1] refers to.

In the following, we will use the (nonstandard) 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

```
202 eq TypeDecl.getBodyDecl(int i).
203     lookupVar(String name) {
204   Variable res = memberField(name);
205   if(res != null)
206     return res;
207   res = lookupVar(name);
208   if(res != null)
209     if((inStaticContext() || isStatic())
210        && res.isInstanceVariable())
211       return null;
212   return res;
213 }
```

Fig. 8. Variable lookup from inside a type declaration.

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 to what declaration a reference binds 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 [7], [13].

JastAddJ is implemented in JastAdd [14], an extension of Java with attribute grammar features. Programs are represented by their abstract syntax trees (ASTs), and analyses are implemented as parameterized attributes on the nodes of the AST. Name lookup is mostly handled by a 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 presentational purposes, is given in Fig. 8.

The equation is of the form

```
216 syn Variable ClassDecl.memberField
217                         (String name) {
218   Variable f = localField(name);
219   if(f != null)
220     return f;
221   if(hasSuperclass()) {
222     f = superclass().memberField(name);
223     if(f != null)
224       if(f.isPrivate() ||
225            !f.accessibleFrom(this))
226         return null;
227       return f;
228  }
229  // search through interfaces omitted
230  return null;
231 }
232
233 syn Variable ClassDecl.localField
234                         (String name) {
235   for(BodyDecl bd : getBodyDecls())
236     if(bd instanceof FieldDecl) {
237       FieldDecl fd = (FieldDecl)bd;
238       if(name.equals(fd.getName()))
239         return fd;
240     }
241   return null;
242 }
```

Fig. 9. Member field lookup.

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

indicating that it defines the value of attribute `lookupVar` on any `BodyDecl` node that is the `i`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 204 to look up n as a member field of t; if a member field named n is found, its declaration is returned (line 206).
- Otherwise, `lookupVar` is recursively invoked on t itself in line 207 to search enclosing scopes. This conforms to a lexical scoping discipline where inner classes can see member fields of enclosing classes. The test in line 205 prevents recursion if a member field of the same name exists, implementing shadowing.
- Finally, the result of the recursive invocation is filtered in line 209: If t is itself declared as static or occurs in a static context, instance variables cannot be accessed inside t [10, Section 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.

3. We refer to the literature for a more detailed discussion of the syntax of JastAdd attribute definitions [7].

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 218), 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 220). Otherwise, `memberField` is invoked recursively on the superclass, if there is one,[4] (line 222) and on all superinterfaces (omitted from the figure). This implements inheritance, with line 225 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.

To resolve an arbitrary (possibly qualified) reference at a certain program point, one first has to determine whether the reference refers to a package, a type, a method, or a variable, and then dispatch to one of the more specialized attributes `lookupType`, `lookupMeth`, or `lookupVariable` to perform the actual lookup. In JastAddJ, there is no single attribute implementing this functionality. 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 [13].

For presentational purposes and 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 that 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` (for type names), `EXPRESSION_NAME` (for a name referring to a variable), or

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

```
243 syn Decl ASTNode.lookup(Reference r) {
244   return r.lookupAt(this);
245 }
246
247 syn Decl SimpleName.lookupAt(ASTNode nd)
248 {
249   String n = this.getName();
250   switch(nd.nameKind()) {
251   case EXPR_NAME:
252     return nd.lookupVar(n);
253   case AMBIGUOUS_NAME:
254     Decl res = nd.lookupVar(n);
255     if(res != null)
256       return res;
257     res = nd.lookupType(n);
258     if(res != null)
259       return res;
260     return nd.lookupPackage(n);
261   // other cases elided
262   }
263 }
264
265 syn Decl Dot.lookupAt(ASTNode nd) {
266   Expr l = getLeft();
267   String n = getRight().getName();
268   switch(nd.nameKind()) {
269   case EXPR_NAME:
270     return l.type().memberField(n);
271   case AMBIGUOUS_NAME:
272     Decl d = nd.lookup((Reference)l);
273     if(d instanceof PkgDecl) {
274       PkgDecl p = (PkgDecl)d;
275       Decl res = p.memberType(n);
276       if(res != null)
277         return res;
278       return p.subPackage(n);
279     } else if(d instanceof TypeDecl) {
280       TypeDecl t = (TypeDecl)d;
281       Decl res = t.memberField(n);
282       if(res != null)
283         return res;
284       return t.memberType(n);
285     } else {
286       return l.type().memberField(n);
287     }
288   // other cases elided
289   }
290 }
```

Fig. 10. Lookup of general references (simplified).

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 [10, Section 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 194 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

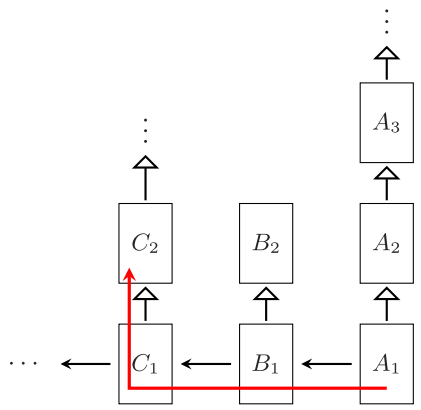

Fig. 11. Schematic illustration of field lookup.

