Specifying and Implementing Refactorings

Max Schäfer Oege de Moor
Oxford University Computing Laboratory, UK
{max.schaefer, oege.de.moor}@comlab.ox.ac.uk

Abstract

Modern IDEs for object-oriented languages like Java provide support for a basic set of simple automated refactorings whose behaviour is easy to describe intuitively. It is, however, surprisingly difficult to specify their behaviour in detail. In particular, the popular precondition-based approach tends to produce somewhat unwieldy descriptions if advanced features of the object language are taken into account. This has resulted in refactoring implementations that are complex, hard to understand, and even harder to maintain, yet these implementations themselves are the only precise "specification" of many refactorings. We have in past work advocated a different approach based on several complementary notions of dependencies that guide the implementation, and on the concept of microrefactorings that structure it. We show in this work that these concepts are powerful enough to provide high-level specifications of many of the refactorings implemented in Eclipse. These specifications are precise enough to serve as the basis of a clean-room reimplementation of these refactorings that is very compact, yet matches Eclipse's for features and outperforms it in terms of correctness.

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

General Terms Languages

Keywords Refactoring, specification, language extensions

1. Introduction

A refactoring can only be useful if it is easy to understand for the programmer, and if it encapsulates operations that programmers find themselves doing over and over again. Indeed, most basic refactorings available in modern IDEs are easily explained in terms of one or two simple examples.

OOPSLA/SPLASH'10, October 17–21, 2010, Reno/Tahoe, Nevada, USA.

But describing a refactoring *precisely* is a perhaps unexpectedly difficult task. The complexities of real-world programming languages conspire to make it a formidable task to account for all corner cases and always produce output programs that are both syntactically correct and semantically equivalent to the input program.

Popular textbooks on refactorings [Fow00, Ker05] hence tend to gloss over the finer details and advise the programmer to rely on frequent testing to ensure behaviour preservation. Even the more precise descriptions in Opdyke's classic work [Opd92] are only loosely based on a simplified subset of C++ and make no attempt to account for all eventualities.

But it seems that the imprecision of these descriptions is directly reflected in the sometimes very low standards of correctness [SEEV10] of popular refactoring implementations like those of Eclipse and IntelliJ [Fou10, Jet10] even for languages like Java where automated refactoring tools enjoy great popularity. These implementations are hard to understand and even harder to maintain: for example, a seemingly rather straightforward bug in Eclipse's INLINE METHOD refactoring has gone unfixed for almost five years. [1]

Without precise descriptions, it is hard to answer even very simple questions about refactorings. For example, the first author recently found himself trying to judge which of Eclipse's built-in refactorings could potentially move a field access out of a `synchronized` block, which is unsafe in the face of concurrency. With existing specifications being too general and vague and implementations being too complicated and lowlevel, there is no good source for gaining the sort of in-depth understanding of individual refactorings that is needed to answer such questions.

This work takes a step towards remedying the situation by providing highlevel specifications of common refactorings that are brief and concise, yet aim to be precise enough to cover all features of the Java 5 language. We furthermore present an implementation of these specifications as part of a refactoring engine based on the JastAddJ Java compiler [EH07a] and evaluate its correctness using Eclipse's internal test suite.[2]

[1] See `https://bugs.eclipse.org/bugs/show_bug.cgi?id=112100`.

[2] The implementation, including its test suite, is available for download from `http://jastadd.org/refactoring-tools`.

```
void m() {
  int[] xs = { 23, 42 };
}

void n() {
  int[] xs;
  xs = new int[] { 23, 42 };
}
```

Figure 1. Array initialisers and array creation expressions

To provide such descriptions, we choose not to follow the traditional precondition-based approach. While preconditions are valuable for specifying shallow conditions that must hold in order for the refactoring to make sense at all, they are not the right tool to ensure behaviour preservation in the face of issues related to name capture or control and data flow preservation.

As we have argued in previous work [SEdM08, SVEdM09, SDS+10], such problems are much easier to tackle if they are expressed as dependency preservation problems. For example, in many refactorings we want to ensure that name bindings do not change, or that a name binds to a given declaration. We accommodate this by introducing a name binding dependency that is tracked by the refactoring engine. After the refactoring has performed its code transformations, the engine checks whether all tracked dependencies are still present, *i.e.* whether the name still binds to the intended declaration. If this is not the case, the engine either adjusts the name to restore its binding (for instance by qualifying it to escape shadowing), or aborts the whole refactoring. Control and data flow preservation is treated in a very similar manner.

Besides these "deep" issues, refactoring engines also have to account for a host of much shallower issues related to irregularities or idiosyncrasies of the object language. For example, Java allows so-called array initialisers to occur in variable initialisations. They are, however, not first-class expressions and cannot occur in many other places, where they have to be elaborated into slightly more complex array creation expressions.

An example is given in Fig. 1, which shows two methods `m` and `n` that both declare a local integer array `xs` and initialise it. But while `m` initialises `xs` directly in its declaration and can hence use an array initialiser, `n` initialises the variable using an explicit assignment and has to use an array creation expression. Obviously, the methods are semantically equivalent, and indeed the same bytecode is generated for both.

Any refactoring that moves an expression from a variable initialisation to some other place in the code has to be aware of this problem and wrap array initialisers accordingly. It would certainly be much preferable to encapsulate this issue and handle it once and for all.

To address this kind of problem, we use the approach of *lightweight language extensions* [SVEdM09]: while our refactoring implementations work on Java programs and produce Java programs as output, intermediate steps can work on a richer language that provides additional features to facilitate implementation, which are then translated away into pure Java. These features are lightweight extensions in the sense that they are transparent to the user of the refactoring tool and never appear in the refactored program. A particularly simple extension is to make array initialisers first-class expressions; translating this extension away is, of course, achieved by wrapping them into array creation expressions.

Other refactorings require more complicated extensions, such as anonymous methods [SVEdM09] or `with` statements [SdM09]. One might then worry that every refactoring needs different language extensions, leading to a proliferation of potentially similar yet incompatible language extensions. It is one of our aims in this work to allay these fears by showing that just a handful of fairly simple extensions suffice to implement many major refactorings, and can indeed often be shared between different refactorings.

Just as many refactoring steps are more easily expressed by allowing them to output programs in an extended language, it is often convenient for refactorings to assume that their input programs are written in a restricted language that does not contain certain features that would otherwise need special treatment. For example, the `synchronized` modifier on methods in Java turns out to be rather troublesome since it hides a dependency on the implicit monitor associated with the receiver object.

Strictly speaking, this modifier is not needed, since it can be replaced by a `synchronized` block around the body of the method, which makes the dependency explicit. We can simplify our implementation by first translating the modifier away in this manner, and then reintroducing it where possible after the refactoring [SDS+10]. Indeed, the long-standing Eclipse bug alluded to above hinges on this issue.

One major benefit of using language extensions and restrictions is that it often allows us to decompose a refactoring into several smaller refactorings, so-called *microrefactorings*, that can be specified, implemented, and tested in isolation. While earlier work [SVEdM09] has shown that such a decomposition is possible for certain refactorings, the present work extends this to many other refactorings.

In summary, the main contributions of this work are as follows:

- We show that the previously introduced techniques of dependency preservation, language extensions and restrictions, and microrefactorings are sufficiently powerful to give **high-level, yet precise specifications** of many refactorings that easily fit within half a page, but **handle the full Java 5 language**.

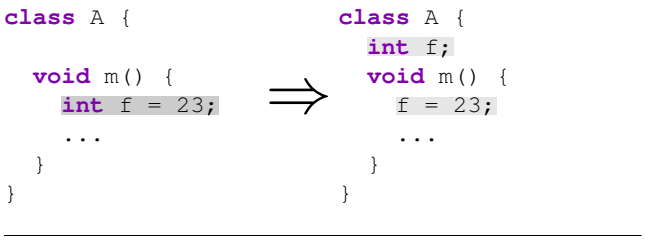

Figure 2. A simple use of PROMOTE TEMP TO FIELD

- We demonstrate on some examples that these specifications are highly **modular** and that microrefactorings and language extensions are **reusable** between refactorings.
- We present an **extensive case study** to show that the specifications give rise to **concise, high-quality implementations** of all the refactorings offered in recent versions of Eclipse[3], which perform very well on Eclipse's own test suite.

The rest of the paper is structured as follows: Section 2 introduces the relevant techniques of dependencies, language extensions, and microrefactorings with the PROMOTE TEMP TO FIELD refactoring as the running example. Section 3 explains how to use these techniques to provide specifications of refactorings and how these can be implemented; Section 4 discusses specifications for PUSH DOWN METHOD and MOVE MEMBER TYPE TO TOPLEVEL in more detail. Section 5 relates our experience with providing specifications and implementations for other Eclipse refactorings and pitches our implementation against Eclipse's on their own test suite. Section 6 puts our work in the context of related work, and Section 7 concludes.