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 nonstatic members are not accessed inside a static context, and performs overloading resolution for methods and constructors.

3.3 Inverting Variable Lookup

To obtain an implementation of reference construction, it would be tempting to try and invert every lookup attribute in isolation, for instance, defining an attribute `accessVar` that is right inverse to `lookupVar` in the sense of (1). But since `lookupVar` cannot resolve qualified names, `accessVar` could never produce qualified names either, leading to a rather simplistic reference construction algorithm.

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 (1).

To motivate the concept of an abstract reference, consider the lookup algorithm for fields presented 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.

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

```
291 class AbstractVarRef {
292   String name;
293   boolean visible;
294   TypeDecl source, bend;
295   // standard constructor elided
296 }
```

Fig. 12. Abstract references.

The path from the point of reference to the declaration can be visualized as an outward motion through enclosing classes until reaching a "bend" at C_1, and then proceeding upward on 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 in Fig. 12.

We will now show that the name lookup equations of the previous section 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 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

```
297 eq TypeDecl.getBodyDecl(int i).
298     accessVar(Variable v)
299 {
300  AbstractVarRef r = accMemberField(v);
301  if(r != null)
302   return r;
303  r = accessVar(v);
304  if(r != null) {
305   if((inStaticContext() || isStatic())
306        && v.isInstanceVariable())
307    return null;
308   if(memberField(v.getName()) != null)
309    r.visible = false;
310  }
311  return r;
312 }
313
314 eq ClassDecl.accMemberField
315     (Variable v)
316 {
317  AbstractVarRef r = accLocalField(name);
318  if(r != null)
319   return r;
320  if(hasSuperclass()) {
321   r = superclass().accMemberField(name);
322   if(r != null) {
323    if(v.isPrivate() ||
324       !v.accessibleFrom(this))
325     return null;
326    if(r.visible &&
327       localField(name) == null)
328     r.source = this;
329    else
330     r.visible = false;
331    r.bend = this;
332    return r;
333   }
334  }
335  return null;
336 }
337
338 eq ClassDecl.accLocalField(Variable v)
339 {
340  for(BodyDecl bd : getBodyDecls())
341   if(bd == v)
342    return new AbstractVarRef(v.getName(),
343                              true, this, this);
344  return null;
345 }
```

Fig. 13. Reference construction.

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 326-331.

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

```
346 Reference ASTNode.mkRef(AbstractVarRef r)
347 {
348   if(nameKind() == EXPR_NAME ||
349      nameKind() == AMBIGUOUS_NAME) {
350     SimpleName n= new SimpleName(r.name);
351     if(r.visible)
352       return n;
353     TypeDecl T = enclosingType(),
354               S = r.source, B = r.bend;
355     if(S == B && B == T)
356       // return access this.n
357     else
358       // return access ((↑S)↑B.this).n
359   }
360   return null;
361 }
```

Fig. 14. Skeleton of an algorithm for constructing an actual reference from an abstract one.

the same way as `localVar`, and is inverse to it in the following sense:

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

1. *`r.bend` encloses `n`; it is a subclass of `r.source`;*
2. *if `r.visible`, then `r.bend = r.source` and `n.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 suffices. 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.

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

```
362 class A {
363  int n() {
364   return 42;
365  }
366  int f() {
367   A a = new B();
368   return a.n();
369  }
370 }
371
372 class B extends A
373 {
374  int m() {
375   return 23;
376  }
377 }
```

(a)

```
378 class A {
379  int n() {
380   return 42;
381  }
382  int f() {
383   A a = new B();
384   return a.n();
385  }
386 }
387
388 class B extends A
389 {
390  int n() {
391   return 23;
392  }
393 }
```

(b)

Fig. 15. Example of a change in dynamic dispatch in spite of same binding structure.

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. 15a method `B.m` does not override method `A.n`, the renamed method `B.n` of Fig. 15b does. Hence, the method invocation on line 368 returns `42`, while its counterpart on line 384 returns `23`, although both of them bind to method `A.n`.

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 Section 4.5) to ensure that every method overrides those (and only those) methods mentioned in the overriding annotation, and abort the refactoring if this cannot be done. In the example of Fig. 15, we could, for instance, avoid the change in overriding by making method `A.n` private.

3.5 Summary

To unlock locked bindings, it is necessary to construct a possibly qualified reference that binds to a given variable, 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 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 source 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 sufficient.
- Else, if the accessed entity and the reference reside in the same package, at least `package` accessibility is required.

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`.

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

Fig. 16. Meaning of access modifiers ("accessibility") for member access, inheritance, and overriding.

- Else, if the accessing reference resides in a subclass of the class in which the accessed entity is declared, at least `protected` accessibility is required.
- Else, `public` accessibility is required.

Fig. 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 404, while the package accessible method `A.p` cannot be accessed from a different package on line 417. However, `protected` accessibility suffices to access method `A.q` from within the body of `B`, which is a subtype of `A`, at line 421, 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 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 subclass 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 [10, Section 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.

Fig. 16 has examples of this: The `package` accessible method `A.p` is not inherited by subtypes in different packages (line 416) and also not by subtypes in the same package (line 429) 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 403).

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 428) 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 415, 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 402).

Thus, the requirements for overriding are as follows:

- For a method to be overridden by another method in the same package, `package` accessibility suffices.
- For a method to be overridden by another method 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 [10, Section 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 an overriding method must be substitutable with that of the overridden method [10, Section 8.4.5]. For reference types that means that both types must conform with the rules for implicit type conversion, allowing covariant return types but limiting their type parameters to remain unaltered. It is also not permitted to override a static method with an instance method [10, Section 8.4.8.1], and the `throws` clauses of both methods must be compatible [10, Section 8.4.8.3]

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 [10, Section 8.4.8.2]. Since the definition of hiding hinges on accessibility (the hidden member must be accessible from where the hiding occurs [10, Section 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 in the accessibility of `A.m` required to preserve the overriding of `B.m` had to be complemented by an increase of accessibility of `B.m`, not to maintain overriding, but to respect subtyping.

Interestingly, Java extends this rule to static methods (i.e., accessibility of static methods hidden in subtypes must not decrease), 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 [1]. Accessibility also needs to be considered for type-related refactorings: Type generalization refactorings such as GENERALIZE DECLARED TYPE and EXTRACT INTERFACE [11] require that the supertypes and their members are accessible to the clients of the generalized type; type hierarchy refactorings such as INFER TYPE [15] and REPLACE INHERITANCE WITH DELEGATION change subclass relationships and thus may render protected entities inaccessible from the former subclass [16].

The unlocking algorithm of Section 3 also has to deal with accessibility in order to avoid constructing a qualified 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 [1]). In all these refactorings, failure to adjust access modifiers, or incorrect adjustment of access modifiers, may lead to noncompiling 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 [11]. 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., [11], [17], [18]). 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 than or equal to ($\geq_A$) 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 [18], [19], 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

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.

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:

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

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

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

Fig. 17. Auxiliary functions α, ι, and ω determining minimum required accessibility for access, inheritance, and overriding.

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

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

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 references to the declaration present in the program, but also by the existing (and nonexisting) inheritance, overriding, hiding, and subtyping relationships. However, the impact of access modifiers on compilability and meaning is seldom spelled out explicitly, but mostly scattered throughout the JLS.

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

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 Section 6.6.1 in the JLS [10]. 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 the Appendix.

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

$$\frac{\mathit{binds}(r, d)}{\langle d \rangle \geq_A \alpha(r, d)}\,(\mathrm{ACC}-1).$$

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 417 of Fig. 16, whereas the constraints for lines 404, 408, 421, and 430 are satisfied.

A second, related rule concerns inherited members:

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

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 403, 407, 416, and 429 in Fig. 16.

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

$$\frac{\mathit{overrides}(m_2, m_1)}{\langle m_1 \rangle \geq_A \omega(m_2, m_1)}\,(\mathrm{OVRPRES}).$$

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.

Preserving override relationships plays a key role in maintaining program behavior when dynamic dispatch arises. In Java, dynamic binding only occurs in case of overriding methods, where the virtual machine selects the method to invoke according to the runtime type of the receiver object. If both (static) name bindings and overriding relationships are preserved, dynamic dispatch behavior is also preserved.

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 [10, Section 8.4.8.3]. We define a predicate *may-hide* to check these conditions (see the Appendix), and use it to define a constraint rule (HID) that lowers the accessibility of m_1 if necessary to prevent invalid hiding:

$$\frac{\tau(m_2) <_T \tau(m_1) \quad \mathit{static}(m_2) \quad \mathit{overr\text{-}eqv}(m_2, m_1) \quad \neg \mathit{may\text{-}hide}(m_2, m_1)}{\langle m_1 \rangle <_A \alpha(m_2, m_1)}\,(\mathrm{HID}).$$

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 $\neg \mathit{hides}(m_2, m_1)$ 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.

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 [10, Section 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:

```
438 class C^t1 {
439   private class B^t2 { }
440   void m^m1(↑t2 b^v1) { }
441   void n^m2() { }
442 }
443
444 interface J^t3 { class B^t4 { } }
445
446 class D^t5 extends ↑t1 implements ↑t3 {
447   D^c1(↑t1 c1^v2, ↑t5 d^v3) { ↑v2.↑m1(null); }
448   D^c2(↑t1 c2^v4, ↑t1 o^v5) {
449     ↑v4.↑m2();
450     ↑t5 d^v6 = new ↑c1(↑v4, null);
451   }
452   ↑t4 f^v7;
453 }
```

Fig. 18. The J_L version of the program in Fig. 5a.

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 $\uparrow l$ to denote a locked binding referring to a declaration labeled 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. For methods that do not override other methods, the list is empty.

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 labeled 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. 5a. 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.

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 techniques 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.