2. Dependencies and Language Extensions

To make the presentation self-contained, this section will introduce the different kinds of dependencies used in our approach, as well as the concepts of language extensions and microrefactorings on the example of the PROMOTE TEMP TO FIELD refactoring.

The PROMOTE TEMP TO FIELD refactoring, offered by both Eclipse (as "Convert Local Variable to Field") and IntelliJ (as "Introduce Field"), converts a local variable into a field, for example as the first step of the REPLACE METHOD WITH METHOD OBJECT refactoring [Fow00]. A very simple example is shown in Fig. 2, where the local variable `f` is promoted to a field.

We use this example to establish some notational conventions: When presenting an example of how a refactoring is performed, we will always display the input program (or fragment) on the left, and the refactored program on the right, connected by a heavy arrow. For clarity, the piece of code in the original program the refactoring is applied to (the local variable in this example) is highlighted in dark grey, whereas the changed code in the resulting program is marked in light grey.

[3] We leave aside a handful of refactorings that have already been studied in the literature.

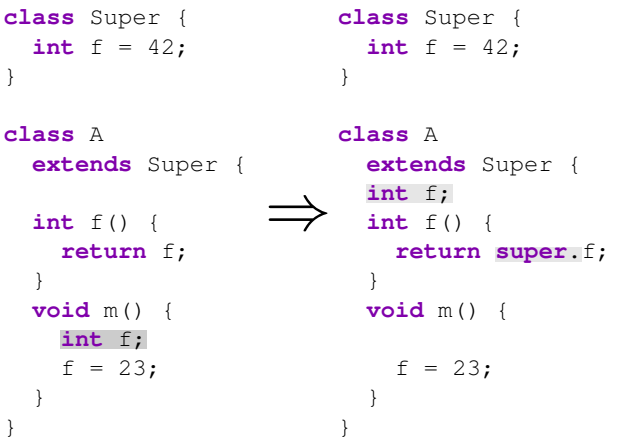

Figure 3. Example of a naming issue with PROMOTE TEMP TO FIELD

Based on this example, the refactoring is indeed easy to describe: if the local variable is initialised, turn its initialisation into an assignment; then move the declaration of the local variable into the surrounding type so that it becomes a field.

Figure 3 gives an example of a naming problem that may complicate the refactoring: here, a field `f` is inherited from the super class and used in method `f`. Promoting the local variable `f` of method `m` to a field hides the inherited field `f` within class `A`, and in particular within method `f`. The solution is, of course, to qualify `f` with **`super`** to circumvent this shadowing. Eclipse neglects to do this, while IntelliJ warns that a field of the same name is declared in the super class (which is arguably not the essence of the problem), but does not insert a qualification either.

Let us consider the general case, where we want to turn a local variable `f` into a field of class `A`. Should we always qualify every unqualified reference to any field `f` from within `A` with **`super`**? This is neither correct nor sufficient: if the reference is within a static method, **`super`** is not available; on the other hand, even references in subclasses or nested classes of `A` may be influenced by the refactoring.

Instead, we can view this as a dependency preservation problem. One necessary condition for PROMOTE TEMP TO FIELD to be behaviour preserving is certainly that all references to the former local variable `f` now bind to the introduced field `f`, and all other name bindings stay the same.

In past work [SEdM08] we have introduced a framework of *locked names* that allow us to achieve this: before promoting, we "lock" every access to any field of name `f` within the

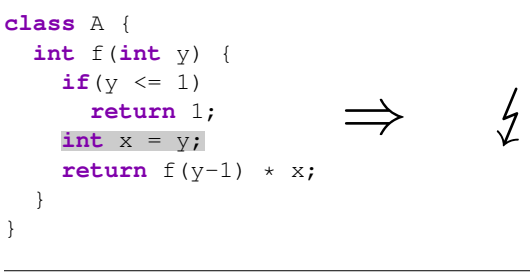

Figure 4. Example of a data flow issue with PROMOTE TEMP TO FIELD

whole program[4], *i.e.* we compute the declaration it binds to and store it along with the name. In the same vein, all references to the local variable `f` are locked to the new field.

After promoting, we check for every name whether it still binds to the declaration it was locked on, and adjust it if it does not. The details of this adjustment process are beyond the scope of the present paper; suffice it to say that in the above example we would adjust the reference to `f` by qualifying it with **`super`**, as expected.

A somewhat thornier issue is presented in Fig. 4. Here we have a function that computes $\prod_{k=1}^{y} k$ for every input `y`, storing the value of `y` (somewhat artificially) in a local variable `x`. The point of this example is that if we blindly promote `x` to a field (as Eclipse and IntelliJ both do), its value will be clobbered by the recursive invocation, and the function will instead compute $\max(2^{y-1}, 1)$. Indeed, it does not seem that `x` is a very good candidate for being turned into a field, so a refactoring tool would arguably be justified in rejecting the proposed refactoring, as indicated by the lightning symbol.

But should we always disallow promoting local variables of recursive methods? For one, it is in general undecidable whether a method is recursive in the presence of dynamic dispatch. But even if we have a conservative check for recursive invocations, disallowing promotion in the presence of recursion altogether seems overly drastic. For instance, if we slightly modify the above example by swapping the operands of the multiplication, the recursion becomes harmless and we can promote `x` without difficulty.

The solution is, again, to track dependencies. In the original program of Fig. 4, the reference to `x` in the **`return`** statement has a single reaching definition, namely the assignment to `x` in its initialisation. If we were to turn `x` into a field, it would pick up an additional reaching definition from the recursive invocation of `f`, whereas reaching definitions would remain unchanged in the variant with swapped multiplicands.

So we introduce a *flow dependency* from every variable to every one of its reaching definitions, and require, just as for name bindings, that flow dependencies for accesses to the promoted local variable are preserved. But while we can often adjust a name to repair a broken name binding, this is generally hard to do for flow dependencies. We choose the simpler option of just aborting the refactoring if flow dependencies change.

Finally, even if naming and flow dependencies are preserved, PROMOTE TEMP TO FIELD can still change the behaviour of multi-threaded programs, since it does nothing to prevent multiple threads from concurrently accessing and updating the promoted variable. Two possible solutions to this problem are as follows:

- Make the method from which the variable was promoted **`synchronized`**. Since even the promoted variable is still only accessed from within that method, we can thus prevent multiple threads from accessing it concurrently. However, this solution drastically changes the concurrent behaviour of the program and might easily introduce deadlocks.
- Wrap the promoted field into a `ThreadLocal`, and then use an escape analysis to determine whether the wrapper can safely be removed. This requires rewriting accesses to the variable to go through the `ThreadLocal` API.

The second solution is obviously preferable, and is again amenable to dependency-based reasoning. In addition to naming and flow dependencies, we keep track of *synchronisation dependencies* [SDS+10] which capture ordering constraints between synchronisation constructs and accesses to shared state. In particular, accesses to local variables generate no constraints (since local variables are never shared between threads), and neither do `ThreadLocal` objects (which are not shared either), so from a concurrency perspective the refactoring becomes a no-op. For the sake of simplicity we will not elaborate this solution in detail. Incidentally, neither Eclipse nor IntelliJ try to address the problem of concurrency in their implementations of PROMOTE TEMP TO FIELD.

With the deep issues related to naming, control and data flow, and concurrency taken care of, it remains to tackle more shallow problems engendered by the syntactic peculiarities of our object language Java.

In the first step of our present refactoring, we need to split off any initialiser of the variable to be promoted into an independent assignment. As discussed earlier, the result of this transformation could contain first-class array initialisers. Also note that the variable could be declared together with several other variables in a compound declaration like this:

```
int x, y = f(), z = g();
```

If we want to promote variable `y` to a field, we should first translate this fragment into the equivalent

```
int x;
int y = f();
int z = g();
```

[4] Names other than `f` obviously will not change their binding. A performance-conscious implementation may, of course, try to limit the scope in which to lock names.

before splitting off the assignment and proceeding as above. It would not do to insert the assignment after the entire compound declaration, since this might change the order in which initialising expressions are evaluated, possibly changing behaviour. Converting between compound declarations and their expansion is, of course, easy to do, so we can simply regard it as a language restriction and formulate our refactoring on the restricted language of Java without compound declarations.