As an example, consider the program of Fig. 19b, which arises as an intermediate result while performing the EXTRACT INTERFACE refactoring of Fig. 5 at the J_L level. We start by computing accessibility constraints for this

```
454 interface I^t6 {
455   void m^m3(↑t2 b^v8);
456 }
457
458 class C^t1 implements ↑t6 {
459   private class B^t2 { }
460   void m^m1(↑t2 b^v1) overrides ↑m3 { }
461   void n^m2() { }
462 }
463
464 interface J^t3 { class B^t4 { } }
465
466 class D^t5 extends ↑t1 implements ↑t3 {
467   D^c1(↑t1 c1^v2, ↑t5 d^v3) { ↑v2.↑m1(null); }
468   D^c2(↑t1 c2^v4, ↑t1 o^v5) {
469     ↑v4.↑m2();
470     ↑t5 d^v6 = new ↑c1(↑v4, null);
471   }
472   ↑t4 f^v7;
473 }
```

(a)

```
474 interface I^t6 {
475   void m^m3(↑t2 b^v8);
476 }
477
478 class C^t1 implements ↑t6 {
479   private class B^t2 { }
480   void m^m1(↑t2 b^v1) overrides ↑m3 { }
481   void n^m2() { }
482 }
483
484 interface J^t3 { class B^t4 { } }
485
486 class D^t5 extends ↑t1 implements ↑t3 {
487   D^c1(↑t6 c1^v2, ↑t5 d^v3) { ↑v2.↑m3(null); }
488   D^c2(↑t1 c2^v4, ↑t6 o^v5) {
489     ↑v4.↑m2();
490     ↑t5 d^v6 = new ↑c1(↑v4, null);
491   }
492   ↑t4 f^v7;
493 }
```

(b)

Fig. 19. An application of EXTRACT INTERFACE on a J_L program.

```
1: procedure Translate to Java (Program p):
2: while p contains locked names do
3:    C ← accessibility constraints for p
4:    if C is unsolvable then
5:       abort
6:    S ← solution of C
7:    for all (d, a) ∈ 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
```

Fig. 20. Algorithm for translating from J_L to Java.

program, and find two unsatisfied constraints. First, since the member type t_2 is referenced in line 475 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. 5b.

Note that the constraints on t_2 could also be solved by making it `public`. Generally, it seems preferable to choose new modifiers as close to the old ones as possible, so raising the accessibility of t_2 all the way to `public` would be considered worse than making it `package`. In some cases, however, there may be no optimal solution; see, for example, rule (INH-1) in the Appendix, which might require either lowering or raising a method's accessibility.

While this simple example does not show it, unlocking of binding and solving of accessibility constraints can in general influence each other, and hence the two have to be interleaved instead of being performed in sequence. This is because our algorithm for unlocking bindings can create new locked bindings used in type casts or qualified `this` expressions (see, for instance, line 358 in Fig. 14), which can potentially 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 labeled 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, 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.

Fig. 20 shows our 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. If this unlocking creates more locked bindings, the process is repeated.

The example in Fig. 5 has shown that accessibility constraints make name unlocking more powerful. Adjusting accessibilities also 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 overridable by m. Conversely, if m is *not* supposed to override m' but would override it according to Java's overriding rules, rule (OVRPREV), shown in the Appendix, will constrain the accessibility of m' to prevent the overriding.

On the other hand, locked bindings also enable more flexible accessibility constraints: In the original formulation of accessibility constraints given by Steimann and Thies [17], there is a constraint rule that would bar us from raising the accessibility of t_2 in Fig. 19. 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 Section 2.

The work by Tip et al. on type-related refactorings [11] 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 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 Section 2, and see how this application of EXTRACT INTERFACE plays out in J_L. The J_L version of the input program of Fig. 5a 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. 19a. In Java, even such a simple transformation is 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 not yet 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 [11]. 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 467 to bind to m_3 instead, as shown in Fig. 19b. 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 [11], 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.

```
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)
```

Fig. 21. EXTRACT INTERFACE on J_L.

5.5 Porting Type-Related Refactorings to J_L

The informal description of EXTRACT INTERFACE for J_L in the previous section 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 on 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 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.

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

6 Implementation

We have implemented our approach in the JRRT refactoring tool [6]. 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 [8]. 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.

For the common case of refactorings that do not alter the type hierarchy, a conservative overapproximation 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 Section 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 1,400 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 *recv-type*) are directly provided by the JastAddJ front end, the implementation of the accessibility constraint generator takes less than 650 lines of code. The accessibility constraints are then translated into constraints over integers and solved using the Cream constraint solving library [21].

10. These and all following code size measurements were generated using David A. Wheeler's "SLOCCount" [20].

TABLE 1
Evaluation of RENAME on Eclipse's Test Suite

Renamed Entity	Total	Inapplicable	Missing Feature	Additional Passing Tests	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

JastAdd's declarative attribute features lend themselves well to implementing lookup and reference construction in a concise and modular fashion, but the same functionality could just as well have been implemented in any other metaprogramming system. Our main reason for choosing JastAdd was the easy integration with JastAddJ, which offers a complete frontend for Java 5.

JastAddJ does not yet support Java 6 or Java 7, and neither does JRRT. Supporting Java 6 would likely not require a major effort since there are few language-level changes from Java 5. Java 7 would not be as straightforward, but at least the reference construction framework could be extended in tandem with the JastAddJ front end.

7 Evaluation

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

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 for a subset of Java [22], 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 official test suite for the refactoring engine of Eclipse JDT 3.6.[11] 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 JRRT 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 labeled "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

11. Available from the public Eclipse CVS repository in project `org.eclipse.jdt.ui.tests.refactoring`.

TABLE 2
Quantitative Evaluation; Sizes in Thousands of Lines of Source Code

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	173	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

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 "Additional Passing Tests" encompasses test cases that Eclipse cannot handle (and which are hence supposed to be rejected), but which JRRT can refactor successfully. This includes test cases where names have to be qualified to avoid capture, which Eclipse does not attempt to do. We inspected all these tests to make sure that the refactoring performed by JRRT preserves behavior.

The final category, "Same Result," is 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 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 (subcolumn "total" of column "successful"), the number of cases that required at least one accessibility adjustment (subcolumn "acc."), and the number of cases where a name adjustment was made (subcolumn "names").[12]

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 4,398 possible refactoring applications on Tomcat, 4,396 were rejected, the most common reasons being changes in the evaluation order of field initializers (2,360 cases) and possible changes in dynamic method resolution (1,602 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 only checked that the output program still compiled.

12. 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.

TABLE 3
100 PULL UP METHOD Refactorings with Accessibility and Naming Issues in Eclipse 3.6.0 and IntelliJ IDEA 9.0.4

		ECLIPSE			INTELLIJ IDEA		
Name	cases	succ.	reject	error	succ.	reject	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

Averaging over all programs, accessibilities were adjusted in about 50.2 percent of all successful applications, names in 23.7 percent. 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 afterward.

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 nonstatic methods of the class, to a new package where it was to serve as a published interface [23]. Accessibility adjustments were needed in 27.8 percent and name adjustments in 3.6 percent of all successful applications.

Given its prototype nature, our current implementation is not ready for a full-fledged performance evaluation. Performance measurements taken during the experiments show that a single application (whether successful or not) of any of the refactorings completes, on average, within 46 seconds or fewer, even on the largest of our benchmarks, and 90 percent of all applications complete within 53 seconds.[13]

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 percent 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 IDEA showed a rather different picture. Eclipse could only successfully refactor in three cases; IDEA did somewhat better and succeeded in 22 cases. In the majority of cases—91 for Eclipse and 76 for IDEA—both tools rejected the refactoring with an error message. In six and two cases, respectively, the tools performed changes leading to uncompilable code. We could not perform a similar comparison for EXTRACT INTERFACE as Eclipse does not directly support extraction into a different package.

13. Timings obtained on a 2 GHz 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

Since our evaluation of correctness in Section 7.1 uses the Eclipse test suite, it does not show examples where Eclipse performs an incorrect refactoring that changes program behavior. Such examples are not hard to find: We have compiled a list of problematic cases concerning a range of different refactorings and made it available online [3]; JRRT handles all of them correctly.

The quantitative evaluation in Section 7.2 does not evaluate the correctness of the performed refactoring, except for checking that the refactored program still compiles: The large number of refactoring applications made it impractical to manually assess their correctness. Recent work by Soares et al. [24], [25] employs automated unit test generation to check behavior preservation of refactorings; they use their approach to compare Eclipse and JRRT, and find the latter to produce far fewer incorrect refactorings.

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 such 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.

Even with such a preview, it may still be unclear to the user *why* a certain change is necessary. We leave it to future work to provide explanations for proposed changes.

8 RELATED WORK

Almost two decades after inception of the discipline as marked by the theses of Griswold [26] and Opdyke [27], 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 [27] and Roberts [28], most previous refactoring research has relied on pre and postconditions to ensure that program behavior is preserved. However, informally justifying that a given set of conditions is sufficient to ensure behavior preservation is very difficult. Hence, many authors have worked on giving a precise semantics of the object language in which programs to be refactored are written and then formally proving that a given refactoring always preserves program semantics. For example, Borba et al. [29] use an axiomatic

semantics based on weakest preconditions, whereas Garrido and Meseguer [30] base their work on rewriting logic. While the proofs of behavior preservation are usually performed by hand, Sultana and Thompson [31] explore the use of a proof assistant in mechanically verifying the correctness of refactorings based on a reduction semantics. To keep the formal development manageable, these works all restrict themselves to very simple object languages.

Others have proposed to formulate refactorings on more high-level program representations to make it easier to both describe refactorings and reason about their correctness. Griswold [26] employs program dependence graphs (PDGs), which incorporate information about control and data flow dependencies, and introduces meaning-preserving graph transformation rules to reason about behavior preservation. The main difficulty with such an approach is mediating between the transformation of the program's AST and the corresponding transformation of the PDG; in Griswold's case this is relatively easy since he bases his work on (a first-order subset of) Scheme, where this correspondence is more straightforward than in most languages.

Similarly, Mens et al. [32] formulate refactorings as graph transformations and formalize certain aspects of a program's behavior as properties of the graph representing it. They do not aim to capture the full semantics in this way, however, but only specific properties such as preservation of method call targets. In a similar vein, Schäfer et al. formulate refactorings in terms of their effect on static program dependencies, of which name bindings are one example [8], [33], [34]. JunGL [35], a domain-specific language for implementing refactorings, also provides a graph-based view of the program to refactor, but its focus is mostly on enabling succinct implementation of new refactorings, rather than reasoning about their correctness.

Several authors have considered composition of refactorings, where small, general-purpose refactorings are composed to yield larger, more special-purpose refactorings [33], [36], [37]. A central question here is how to compose the preconditions of constituent refactorings to obtain preconditions for the composite refactoring, and how to reuse analysis results between different stages of the refactoring. While this issue is mostly orthogonal to the problems of naming and accessibility considered in this work, using a representation like J_L could still help since it decreases the number of explicit preconditions to consider and even allows intermediate programs that do not conform to the object language's naming or accessibility discipline as long as the final program can be translated back to the object language, thus avoiding spurious rejections.