Based on the framework of dependency preservation and language extensions and restrictions, we can now give an informal, yet precise description of the refactoring. For modularity and reusability, we split the refactoring into four steps, three of which are themselves microrefactorings:

1. SPLIT DECLARATION. Given a local variable declaration d for a variable `x` with initialising expression `e` (where d must not be compound), ensure that d is a statement in a block (not, for example, the initialising statement of a `for` loop). Remove its initialiser and instead insert a new statement a of the form `x = e;` after d, where the name `x` is locked to d. The output of this microrefactoring may make use of first-class array initialisers.
2. INSERT FIELD. Insert a declaration d' for a field `x` with the same type as the local variable `x` into the surrounding class. If the local variable occurs in a static context, the field should be static as well. Check that there is not already a field with the same name in the same class, and ensure that bindings are preserved.
3. For every use of the local variable `x`, lock its flow dependencies, and make its name binding locked onto d'.
4. REMOVE DECLARATION. If there are no remaining references to d, delete it.

3. Specification and Implementation

Prose descriptions like that of PROMOTE TEMP TO FIELD in the previous section tend to be quite hard to read, so we will introduce a number of conventions to enable us to cast our descriptions into more concise pseudocode. This will allow us to successfully tackle several more complicated refactorings.

Algorithm 1 corresponds to the prose description of PROMOTE TEMP TO FIELD given before. We specify refactorings as imperative procedures that modify the input program's abstract syntax tree. They may take arguments (such as argument d of PROMOTE TEMP TO FIELD) and return results (although this one does not). Both arguments and results are given informal types, for example d is supposed to be an AST node representing a local variable declaration. A full list of the node types used in this paper is given in Fig. 7 in Appendix A, where we also summarise syntactic conventions used in our pseudocode specifications.

As the first part of the specification, we list the input and output language of every refactoring, indicating whether it requires any restrictions on the former or extensions of the latter. For this refactoring, for instance, the output program may still contain locked name bindings and flow dependencies, which can be eliminated as described in the previous section.

Algorithm 1 PROMOTE TEMP TO FIELD(d : *LocalVar*)

Input Language: Java
Output Language: Java ∪ locked dependencies

```
1: ⌊SPLIT DECLARATION⌋(d)
2: d′ ← new private field of same type and name as d
3: make d′ static if d is in static context
4: ⌊INSERT FIELD⌋(hostType(d), d′)
5: for all uses u of d do
6:     lock u onto d′
7:     lock reaching definitions of u
8: REMOVE DECL(d)
```

Since this elimination process is not refactoring-specific, we do not consider it part of the specification proper. Indeed, as we shall see, it is sometimes useful to preserve extended features in the output program if the refactoring is invoked as part of a larger refactoring. Hence we leave it up to the caller whether language extensions in the output program should be eliminated or not.

The main body of the implementation closely follows the prose description given above: first, we invoke the microrefactoring SPLIT DECLARATION to strip off d's initialiser, if any. We put floor brackets $\lfloor\cdot\rfloor$ around the invocation of this microrefactoring to indicate that any language extensions in the output program it produces should be immediately eliminated.

We then construct a new node d' corresponding to the new field, which has the same name and type as d (an implementation may need to use the locked naming framework to build a concrete type access that binds to this type).

If d appears in a static context (*i.e.*, either in a static method or a static initialiser), the promoted field is made static as well to ensure that it is accessible. Then we invoke the microrefactoring INSERT FIELD to insert the promoted field into the closest lexically enclosing type around d, which we assume to be computed by the utility function `hostType`. Again, we require any language extensions to be eliminated.

As we shall see, INSERT FIELD may fail if it cannot perform its assigned task, for instance if there already is a field of the same name in the type. By default this means that the whole refactoring fails; an actual implementation would need to provide descriptive error messages and roll back any changes to the syntax tree.

Now we lock every use of d onto the field d' and also lock its flow dependencies, before finally invoking REMOVE DECL to remove the old declaration d if possible. Finally,

the remaining locked names and flow dependencies can be unlocked, which may cause the refactoring to fail.

Let us now consider the constituent microrefactorings of PROMOTE TEMP TO FIELD. The first one, SPLIT DECLARATION is specified as Alg. 2. It also takes a *LocalVar* as argument, and its output program may contain locked names and first-class array initialisers. As mentioned above, we restrict the input programs to this microrefactoring to disallow compound declarations that declare several variables at once.

In contrast to language extensions in the output program, which may be allowed to persist after the end of a microrefactoring, language restrictions are always enforced when a refactoring begins to execute. We leave this implicit in the specification, since the input language specification makes it clear which restrictions to enforce.

If the declaration d does not have an initialiser, SPLIT DECLARATION does not need to do anything. Otherwise, it constructs a new assignment a that assigns the initialising expression of d to the variable declared in d, inserts it as the next statement after d (or fails if d is not immediately surrounded by a block), and then removes its initialiser.

In spite of its simplicity, SPLIT DECLARATION is an important building block for other refactorings; besides PROMOTE TEMP TO FIELD, it is also used in the implementation of EXTRACT METHOD and INLINE TEMP [SVEdM09].

Algorithm 2 SPLIT DECLARATION(d : *LocalVar*)

Input Language: Java $\setminus$ compound declarations
Output Language: Java $\cup$ locked names, first-class array init

1: **if** d has initialiser **then**
2: $x \leftarrow$ variable declared in d
3: $a \leftarrow$ new assignment from initialiser of d to x
4: insert a as statement after d
5: remove initialiser of d

The next refactoring is INSERT FIELD, which inserts the declaration d of a field into a type declaration T, ensuring that the output is syntactically valid and the binding structure is undisturbed.

It uses assertions to check several preconditions; if any of them are violated, the refactoring cannot be meaningfully executed and is aborted. In particular, it checks that no other field of the same name is already declared in T, which would lead to a syntactically incorrect program; that d does not have an initialiser, since an initialiser might have side effects that change program behaviour; and that we are not trying to insert a static field that is not a compile-time constant into an inner class, which is disallowed by the language specification.

Note that the latter two checks are unnecessary in the context of the PROMOTE TEMP TO FIELD refactoring (the promoted field never has an initialiser, and if the field is static, then so is the method it is promoted from, so the host type cannot have been an inner class to begin with), but we include them for modularity and since we want INSERT FIELD to be a behaviour-preserving refactoring on its own.

If all the checks pass, the refactoring locks all accesses to types or variables with the same name as d: we assume that `name` gives us the name of a declaration, and that `lockNames` performs the locking. Finally, the field is inserted into the syntax tree of T.

Like SPLIT DECLARATION, INSERT FIELD is quite versatile, and is also used as part of the EXTRACT CONSTANT, MOVE MEMBERS, and MOVE MEMBER TYPE TO TOPLEVEL refactorings.

Algorithm 3 INSERT FIELD(T : *ClassOrInterface*, d : *Field*)

Input Language: Java
Output Language: Java $\cup$ locked names

1: **assert** T has no local field with same name as d
2: **assert** d has no initialiser
3: **assert** if T is inner and d is static, then d is a constant
4: `lockNames(name(`d`))`
5: insert field d into T

The final microrefactoring REMOVE DECL (which also plays a role in INLINE TEMP) is now easily described as Alg. 4: If d has no initialiser and is not referenced anywhere, we remove it from the AST; otherwise we do nothing.

Algorithm 4 REMOVE DECL(d : *LocalVar*)

Input Language: Java $\setminus$ compound declarations
Output Language: Java

1: **if** d has no initialiser and is not used **then**
2: remove d

Note that our specification of PROMOTE TEMP TO FIELD does not allow for the promoted local variable's initialiser to be promoted into an initialiser of the field. Doing so not only requires moving an expression across method boundaries, which is hardly ever behaviour preserving, but also might change the order in which assignments to the variable occur. Eclipse, which offers this feature, performs almost no checks to guarantee behaviour preservation in this case, which we consider unacceptable.

While the specifications we have given so far are in pseudocode, they are not very difficult to implement. Our own implementation of PROMOTE TEMP TO FIELD and the other refactorings discussed in the sequel is based on the JastAddJ compiler frontend, which provides the general AST infrastructure and some basic analysis capabilities, such as name lookup and control flow analysis. The required utility functions are either already provided by the compiler frontend or are very easy to implement on top of it. Earlier work has already discussed how we implement the dependency tracking

```
public void VariableDeclaration
  .promoteToField()
{
  split();
  Modifiers mods = new Modifiers("private");
  if(inStaticContext())
    mods.addModifier("static");
  TypeAccess ta = type().createLockedAccess();
  FieldDeclaration f
    = new FieldDeclaration(mods, ta, name());
  hostType().insertField(f);
  for(VarAccess va : uses()) {
    va.lock(f);
    va.lockReachingDefs();
  }
  flushCaches();
  remove();
}

public void VariableDeclaration
  .doPromoteToField()
{
  Program root = programRoot();
  promoteToField();
  root.eliminate(LOCKED_NAMES, LOCKED_FLOW);
}
```

Figure 5. JastAdd implementation of PROMOTE TEMP TO FIELD

framework used for name binding, flow, and synchronisation preservation. Utilising this framework, the implementation of a given refactoring can closely follow its pseudocode specification and is generally not much longer.