Other work considered various programming language features such as class hierarchies [11], [16], generics [38], [39], [40], and design patterns [41], [42]. Most of these works focus on a single language feature in isolation, and do not consider interactions between different features.

A more pragmatic way of enhancing the reliability of refactoring tools is automated testing. ASTGen [43] is a combinator-based program generation tool that has been used to find bugs in the refactoring tools of Eclipse and NetBeans. ASTGen is used to generate input programs to the tool, and different kinds of oracles are used to determine whether the refactoring performed by the tool is behavior preserving. In particular, one oracle checks for compilability of the output program as a coarse indicator of correctness, like we did in our quantitative evaluation.

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 [10, §8.4.8]
inherits $\subseteq T \times M$	*inherits*(t, m) holds if type t inherits its 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 [10, §8.4.8.3]
method $\subseteq M$	*method*(m) holds if m is a method
overr-eqv $\subseteq M \times M$	*overr-eqv*(m_1, m_2) holds if methods m_1 and m_2 have override equivalent signatures [10, §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 [10, §8.4.8.1]) and there exists no other method m_3 that overrides m_1 and that is overridden by m_2
recv-type $\subseteq R \times T$	for a field access or method invocation r, *recv-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

Fig. 22. Program queries used in the rules of Fig. 23.

More recently, Soares et al. [24], [25] have suggested techniques for automated behavioral testing of refactoring tools. They also employ a program generator to generate input programs to refactor, but instead of using special-purpose oracles, they automatically create unit tests to find behavioral differences between the original and the refactored program. One subtle issue here is that they do not refactor the unit tests together with the program, which

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

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

$$\frac{\mathit{recv\text{-}type}(r,t)}{\langle t\rangle \geq_A \alpha(r,t)}\ (\textsc{Acc-3})$$

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

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

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

$$\frac{\tau(m_2) <_T \tau(m_1) \quad \mathit{overr\text{-}eqv}(m_2,m_1) \quad \neg\mathit{static}(m_2) \quad \neg\mathit{overrides}(m_2,m_1)}{\langle m_1\rangle <_A \omega(m_2,m_1)}\ (\textsc{OvrPrev})$$

$$\frac{\tau(m_2) <_T \tau(m_1) \quad \mathit{overr\text{-}eqv}(m_2,m_1) \quad \mathit{static}(m_2) \quad \neg\mathit{may\text{-}hide}(m_2,m_1)}{\langle m_1\rangle <_A \alpha(m_2,m_1)}\ (\textsc{Hid})$$

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

$$\frac{\tau(m_1)=i \quad \mathit{interface}(i) \quad \mathit{inherits}(c,m_1) \quad \mathit{inherits}(c,m_2) \quad \neg\mathit{abstract}(m_2) \quad \mathit{overr\text{-}eqv}(m_2,m_1)}{\langle m_2\rangle = \mathit{public} \vee \langle m_2\rangle <_A \iota(c,\tau(m_2))}\ (\textsc{Inh-1})$$

$$\frac{\tau(m_1)=i \quad \mathit{interface}(i) \quad \mathit{inherits}(c,m_1) \quad \mathit{inherits}(c,m_2) \quad \neg\mathit{abstract}(m_2) \quad \neg\mathit{abstract}(c) \quad \mathit{overr\text{-}eqv}(m_2,m_1)}{\langle m_2\rangle = \mathit{public}}\ (\textsc{Inh-2})$$

$$\frac{\tau(m_1)=i \quad \mathit{interface}(i) \quad c <_T i \quad c <_T \tau(m_2) \quad \mathit{static}(m_2) \quad \mathit{overr\text{-}eqv}(m_2,m_1)}{\langle m_2\rangle <_A \iota(c,\tau(m_2))}\ (\textsc{Inh-3})$$

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

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

$$\frac{\mathit{interface}(\tau(m))}{\langle m\rangle = \texttt{public}}\ (\textsc{IMember}) \qquad \frac{\mathit{constructor}(c) \quad \mathit{enumeration}(\tau(c))}{\langle m\rangle = \texttt{private}}\ (\textsc{EnumCtor})$$

$$\frac{\mathit{main\text{-}method}(m)}{\langle m\rangle = \texttt{public}}\ (\textsc{MainMeth})$$

Fig. 23. The accessibility constraint rules.

may lead to false positives since refactoring tools generally assume the whole program to be given. Nevertheless, they were able to find genuine bugs in Eclipse, NetBeans, and an earlier version of JRRT, the latter being due to a mistake in the implementation of the reference construction algorithm of Section 3. Overall, they found significantly fewer bugs in JRRT than in Eclipse and NetBeans, suggesting that our techniques increase robustness.

In recent work, Steimann et al. [18] 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 and Future Work

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 to ensure that the transformed program has the same behavior as the input program, or 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 that bind to a target declaration, and 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 popular refactoring engines.