As an example, we show our implementation of PROMOTE TEMP TO FIELD in Fig. 5. The implementation, just like the JastAddJ compiler it is based on, is written in JastAdd [EH07b], an attribute grammar system that extends Java with support for circular reference attribute grammars.

The refactoring is implemented as a method `promoteToField` on class `VariableDeclaration` using an inter-type declaration. We also provide a wrapper method `doPromoteField`, which simply calls method `promoteToField` and then eliminates language extensions from the resulting program. Extensions are implemented as visitors (`LOCKED_NAMES` and `LOCKED_DATAFLOW`) that traverse the syntax tree and rewrite it into plain Java.

Method `promoteToField` follows the specification in Alg. 1 quite closely, in particular the three subrefactorings are invoked as methods `split`, `insertField`, and `remove`. The construction of field `f` is somewhat more verbose than in the pseudocode, since we need to explicitly construct the syntax tree nodes that the declaration is composed of. Note in particular that we use method `createLockedAccess`, which is part of the naming framework, to construct a locked access to the field's type.

The only part of the implementation that does not immediately correspond to the specification is the call to `flushCaches` in the penultimate line. This is where the underlying JastAdd implementation shines through: the information computed by attributes such as `type` or `uses` is automatically cached, and needs to be flushed manually whenever the syntax tree has changed. In this example, the value of attribute `uses` has become stale, since we have changed all uses of the local variable being promoted to bind to the `f`. If we did not flush the attribute's value, REMOVE DECL would conclude that references to the variable still exist and refrain from deleting it.

Altogether, eight lines of pseudocode specification translate into about 20 lines of implementation, which is a reasonable ratio given the verbosity of Java syntax.

4. PUSH DOWN METHOD and MOVE MEMBER TYPE TO TOPLEVEL

While PROMOTE TEMP TO FIELD is an important tool, it is a fairly simple and small-scale refactoring. In this section we show that the techniques used for its specification in the previous section carry over to more complicated refactorings by giving specifications of PUSH DOWN METHOD and MOVE MEMBER TYPE TO TOPLEVEL. The next section will then provide further empirical consolidation by discussing, albeit not in as much detail, many other refactorings.

4.1 PUSH DOWN METHOD

The PUSH DOWN METHOD refactoring pushes a method m from a class A to all its subclasses $B_1, \ldots, B_n$; it then either removes the definition of m from A or turns it into an abstract method.

A simple example is shown in Fig. 6, which demonstrates how we can view PUSH DOWN METHOD as being composed of three simpler operations: We first introduce trivial overriding methods into each of the subclasses which just invoke the overridden method through `super` calls, then we inline these calls, and finally remove the original method. Where it is impossible to remove the method outright, we can instead turn it into an abstract method.

The beauty of this approach lies in devolving the perhaps most tricky part of the refactoring, actually copying the code from the superclass to the subclass without changing its meaning, to the well-known INLINE METHOD refactoring. All that remains is to implement the first step, which is itself a refactoring we call TRIVIALLY OVERRIDE, and the third step, which is a choice between the two refactorings REMOVE METHOD and MAKE METHOD ABSTRACT. Note, however, that the decomposition we have given here only works for virtual methods (*i.e.* non-`private` instance methods). We will briefly consider the case of non-virtual methods below.

We start with the specification of TRIVIALLY OVERRIDE, given in Alg. 5. It takes the method m to produce an override

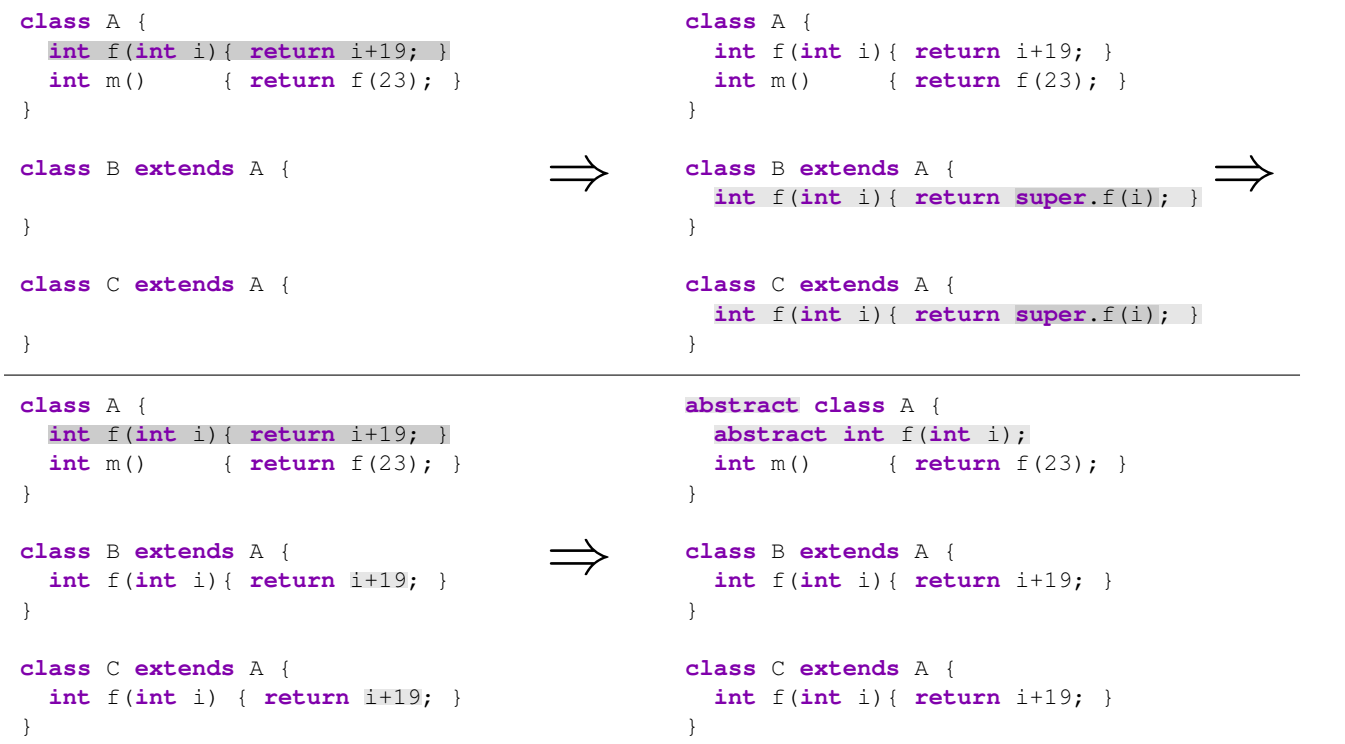

Figure 6. Applying PUSH DOWN VIRTUAL METHOD to push `f` from `A` to `B` and `C`

for and the type B in which to insert the overriding method, and returns the node corresponding to the created **`super`** call. As we shall see, such a call is not created in all cases, so the refactoring has return type `option` *MethodCall* and returns either `None` (if no call was created), or `Some` c, where c is the created call.

Algorithm 5 TRIVIALLY OVERRIDE(B : *Type*, m : *VirtualMethod*) : `option` *MethodCall*

Input Language: Java \ implicit method modifiers
Output Language: Java + locked names, **`return void`**

1: **assert** m is not **`final`**
2: **if** m not a member method of B **then**
3: **return** `None`
4: $m' \leftarrow$ copy of m with locked names
5: **if** m is **`abstract`** **then**
6: insert method m' into B
7: **return** `None`
8: **else**
9: $xs \leftarrow$ list of locked accesses to parameters of m'
10: $c \leftarrow$ **`super`**$.m(xs)$
11: set body of m' to **`return`** c;
12: insert method m' into B
13: **return** `Some` c

The refactoring first checks for several special cases. Obviously, we cannot override a final method, so this case is excluded right away.

If m is not a member method of B (for example because B already contains an overriding definition of m), the refactoring does not need to do anything and simply returns `None`.

If m is abstract we cannot introduce a **`super`** call to it, either. Instead, the refactoring creates a copy m' of m, where all names within m' are locked to ensure they keep their original bindings, and inserts it into B. As before, `None` is returned.

It remains to handle the case where a **`super`** call is actually created. We construct a list xs with locked accesses to the parameters of m', which becomes the argument list for the **`super`** call c. Finally, the body of m' is replaced by a single statement returning c, and m' is inserted as a body declaration into B.

Like many of its brethren, TRIVIALLY OVERRIDE produces programs with locked names. It also makes use of a further language extension: Recall from Fig. 6 that the refactoring constructs a method of the form

```
int f(int i) {
  return super.f(i);
}
```

Obviously, this is only well-formed if the overridden method returns a value. If it had return type **`void`**, we would

have to omit the `return`. This situation arises frequently enough to be a bit of a nuisance, so we introduce a language extension: The refactoring will on occasion produce statements of the form `return e;` where `e` is of type `void`. Such statements can simply be translated into `e; return;` (note that all expressions of type `void` are statement expressions, and hence can can form statements on their own).

As a final finesse, TRIVIALLY OVERRIDE assumes that all method modifiers in its input (or at least on method m) are explicit. For example, if B were a class and A an interface, then m would be public and abstract whether or not it carries the corresponding modifiers, but the modifiers have to be made explicit in B. Our life becomes much easier if we assume instead that the modifiers are always explicit.

Let us briefly pause to convince ourselves that this microrefactoring indeed preserves program semantics, in particular that the semantics of method calls does not change. Consider an invocation of a virtual method n on an object of dynamic type T in the original program. If T is not a descendant of A or n does not have the same signature as m, the call will be dispatched to the same method in the new program as in the original program. Otherwise, if the original call dispatched to m, the new call will either dispatch to m as well, or to the newly inserted method m', which immediately invokes m. In either case, the semantics stays unchanged.

The next step of the PUSH DOWN METHOD refactoring is to inline the `super` call if one was created. This will copy the body of m into the subclass, ensuring in particular that names in m will not inadvertently change their binding. Since INLINE METHOD has already been discussed in the context of our framework [SVEdM09], we will not give its precise specification here.

The final step is to remove m from A if possible, or else to make it abstract. The two factors to consider here are whether m is referenced outside its own body, *i.e.* whether there is any method call in the program (outside m itself) that statically resolves to m, and whether m could be dynamically called from any call site in the program (again, outside its own body). If neither is the case, the method can safely be removed. If only the former is the case, it cannot be removed altogether, but can be made abstract. In all other cases, the method must stay.

While it is straightforward to determine the set $\texttt{uses}(m)$ of all calls that statically resolve to m, determining the calls that might at runtime resolve to m is quite another matter. The coarser this analysis the oftener the refactoring will refuse to remove a pushed-down method, making it abstract instead. The specification of the refactoring, however, is independent of the details of this analysis; we simply require that its results are available as $\texttt{calls}(m)$.[5]

[5] Our implementation of `calls` assumes that a virtual method call may resolve either to its static target or to any method that (transitively) overrides it.

Algorithm 6 REMOVE METHOD(m : *Method*)

Input Language: Java
Output Language: Java

1: **assert** $(\texttt{uses}(m) \cup \texttt{calls}(m)) \setminus \texttt{below}(m) = \emptyset$
2: $o \leftarrow \{m' \mid m <: m'\}$
3: **if** $o \neq \emptyset \wedge \forall m' \in o.m'$ is abstract **then**
4: **for all** types B that inherit m **do**
5: MAKE TYPE ABSTRACT(B)
6: remove m

It is now quite easy to write out a specification for REMOVE METHOD (Alg. 6): The refactoring fails if there are any calls to m that do not originate in the set $\texttt{below}(m)$ of nodes inside the definition of m itself. Otherwise, it considers the set o of all methods m' that m overrides (which we write as $m <: m'$). If there is at least one such method, and furthermore they are all abstract[6], the host type of m (and any other type that inherits m without overriding it) will now inherit an abstract method and hence has to be made abstract itself.

When making a method m abstract (Alg. 7), we check that it cannot actually be called, and then make every type abstract that inherits m before putting an `abstract` modifier onto it. Of course, a type can only be made abstract (Alg. 8) if it is either an interface (and hence already abstract), or it is a class that is not instantiated anywhere.

Algorithm 7 MAKE METHOD ABSTRACT(m : *Method*)

Input Language: Java
Output Language: Java

1: **assert** $\texttt{calls}(m) \setminus \texttt{below}(m) = \emptyset$
2: **for all** types B that inherit m **do**
3: MAKE TYPE ABSTRACT(B)
4: make m `abstract`

Algorithm 8 MAKE TYPE ABSTRACT(T : *Type*)

Input Language: Java
Output Language: Java

1: **if** T is not an interface **then**
2: **assert** T is class and never instantiated
3: make T `abstract`

The specification of PUSH DOWN VIRTUAL METHOD, shown as Alg. 9, is now simply a matter of plugging together the previously specified microrefactorings. We use the operator **or** to indicate an alternative of several refactorings to try from left to right until the first one succeeds. ID, of course, is the identity refactoring that does nothing and never fails.

[6] Remember that there could be several such m', *e.g.* one from the superclass and the others from implemented interfaces.

Algorithm 9 PUSH DOWN VIRTUAL METHOD(m : *VirtualMethod*)

Input Language: Java
Output Language: Java ∪ locked names

1: **for all** types $B <:$ `hostType`(m) **do**
2: $c \leftarrow \lfloor$TRIVIALLY OVERRIDE$\rfloor(B, m)$
3: **if** $c \neq$ `None` **then**
4: INLINE METHOD(c)
5: REMOVE METHOD(m)
6: **or** MAKE METHOD ABSTRACT(m)
7: **or** ID()

Note that INLINE METHOD is invoked without floor brackets, so any locked names introduced by it are not unlocked immediately, which can be helpful if they refer to methods that are hidden by m.

Let us now briefly consider the case of non-virtual methods. There is probably not much sense in pushing down a **`private`** method, since it can only be referenced by members of the same class, and these references will be broken if the method is pushed into a child class. Static methods can be handled in almost the same way as virtual methods, except that instead of introducing an overriding method with a **`super`** call we introduce a hiding method that directly invokes the hidden method on the super class. The rest of the refactoring need not change.

One feature that this specification does not provide for is to push several inter-dependent methods at the same time, or indeed to determine which other methods have to be pushed together with the selected method. We leave this extension to future work.

4.2 MOVE MEMBER TYPE TO TOPLEVEL

We now turn to a type-level refactoring, MOVE MEMBER TYPE TO TOPLEVEL, which converts a member type that is nested within one or more other types to a toplevel type.

This refactoring, which incidentally must be performed by a Java compiler as part of the translation to bytecode, makes for an interesting example in our framework since it reuses a language extension that was originally introduced for a different refactoring: As explained in [SdM09], the MOVE METHOD and MAKE METHOD STATIC refactorings can be implemented rather easily by going through an enriched language featuring a JavaScript-like **`with`** construct.

In a nutshell, the compound statement

`with``(`e_n`, ..., `e_0`)`
s

should be understood to mean that statement s is executed with the value of e_0 as its zeroth enclosing instance (*i.e.*, the value of **`this`**), e_1 as the first enclosing instance, and so on. No enclosing instances beyond e_n are available.

Moving a static member type to the toplevel is easy, since it cannot access any non-static data of its enclosing types. So all we need to do is to ensure that names still bind to their intended declaration. This is the gist of the specification in Alg. 10.

Algorithm 10 MOVE MEMBER TYPE TO TOPLEVEL(M : *MemberType*)

Input Language: Java
Output Language: Java ∪ locked names

1: **if** M is not static **then**
2: $\lfloor$MAKE TYPE STATIC$\rfloor(M)$
3: $p \leftarrow$ `hostPkg`(M)
4: lock all names in M
5: remove M from its host type
6: INSERT TYPE(p, M)

We use an ancillary refactoring INSERT TYPE (Alg. 11) to actually insert type M into its target package p, which is very similar to INSERT FIELD: We need to check that p does not already contain a type or subpackage of the same name. Then we globally lock all accesses to types of the same name to ensure that they will not become shadowed. We remove modifiers that are inapplicable for toplevel types from T, and finally insert it into the syntax tree.

Algorithm 11 INSERT TYPE(p : *Package*, T : *ClassOrInterface*)

Input Language: Java
Output Language: Java ∪ locked names

1: **assert** no type or subpackage of same name as T in p
2: `lockNames(name(`T`))`
3: remove modifiers **`static`**, **`private`**, **`protected`** from T
4: insert T into p

Things are more complicated for non-**`static`** member types, since they can access enclosing instances. We use another refactoring MAKE TYPE STATIC to make the member type **`static`** first. Applied to a type M, this refactoring adds to M one field for each enclosing instance, say `this$i` for the ith enclosing instance, and surrounds the body of every constructor and method in B by

`with``(this$n, ..., this$1, `**`this`**`) { ... }`

The fields corresponding to the enclosing instances, of course, have to be properly initialised, which accounts for most of the complexity of the specification in Alg. 12. This initialisation has to happen in M's constructors, which therefore need to be equipped with additional parameters to pass in the enclosing instances, which again means that all calls to the constructors have to be updated to provide corresponding arguments.