As our work shows, the name binding and accessibility rules of Java are highly complex and full of sometimes surprising quirks. On the one hand, this complexity gives the programmer a lot of flexibility in reusing names and omitting qualifiers such as `this` wherever possible. On the other hand, this flexibility strikes back as a degraded capability of statically detecting logical programming errors, and makes it easy to introduce errors when changing the program, for instance by a refactoring. A language designed for ease of refactoring would have much simpler binding rules at the price of reduced freedom in choosing names.

There are, of course, many other problems besides naming and accessibility that refactoring engines have to handle and which J_L does not solve. Examples include control and data flow properties that have to be preserved during method extraction [33] or synchronization of shared data in concurrent programs [34]. In our experience, however, these issues are largely orthogonal to naming and can be dealt with separately.

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 and access control-free program representation such as J_L.

Appendix

Accessibility Constraint Rules

The constraint rules governing access modifiers are defined as shown in Fig. 23; they rely on the program queries listed in Figs. 17 and 22.

We briefly explain the most important rules:

- (ACC-1): This is the basic rule for type and member access [10, Section 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` [10, Section 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 [10, Section 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 [10, Section 8.4.8.4].
- The remaining rules ensure various other accessibility requirements found in the language specification.

References

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

[2] T. Mens and T. Tourwé, "A Survey of Software Refactoring," *IEEE Trans. Software Eng.*, vol. 30, no. 2, pp. 126-139, Feb. 2004.

[3] M. Schäfer, T. Ekman, R. Ettinger, and M. Verbaere, "Refactoring Bugs," http://code.google.com/p/jrrt/wiki/RefactoringBugs, 2011.

[4] E. Foundation, "Eclipse 3.6 JDT," http://www.eclipse.org/jdt, 2011.

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

[6] M. Schäfer, T. Ekman, and A. Thies, "JRRT—JastAdd Refactoring Tools," http://code.google.com/p/jrrt, 2011.

[7] T. Ekman and G. Hedin, "The JastAdd Extensible Java Compiler," *Proc. 22nd Ann. ACM SIGPLAN Conf. Object-Oriented Programming Systems and Applications,* pp. 1-18, 2007.

[8] M. Schäfer and O. de Moor, "Specifying and Implementing Refactorings," *Proc. ACM Int'l Conf. Object-Oriented Programming Systems and Applications,* M. Rinard, ed., 2010.

[9] E. Bodden, A. Sewe, J. Sinschek, H. Oueslati, and M. Mezini, "Taming Reflection: Aiding Static Analysis in the Presence of Reflection and Custom Class Loaders," *Proc. Int'l Conf. Software Eng.*, pp. 241-250, May 2011.

[10] J. Gosling, B. Joy, G. Steele, and G. Bracha, *The Java Language Specification,* third ed. Addison Wesley, 2005.