Algorithm 12 MAKE TYPE STATIC(M : *MemberType*)

Input Language: Java
Output Language: Java ∪ `with`, locked names

```
1:  [A_n; ...; A_1] ← enclosing types of M
2:  for all i ∈ {1, ..., n} do
3:      f ← new field of type A_i with name this$i
4:      INSERT FIELD(M, f)
5:      for all constructors c of M do
6:          p ← parameter of type A_i with name this$i
7:          assert no parameter or variable this$i in c
8:          insert p as first parameter of c
9:          if c is chaining then
10:             add this$i as first argument of chaining call
11:         else
12:             a ← new assignment of p to f
13:             insert a after super constructor call
14: for all constructors c of M do
15:     for all non-chaining invocations u of c do
16:         es ← enclosing instances of u
17:         assert |es| = n
18:         insert es as initial arguments to u
19:         discard qualifier of u, if any
20: put modifier static on M
21: for all callables m of M do
22:     if m has a body then
23:         surround body of m by
            with(this$n, ..., this$1, this) {...}
```

Let us go through the specification step by step. We first have to introduce fields for the enclosing instances. Let $A_1, \ldots, A_n$ be the enclosing types of M; we consider each A_i in turn. First we create a field f to hold the enclosing instance, which has type A_i and name `this$i`, and insert it into M using the INSERT FIELD refactoring. We then create a corresponding parameter p for every constructor c of M. This parameter has the same type and name as f, and is added to the constructor's parameter list; the refactoring will fail if c has a parameter or local variable with name `this$i`.

We now need to make sure that every constructor initialises f to the value of its parameter p. This happens in one of two ways, depending on what kind of constructor c is. If c is a chaining constructor that invokes another constructor of the same class by means of a `this(...)` constructor invocation, we add p to the argument list of that call, and rely on the invoked constructor to initialise the field. Otherwise we insert an assignment from p to f into the body of c.

Now we have added the required instance fields to M and set up its constructors to initialise them, but of course we need to adjust all invocations of constructors of M to actually provide values for the extra parameters. So we again consider every constructor c of M and its every invocation u in turn. Since we have already treated `this(...)` calls in the previous step, we do not have to consider them again.

For any other invocation, we obtain the list es of expressions that provide values for the enclosing instances of M and add them as arguments to the call. If the invocation was qualified, then that qualifier was the single enclosing instance of the invocation which we have already added to the argument list and can hence discard.

Finally, we wrap the bodies of all constructors and methods of M (collectively referred to as *callables*) in `with` blocks, and give M a `static` modifier.

Notice that we did not wrap field initialisers (or indeed instance initialisers) of M into `with` blocks. Hence, if an initialiser contains a reference to a field f of an enclosing class A_i, that reference will simply be locked, but unless f is static it cannot be successfully unlocked after M is moved to the toplevel, so the refactoring will fail.

Eclipse and IntelliJ both treat variable uses in field initialisers the same as variable uses in methods, so in the example the reference will be qualified with the field holding the ith enclosing instance. This, however, is wrong: initialisers are evaluated before constructor bodies, so the field is not initialised yet and a null pointer exception will result at runtime.

Java compilers avoid this problem by inlining field initialisers into constructor bodies first, but this seems an overly drastic transformation for a refactoring engine to attempt. Likewise, our specification does not make any special arrangements for accessing private members of enclosing classes; unlocking references to such members will fail, thus aborting the whole refactoring. Eclipse tackles this issue by increasing the visibility of the referenced members, but, as shown by Steimann *et al.* [ST09], adjusting accessibility is quite a tricky problem in itself. A better solution might be to introduce getter and setter methods as is usually done by Java compilers.

5. Towards a Full-Featured Refactoring Engine

In order to show that the techniques introduced so far provide a sufficient basis for implementing a refactoring engine, we set ourselves the goal of implementing all the refactorings offered by Eclipse 3.5 and validating them against their publicly available test suite.

We decided to exclude the type-based refactorings EXTRACT INTERFACE, EXTRACT SUPERCLASS, GENERALIZE DECLARED TYPE, USE SUPERTYPE WHERE POSSIBLE, and INFER GENERIC TYPE ARGUMENTS, as these have been thoroughly explored in the literature [Tip07].

Also excluded from consideration was CHANGE METHOD SIGNATURE, since its implementations in Eclipse performs very few semantic checks. Various aspects of this refactoring, such as adding, removing, or permuting parameters can be implemented in a safe way using our framework, but in its full generality CHANGE METHOD SIGNATURE is per-

Refactoring	total # of tests	inap-plicable	missing feature	we reject	Eclipse rejects	same result	lines of code Eclipse	lines of code we
CONVERT ANONYMOUS TO NESTED	45	4	1	0	1	39	997	220
EXTRACT CLASS	24	2	1	1	1	19	760	243
EXTRACT CONSTANT	60	13	9	10	0	28	683	42
EXTRACT TEMP	133	1	28	5	1	98	854	107
INLINE CONSTANT	38	11	0	8	0	19	827	44
INLINE TEMP	55	12	5	2	1	35	381	82
INTRODUCE FACTORY	49	3	1	0	0	45	799	81
INTRODUCE INDIRECTION	31	0	1	5	0	25	933	61
INTRODUCE PARAMETER	20	1	1	5	0	13	448	26
INTRODUCE PARAMETER OBJECT	19	0	6	0	0	13	628	61
MOVE INNER TO TOPLEVEL	94	15	0	10	1	68	1427	125
MOVE INSTANCE METHOD	54	4	0	14	4	32	2038	99
MOVE MEMBERS	90	5	8	20	5	52	945	120
PROMOTE TEMP TO FIELD	55	19	14	0	0	22	829	62
PULL UP	143	36	5	0	1	101	1694	208
PUSH DOWN	95	27	10	4	1	53	872	374
SELF-ENCAPSULATE FIELD	36	0	3	0	0	33	751	85

Table 1. Evaluation of the correctness of our refactoring engine on Eclipse's test suite

haps best described as a general code transformation, not as a behaviour preserving refactoring.

Table 1 summarises the performance of our refactoring engine when applied to the internal test suite of Eclipse's refactoring engine, as available from the Eclipse source distribution. We exclude test cases for the various RENAME refactorings, results of which are reported in [SEdM08], and for EXTRACT METHOD and INLINE METHOD, which have been treated in [SVEdM09].

For the remaining 17 refactorings[7] the table shows the total number of test cases in the first column, and the number of test cases where our engine produced the same results as Eclipse, modulo trivial differences, in the sixth column. The other columns categorise sources of disagreement.

The second column lists the number of test cases that we could not test our engine on, either because they were disabled or because they were not valid Java programs. Since our refactoring engine is implemented as an extension to the JastAddJ compiler frontend, we can only process programs that successfully pass syntactic and semantic checks. We include in this category test cases whose expected result is arguably wrong, as well as tests that pertain to non-functional aspects of the Eclipse refactoring engine.

The third column shows the number of test cases where our implementation produces output programs that are correct and behave the same as the input program, but where we do not perform all the refactoring steps that Eclipse does. Many of the failures here relate to clone detection: for example, the EXTRACT CONSTANT and EXTRACT TEMP refactorings in Eclipse extract all copies of the selected expression, whereas our engine only extracts the selected one. While an *ad hoc* extension of our engine could presumably refactor most or all of these test cases, we leave the implementation of a principled clone detector and its integration with the refactoring process to future work.

Further, columns four and five summarise the test cases where one engine produced results while the other rejected the refactoring. The latter gives the number of test cases where Eclipse rejects a refactoring which our engine is able to perform. Conversely, the former tallies the number of spurious rejections by our engine.

For the refactorings INLINE CONSTANT and INTRODUCE PARAMETER, these rejections are generally due to the quite conservative dataflow analysis of our implementation. In some cases a more precise analysis would be able to prove that the refactoring can go ahead. Another possible solution would be to push this kind of issue into the UI, reporting failures of flow preservation to the user as warnings instead of aborting the refactoring outright. In any case, Eclipse's current hands-off approach in which almost no analysis is performed seems quite unsatisfactory.

Another major source of rejection, affecting INTRODUCE INDIRECTION, MOVE INNER TO TOPLEVEL and MOVE INSTANCE METHOD, is visibility adjustment. When moving members between types, it is sometimes necessary to increase the visibility of referenced members for them to remain accessible after the refactoring. Eclipse has some heuristics for doing this, but as shown by Steimann and Thies [ST09] these heuristics are rather crude and can easily lead to subtle changes in behaviour.

[7] Space restrictions prevent us from presenting detailed specifications of these refactorings; they are, however, available as part of the distribution of our refactoring engine.

Our refactoring engine does not attempt to solve this problem. The locked naming framework does, however, check that names only bind to accessible declarations, and aborts the refactoring if this is not the case. Hence our implementation never produces output programs that violate accessibility rules, although it may on occasion reject refactorings that could be performed if the visibility were adjusted. To improve on this, we plan to integrate Steimann and Thies' system of accessibility constraints with our name unlocking mechanism.

In summary, while there is certainly a lot of disagreement in detail between our refactoring engine and Eclipse, we think that the results show that our techniques are powerful enough to be applied to the implementation of just about any refactoring and can produce high-quality implementations with comparatively little effort.

This is brought into sharp relief if we compare the source code size of the two implementations, given in the last two columns.[8]. For Eclipse, we measured the size of the `*Refactoring.java` or `*Processor.java` files, respectively, for each refactoring. These files only form the core of the implementation, and contain neither shared utility code nor user-interface related functionality. For our implementation, we give the size of the refactoring implementation including all microrefactorings, but again excluding utility code. As can be seen, our implementations are always more compact, in some cases dramatically so. This is partly due to JastAdd's aspect-oriented features that enable us to much more cleanly separate the essence of the refactoring implementation from supporting code.

Adding up the line counts in the last column, one would obtain a size of about 2100 lines of code for the core of our implementation. This would, however, count reused microrefactorings once for every refactoring that uses them. The actual amount of code, counting every refactoring only once, is only slightly more than 1300 lines, which attests to the reusability of the microrefactorings.

This tally does not include the naming framework, at about 1400 lines of code, as well as the flow dependency framework, at about 300 lines, both of which have already been discussed in previous papers. A further 800 lines provide functionality for introducing and removing the language restrictions and extensions used by the refactorings.

Pleasingly, no further major language extensions besides **`with`** and anonymous methods are needed for the above refactorings. The five other extensions provided are all very simple ones, like the first-class array initialisers mentioned in the introduction. As for language restrictions, we have mentioned restrictions on compound variable declarations, the **`synchronized`** modifier and implicit method modifiers. Two other restrictions of a similar kind are used by other refactorings and are likewise quite easy to realise.

[8] This data was generated using David A. Wheeler's 'SLOCCount'.

In total, the complete source code of our refactoring engine, including all of the above as well as supporting and utility code (but excluding the JastAddJ frontend and its control flow analysis package, which are independent projects) amounts to about 6250 lines of code. This is just a little more than the combined size of the core implementations of the four largest refactorings of Table 1 in Eclipse.

We do, however, want to point out two major weaknesses of our refactoring engine in comparison with Eclipse: It is currently not integrated into an IDE, and hence not usable for developers, and it is somewhat lacking in performance. Refactorings on even very small programs can take up to five seconds, which is perhaps acceptable for a prototype implementation but excludes the possibility of using our engine in a production environment. This is partly due to the general approach of decomposing refactorings into smaller units, but a more important factor is the choice of implementation language.

While attribute grammar systems like JastAdd excel at tasks involving the computation of information on immutable trees, as is typical in compiler frontends, updates to the syntax tree, as they are performed by refactorings, necessitate frequent flushing and recomputation of cached information. While in some cases it is possible to pinpoint precisely which attributes need to be invalidated, this is a Sisyphean task in general and can lead to extremely subtle bugs. Hence our implementation elects to always flush all attributes whenever the syntax tree has changed, which incurs a heavy performance penalty. A principled solution to this problem requires significant engineering, which we leave to future work.

6. Related Work

This work is inspired by a long line of work on the specification and implementation of refactorings. The classical work in this area is surely Opdyke's thesis [Opd92], which provides a catalogue of refactorings for C++. While the actual transformations are described only informally, the preconditions that ensure behaviour preservation are spelled out in more detail, although only for a restricted subset of the object language. It is not clear how easy the given conditions would be to implement.

Another important source of inspiration is Griswold's thesis [Gri91]. In contrast to Opdyke, he does not present his refactorings primarily in terms of preconditions, but rather views them in terms of their effects on the program dependence graph (PDG), presenting them as the composition of meaning-preserving PDG transformations. This is, of course, very close to our approach emphasising dependency preservation and the decomposition of larger refactorings into smaller microrefactorings. The object language for which Griswold describes his refactorings is a first-order subset of Scheme. The choice of such a simple language with

such a simple and regular syntax greatly simplifies the presentation of his refactorings.

The composition of smaller refactorings into larger ones is also the focus of work by Kniesel and Koch [KK04]. Their work is concerned with composing the pre- and post-conditions of constituent refactorings, which does not directly apply to our dependency based presentation.

There has been some interest in providing executable specifications of refactorings. For instance, Lämmel [Läm02] shows how refactorings can be implemented very concisely by means of rewriting strategies, with the syntax tree represented as a generic data type that abstracts away from peculiarities of the object language. It is, however, unclear if these techniques are usable beyond the very simple refactorings and simple object languages discussed in the paper.

Rewriting is also the technique used by Garrido and Meseguer [GM06], who give executable specifications of several Java refactorings in the rewrite logic-based system Maude. Again, however, the refactorings covered are of a very simple and local nature.

The JunGL language proposed by Verbaere [Ver08] is specifically aimed at making it easy to implement refactorings. The language's unique combination of functional and logic programming features, in particular the concept of path queries, are well-suited for specifying the kind of static semantic analyses needed for refactorings. Many refactorings can be described elegantly in JunGL [VEdM06], and even complex type-based refactorings are within reach of the language [Pay06].

Recently, a series of papers by Tip and others [TKB03, BTF05, Tip07, KETF07] have examined a number of type-based refactorings such as EXTRACT INTERFACE and INFER GENERIC TYPE ARGUMENTS. They make pervasive use of type constraints to ensure behaviour preservation, but while the constraint rules are given in great detail, the actual refactorings are only described informally. The implementation of type-based refactorings in Eclipse is based on this work.

Steimann and Thies [ST09] introduce accessibility constraints that can be used to systematically adjust the visibility of program elements that are no longer accessible due to code movement. Again, the constraint rules are presented formally, whereas the refactorings are only given prose descriptions.

Both lines of work use similar concepts of constraints that can be seen as a natural extension of our dependency edges, which are just single constraints to be solved in isolation. Layering a more advanced constraint solving mechanism that can take accessibility and type constraints into account on top of this existing framework is an intriguing perspective.

Kegel and Steimann [KS08] give a very thorough precondition-based description of the REPLACE INHERITANCE WITH DELEGATION refactoring that tries to account for all features of the Java language. Being given in prose only, the description is somewhat verbose and hard to grasp. Despite all their efforts, the authors report that "whenever we believed that we had made correctness of the refactoring plausible, testing it on a new project revealed a new problem we had not previously thought of", and refrain from arguing for the correctness of their specification.

The only work that we are aware of which actually presents pseudocode specifications of refactorings for Java is the recent paper by Wloka *et al.* on refactoring programs for reentrancy [WST09], which describes a transformation for wrapping static state into `ThreadLocal` fields. Their specifications concentrate on issues relevant to the proposed refactoring, and ignore possible problems arising, *e.g.*, from naming conflicts.

Recent work by Soetens [Soe09] takes the other direction: instead of giving a specification and then using it as the basis for an implementation, he takes the implementations of PULL UP METHOD, ENCAPSULATE FIELD and EXTRACT CLASS in Eclipse, and reverse-engineers their specification, in particular their preconditions.

The problem of how to efficiently compute attribute values without introducing stale values is a mainstay of the attribute grammar literature, and there is a large amount of work on incremental evaluation of attributes that handles this issue transparently. The classic work of Demers, Reps, and Teitelbaum [DRT81] shows how to solve this problem for traditional attribute grammars without reference attributes or circular attributes by computing dependencies and propagating changes along them. Work by Pennings and others [PSV92, SSK00] concentrates on caching attribute values, but again for traditional attribute grammars without reference attributes.

The doctoral thesis of Maddox [Mad97] presents an incremental attribute grammar system with some more advanced features, although the eclectic nature of the system makes it hard to judge how much of the results would apply to JastAdd. More recently, Boyland has proposed an incremental evaluation mechanism for remote attribute grammars [Boy02], which are somewhat similar to reference attribute grammars. These two papers might make a good starting point for exploring the incremental evaluation of JastAdd attribute grammars, although the power of the system (which allows arbitrary Java code in attribute equations) makes it unrealistic to expect that a fully automated approach would work well in practice.