[11] F. Tip, A. Kieżun, and D. Bäumer, "Refactoring for Generalization Using Type Constraints," *Proc. 18th Ann. ACM SIGPLAN Conf. Object-Oriented Programming Systems and Applications,* pp. 13-26, 2003.
[12] F. Tip, R.M. Fuhrer, A. Kieżun, M.D. Ernst, I. Balaban, and B. De Sutter, "Refactoring Using Type Constraints," *ACM Trans. Programming Languages and Systems,* vol. 33, no. 3, p. 9, 2011.
[13] T. Ekman and G. Hedin, "Modular Name Analysis for Java Using JastAdd," *Proc. Int'l Conf. Generative and Transformational Techniques in Software Eng.*. pp. 422-436, 2006.
[14] T. Ekman and G. Hedin, "The JastAdd System—Modular Extensible Compiler Construction," *Science of Computer Programming,* vol. 69, nos. 1-3, pp. 14-26, 2007.
[15] F. Steimann, "The Infer Type Refactoring and Its Use for Interface-Based Programming," *J. Object Technology,* vol. 6, no. 2, pp. 99-120, 2007.
[16] H. Kegel and F. Steimann, "Systematically Refactoring Inheritance to Delegation in Java," *Proc. ACM/IEEE 30th Int'l Conf. Software Eng.,* pp. 431-440, 2008.
[17] F. Steimann and A. Thies, "From Public to Private to Absent: Refactoring Java Programs under Constrained Accessibility," *Proc. European Conf. Object-Oriented Programming,* S. Drossopoulou, ed., pp. 419-443, 2009.
[18] F. Steimann, C. Kollee, and J.von Pilgrim, "A Refactoring Constraint Language and Its Application," *Proc. European Conf. Object-Oriented Programming,* 2011.
[19] F. Steimann, "Constraint-Based Model Refactoring," *Proc. 14th Int'l Conf. Model Driven Eng. Languages and Systems,* J. Whittle, Clark, and T. Kühne, eds., 2011.
[20] D.A. Wheeler, "SLOCCount," http://www.dwheeler.com/sloccount/, 2006.
[21] N. Tamura, "Cream: Class Library for Constraint Programming in Java," http://bach.istc.kobe-u.ac.jp/cream. 2009.
[22] N. Schirmer, "Analysing the Java Package/Access Concepts in Isabelle/HOL," *Concurrency—Practice and Experience,* vol. 16, no. 7, pp. 689-706, 2004.
[23] M. Fowler, "Public versus Published Interfaces," *IEEE Software,* vol. 19, no. 2, pp. 18-19, Mar./Apr. 2002.
[24] G. Soares, R. Gheyi, D. Serey, and T. Massoni, "Making Program Refactoring Safer," *IEEE Software,* vol. 27, no. 4,, pp. 52-57, July/Aug. 2010.
[25] G. Soares, R. Gheyi, and T. Massoni, "Automated Behavioral Testing of Refactoring Engines," *IEEE Trans. Software Eng.,* doi: 10.1109/TSE.2012.19, 2012.
[26] W.G. Griswold, "Program Restructuring as an Aid to Software Maintenance," PhD thesis, Univ. Washington, 1991.
[27] W.F. Opdyke, "Refactoring Object-Oriented Frameworks," PhD thesis, Univ. Illinois at Urbana-Champaign, 1992.
[28] D.B. Roberts, "Practical Analysis for Refactoring," PhD thesis, Univ. Illinois at Urbana-Champaign, 1999.
[29] P. Borba, A. Sampaio, A. Cavalcanti, and M. Cornélio, "Algebraic Reasoning for Object-Oriented Programming," *Science of Computer Programming,* vol. 52, pp. 53-100, 2004.
[30] A. Garrido and J. Meseguer, "Formal Specification and Verification of Java Refactorings," *Proc. IEEE Sixth Int'l Workshop Source Code Analysis and Manipulation,* pp. 165-174, 2006.
[31] N. Sultana and S. Thompson, "Mechanical Verification of Refactorings," *Proc. ACM SIGPLAN Symp. Partial Evaluation and Semantics-Based Program Manipulation,* pp. 51-60, 2008.
[32] T. Mens, N.V. Eetvelde, S. Demeyer, and D. Janssens, "Formalizing Refactorings with Graph Transformations," *J. Software Maintenance,* vol. 17, no. 4, pp. 247-276, 2005.
[33] M. Schäfer, M. Verbaere, T. Ekman, and O. deMoor, "Stepping Stones over the Refactoring Rubicon—Lightweight Language Extensions to Easily Realise Refactorings," *Proc. European Conf. Object-Oriented Programming,* S. Drossopoulou, ed., pp. 369-393, 2009.
[34] M. Schäfer, J. Dolby, M. Sridharan, F. Tip, and E. Torlak, "Correct Refactoring of Concurrent Java Code," *Proc. European Conf. Object-Oriented Programming,* T. D 'Hondt, ed., 2010.
[35] M. Verbaere, R. Ettinger, and O. de Moor, "JunGL: A Scripting Language for Refactoring," *Proc. Int'l Conf. Software Eng.,* D. Rombach and M.L. Soffa, eds., pp. 172-181, 2006.
[36] M.O. Cinnéide, "Automated Application of Design Patterns: A Refactoring Approach," PhD thesis, Univ. of Dublin, Trinity College, 2000.
[37] G. Kniesel and H. Koch, "Static Composition of Refactorings," *Science of Computer Programming,* vol. 52, no. 1-3, pp. 9-51, 2004.
[38] A. Donovan, A. Kieżun, M. Tschantz, and M.D. Ernst, "Converting Java Programs to Use Generic Libraries," *Proc. 19th ACM SIGPLAN Conf. Object-Oriented Programming Systems and Applications,* pp. 15-34, 2004.
[39] D. von Dincklage and A. Diwan, "Converting Java Classes to Use Generics," *Proc. 19th ACM SIGPLAN Conf. Object-Oriented Programming Systems and Applications,* pp. 1-14, 2004.
[40] R.M. Fuhrer, F. Tip, A. Kieżun, J. Dolby, and M. Keller, "Efficiently Refactoring Java Applications to Use Generic Libraries" *Proc. European Conf. Object-Oriented Programming,* 2005.
[41] L. Tokuda and D. Batory, "Evolving Object-Oriented Designs with Refactorings," *Automated Software Eng.,* vol. 8, no. 1, pp. 89-120, Jan. 2001.
[42] J. Kerievsky, *Refactoring to Patterns.* Addison Wesley, 2005.
[43] B. Daniel, D. Dig, K. Garcia, and D. Marinov, "Automated Testing of Refactoring Engines," *Proc. Joint Meeting of the European Software Eng. Conf. and Symp. Foundations of Software Eng.*. pp. 185-194, 2007.

Max Schäfer received the doctorate in computer science from the University of Oxford in 2010. He is currently a postdoctoral researcher at the IBM T.J. Watson Research Center in Hawthorne, New York. His research interests include specification and implementation of automated refactorings, automated program repair, and program analysis for dynamic programming languages.

Andreas Thies is working toward the PhD degree at the Fernuniversität in Hagen, Germany. His research interests focus on constraint-based program analysis, mainly for automated refactoring and testing.

Friedrich Steimann is head of Programming Systems at the Fernuniversität in Hagen, Germany. His group conducts research on software modeling, programmers' productivity, and object-oriented development tools.

Frank Tip received the PhD degree from the University of Amsterdam in 1995. After spending about 17 years at IBM Research, he recently joined the David R. Cheriton School of Computer Science at the University of Waterloo, Canada, as a David R. Cheriton Chair in Software Systems. His current research interests include: refactoring, test generation, fault localization and repair, analysis of web applications, and data-centric synchronization.