On the other end of the spectrum, Acar and others have recently published a series of papers on self-adjusting computation [AAB08], which generalises the problem of incremental evaluation to a much broader class of programs, including stateful computations. It seems, however, that their approach incurs a certain performance penalty as well, and it is unclear whether it would scale to a system the size of JastAddJ, on which our refactoring engine is based.

7. Conclusions

We have presented a case study showing the feasibility of a new approach to the implementation of refactorings, which is based on the concepts of dependency preservation, language extensions, and microrefactorings.

We have shown that this approach allows us to give clean, modular specifications of important refactorings that are precise enough to serve as the basis of a reimplementation of the majority of refactorings offered in recent versions of Eclipse. We have tested our implementation on Eclipse's own extensive test suite, and presented the results showing that our implementation is on par with industrial-strength refactoring engines with regard to features, and often surpasses them with regard to correctness.

These results notwithstanding, our present work does *not* claim to give completely bug-free specifications that account for all corner cases. While this is our ultimate goal which we believe to have made good progress towards achieving, some cases may very well not be covered by the current specifications.

But one of the advantages of modular specifications of refactorings is exactly this: a refactoring can be understood part by part, and we can reason about its correctness in a modular way. If a bug is found, we do not need to content ourselves with a half-baked fix in one particular implementation that may exploit some idiosyncrasies of the system to provide an ill-fitting bandaid. Instead we can amend the high-level specification of the refactoring, so that every implementation based on it can reap the benefits.

A. Definitions

A.1 Pseudocode Conventions

We give our specifications in generic, imperative pseudocode. Parameters and return values are informally typed, with syntax tree nodes having one of the types from Fig. 7. Additionally, we use an ML-like `option` type with constructors `None` and `Some` for functions that may or may not return a value.

Where convenient, we make use of ML-like lists, with list literals of the form $[1; 2; 3]$ and $|xs|$ indicating the length of list xs.

The names of refactorings are written in SMALL CAPS, whereas utility functions appear in `monospace`. A list of utility functions with brief descriptions is given in Fig. 8. An invocation of a refactoring is written with floor-brackets $\lfloor$LIKE THIS$\rfloor()$ to indicate that any language extensions used in the output program produced by the refactoring should be eliminated before proceeding.

We write $A <: B$ to mean that type A extends or implements type B, and $m <: m'$ to mean that method m overrides method m'.

Node Type	Description
ClassOrInterface	either a class or an interface; is a *Type*
Field	field declaration
LocalVar	local variable declaration
MemberType	type declared inside another type; is a *Type*
Method	method declaration
MethodCall	method call
Package	package
Type	type declaration
VirtualMethod	non-`private` instance method; is a *Method*

Figure 7. Node Types

Name	Description
`below`(n)	returns the set of all nodes below n in the syntax tree
`calls`(m)	returns all calls that may dynamically resolve to method m; can be a conservative over-approximation
`hostPkg`(e)	returns the package of the compilation unit containing e
`hostType`(e)	returns the closest enclosing type declaration around e
`lockNames`(n)	locks all names anywhere in the program that refer to a declaration with name n
`name`(e)	returns the name of program entity e
`uses`(m)	returns all calls that statically bind to method m

Figure 8. Utility Functions

References

[AAB08] Umut A. Acar, Amal Ahmed, and Matthias Blume. Imperative Self-Adjusting Computation. In *POPL*, pages 309–322, 2008.

[Boy02] John Boyland. Incremental Evaluators for Remote Attribute Grammars. *Electr. Notes Theor. Comput. Sci.*, 65(3), 2002.

[BTF05] Ittai Balaban, Frank Tip, and Robert Fuhrer. Refactoring support for class library migration. In *OOPSLA*, pages 265–279, 2005.

[DRT81] A. Demers, T. Reps, and T. Teitelbaum. Incremental Evaluation for Attribute Grammars with Applications to Syntax-directed Editors. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 105–116. ACM Press, 1981.

[EH07a] Torbjörn Ekman and Görel Hedin. The JastAdd Extensible Java Compiler. In *OOPSLA*, 2007.

[EH07b] Torbjörn Ekman and Görel Hedin. The JastAdd system - modular extensible compiler construction. *Sci. Comput. Program.*, 69(1-3):14–26, 2007.

[Fou10] Eclipse Foundation. Eclipse 3.5. `http://www.eclipse.org`, 2010.

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

[GM06] Alejandra Garrido and José Meseguer. Formal Specification and Verification of Java Refactorings. In *SCAM*, 2006.

[Gri91] William G. Griswold. *Program Restructuring as an Aid to Software Maintenance*. Ph.D. thesis, University of Washington, 1991.

[Jet10] JetBrains. IntelliJ IDEA 9.0.1. `http://www.jetbrains.com`, 2010.

[Ker05] Joshua Kerievsky. *Refactoring to Patterns*. Addison Wesley, 2005.

[KETF07] Adam Kiezun, Michael D. Ernst, Frank Tip, and Robert M. Fuhrer. Refactoring for Parameterizing Java Classes. In *ICSE*, pages 437–446, 2007.

[KK04] Günter Kniesel and Helge Koch. Static Composition of Refactorings. *The Science of Computer Programming*, 52(1-3):9–51, 2004.

[KS08] Hannes Kegel and Friedrich Steimann. Systematically Refactoring Inheritance to Delegation in Java. In *International Conference on Software Engineering (ICSE)*, pages 431–440. ACM Press, 2008.

[Läm02] Ralf Lämmel. Towards Generic Refactoring. In *Rule-based Programming (RULE)*, pages 15–28. ACM Press, 2002.

[Mad97] William Harry Maddox. *Incremental Static Semantic Analysis*. Ph.D. thesis, University of California, Berkeley, 1997.

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

[Pay06] Arnaud Payement. Type-based Refactoring using JunGL. Master's thesis, Oxford University Computing Laboratory, 2006.

[PSV92] M. Pennings, D. Swierstra, and H. Vogt. Using cached functions and constructors for incremental attribute evaluation. In *PLILP*, pages 130–144. Springer Verlag, 1992.

[SdM09] Max Schäfer and Oege de Moor. Of Gnats and Dragons – Sources of Complexity in Implementing Refactorings. In *WRT*, 2009.

[SDS+10] Max Schäfer, Julian Dolby, Manu Sridharan, Frank Tip, and Emina Torlak. Correct Refactoring of Concurrent Java Code. In Theo D'Hondt, editor, *ECOOP*, 2010.

[SEdM08] Max Schäfer, Torbjörn Ekman, and Oege de Moor. Sound and Extensible Renaming for Java. In Gregor Kiczales, editor, *OOPSLA*. ACM Press, 2008.

[SEEV10] Max Schäfer, Torbjörn Ekman, Ran Ettinger, and Mathieu Verbaere. Refactoring bugs. `http://progtools.comlab.ox.ac.uk/projects/refactoring/bugreports`, 2010.

[Soe09] Quinten David Soetens. Formalizing Refactorings Implemented in Eclipse. Master's thesis, University of Antwerp, 2009.

[SSK00] J. Saraiva, S. D. Swierstra, and M. F. Kuiper. Functional incremental attribute evaluation. In *Compiler Construction*, pages 279–294, 2000.

[ST09] Friedrich Steimann and Andreas Thies. From Public to Private to Absent: Refactoring Java Programs under Constrained Accessibility. In Sophia Drossopoulou, editor, *ECOOP*, 2009.

[SVEdM09] Max Schäfer, Mathieu Verbaere, Torbjörn Ekman, and Oege de Moor. Stepping Stones over the Refactoring Rubicon – Lightweight Language Extensions to Easily Realise Refactorings. In Sophia Drossopoulou, editor, *ECOOP*, 2009.

[Tip07] Frank Tip. Refactoring Using Type Constraints. In *SAS*, pages 1–17, 2007.

[TKB03] Frank Tip, Adam Kiezun, and Dirk Bäumer. Refactoring for Generalization using Type Constraints. In *OOPSLA*, pages 13–26, 2003.

[VEdM06] Mathieu Verbaere, Ran Ettinger, and Oege de Moor. JunGL: a Scripting Language for Refactoring. In Dieter Rombach and Mary Lou Soffa, editors, *International Conference on Software Engineering (ICSE)*, pages 172–181. ACM Press, 2006.

[Ver08] Mathieu Verbaere. *A Language to Script Refactoring Transformations*. D.Phil. thesis, Oxford University Computing Laboratory, 2008.

[WST09] Jan Wloka, Manu Sridharan, and Frank Tip. Refactoring for Reentrancy. In *ESEC/SIGSOFT FSE*, pages 173–182, 2009.